ОГЛАВЛЕНИЕ

Введение

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ JAVA

Это же вам не лезгинка, а твист!

Из к/ф «Кавказская пленница»

Среди языков программирования Java — самый популярный и самый востребованный. Эта книга о том, как программировать на Java.

Особенности языка Java

Тот, кто нам мешает, тот нам и поможет.

Из к/ф «Кавказская пленница»

История Java началась в 1990-х годах, когда группа инженеров компании Sun Microsystems в рамках проекта под названием Green приступила к разработке достаточно универсального, компактного и платформенно-независимого языка программирования Oak, предназначенного для использования в бытовых устройствах. В процессе реализации проекта изменились не только основные приоритеты, но и название языка программирования. Как бы то ни было, в 1995 году мир познакомился с языком программирования Java.

НА ЗАМЕТКУ

С появления первой версии Java было несколько обновлений платформы. На момент написания книги актуальной является версия Java 8. Именно она обсуждается в книге.

Язык Java, хоть и не без труда, но завоевал свое «место под солнцем». Сегодня Java прочно удерживает позиции самого востребованного языка программирования. Успеху языка способствовало бурное развитие интернет-технологий. Дело в том, что для Java-программ характерна

высокая степень универсальности и независимости от аппаратного обеспечения. Это важно при создании программ, ориентированных на работу в Сети, поскольку конечные пользователи используют различные операционные системы и оборудование. К тому же важную роль сыграла применимость Java для программирования всевозможных мобильных устройств. Поэтому нет ничего удивительного, что значительная доля коммерческих и свободно распространяемых программ написана на языке Java. Соответственно, спрос на программистов, работающих с языком Java, стабильно высок, а общие тенденции таковы, что он останется высоким и в ближайшее время.

Универсальность программ, написанных на языке Java, базируется на использовании *виртуальной машины*. Это такой специфический «посредник», под управлением которого выполняется байт-код, получаемый при компиляции программы. Здесь нужны пояснения.

После того как программа написана, она компилируется. Обычно в результате компиляции программы создается исполнительный файл с машинным кодом, который и выполняется, когда необходимо выполнить программу. Проще говоря, при компиляции команды, понятные для программиста, переводятся на «язык», понятный для компьютера. Если речь идет о программе, написанной на языке Java, то все происходит похожим образом, но с некоторыми особенностями. Самое важное, что в результате компиляции Java-программы получается не машинный код, а промежуточный *байт-код*. Это нечто среднее между машинным кодом и кодом программы. Если машинный код, как правило, выполняется под управлением операционной системы, то байт-код выполняется под управлением специальной программы, которая называется виртуальной машиной (или виртуальной Java-машиной). Понятно, что такую программу на компьютер предварительно следует установить.

Возникает вопрос: а в чем же выигрыш от использования виртуальной машины и как все описанное влияет на универсальность кодов? Выигрыш в том, что при написании кода можно абстрагироваться от особенностей операционной системы и аппаратного обеспечения, используемых конечным пользователем. Эти особенности учитываются — но учитываются на уровне виртуальной машины. Именно виртуальная машина при выполнении байт-кода «принимает в расчет» особенности операционной системы и аппаратного обеспечения компьютера, на котором выполняется программа.

НА ЗАМЕТКУ

Допустим, есть программа, написанная на языке C++. При ее компиляции получается машинный код, который для разных операционных систем будет разным. Если компилируется программа, написанная на Java, то получающийся в результате байт-код не зависит от операционной системы, которая установлена на компьютере, — он будет одним и тем же для разных операционных систем. Но вот виртуальная машина для каждой операционной системы своя. Разница в операционных системах «учитывается», когда на компьютер устанавливается виртуальная машина.

Описанный выше механизм, в общем и целом, обеспечивает высокую степень универсальности программ, написанных на Java. Особенно это заметно при создании программ с графическим интерфейсом.

НА ЗАМЕТКУ

Забегая вперед, отметим, что в плане создания приложений с графическим интерфейсом язык Java особенно хорош.

Есть еще один важный аспект, касающийся языка Java, на который сразу хочется обратить внимание. Язык Java — полностью *объектно-ориентированный* язык. Сказанное означает, что для написания даже самой маленькой и самой простой программы придется описать по меньшей мере один *класс*. Это автоматически создает некоторые трудности в освоении премудростей Java. Особенно сложно тем, кто не имеет опыта программирования. Ведь фактически сразу, с первых шагов, приходится знакомиться с концепцией *объектно-ориентированного программирования* (сокращенно ООП), которая, надо признать, не самая тривиальная. Но паниковать не стоит — мы найдем способ донести нужные сведения даже до самых неподготовленных читателей. Главное, чтобы было желание освоить язык Java.

Программное обеспечение

Будь проклят тот день, когда я сел за баранку этого пылесоса!

Из к/ф «Кавказская пленница»

Если подойти к вопросу формально, то сам по себе язык Java — набор правил, в соответствии с которыми составляется программный код.

Но программы пишутся для того, чтобы они выполнялись. А раз так, то нам понадобится специальное программное обеспечение. Хорошая новость в том, что все необходимое программное обеспечение может быть получено совершенно свободно, просто, легально и бесплатно.

> **НА ЗАМЕТКУ**
>
> Понятно, что есть и коммерческие приложения, предназначенные для написания программ в Java. Но для решения тех задач, которые мы ставим перед собой, стандартного свободно распространяемого программного обеспечения более чем достаточно.

Что же нам понадобится? В принципе, можно обойтись минимальными средствами в виде пакета приложений JDK (сокращение от *Java Development Kit* — средства разработки Java). В состав пакета JDK, кроме прочего, входит компилятор, всевозможные библиотеки, документация и исполнительная система JRE (сокращение от *Java Runtime Environment* — среда выполнения Java) — фактически виртуальная машина Java. Пакет приложений JDK распространяется бесплатно компанией Oracle (сайт компании www.oracle.com).

> **НА ЗАМЕТКУ**
>
> В свое время разработчика Java, компанию Sun Microsystems, поглотила корпорация Oracle. Так что теперь поддержкой Java-технологий занимается именно она.

Ситуация такая, что без JDK нам не обойтись, но и ограничиваться только пакетом JDK не стоит. Если ограничиться только пакетом JDK, то программные коды придется набирать в текстовом редакторе, а компилировать программу придется «вручную» из командной строки. Поэтому желательно использовать *среду разработки* (сокращенно IDE от *Integrated Development Environment*).

Среда разработки содержит редактор кодов, отладчик, позволяющий в интерактивном режиме отслеживать код на наличие синтаксических ошибок, набор прочих утилит, позволяющих сделать процесс написания, тестирования и компиляции программ простым, удобным и где-то даже комфортным (насколько это вообще возможно). Проще говоря, среда разработки должна использоваться — тем более, если учесть, что имеются очень приличные бесплатно распространяемые среды разработки.

Мы остановим свой выбор на среде разработки NetBeans. Среда распространяется бесплатно, ее установочные файлы можно загрузить на сайте поддержки проекта www.netbeans.org.

Далее кратко рассмотрим, какое программное обеспечение и откуда следует загрузить перед тем, как мы непосредственно приступим к изучению языка программирования Java.

Задача наша простая:

- загрузить и установить пакет приложений JDK;
- после установки JDK следует загрузить и установить среду разработки NetBeans.

Действия по загрузке и установке программного обеспечения выполняются именно в том порядке, как они перечислены выше.

ДЕТАЛИ

Среда разработки NetBeans в процессе работы с программными кодами обращается к системе JDK. Если систему JDK установить до установки NetBeans, то все настройки среды разработки, связанные с JDK, выполняются автоматически. Если систему JDK устанавливать после установки среды разработки NetBeans, то настройки среды разработки придется выполнять самостоятельно.

Итак, в первую очередь устанавливаем пакет JDK, для чего предварительно с сайта компании Oracle загружаем установочные файлы. На рис. В.1 показано окно браузера, открытое на странице www.oracle.com.

В разделе загрузок (вкладка **Downloads**) следует найти ссылку на загрузку программного обеспечения для Java.

ДЕТАЛИ

Существует несколько редакций, или дистрибутивов, Java. Например, платформа Java для создания программного обеспечения уровня больших корпораций называется Java Enterprise Edition (сокращенно Java EE). Стандартная редакция Java предназначена для создания пользовательских приложений и называется Java Standard Edition (сокращенно Java SE). Также существует редакция Java Micro Edition (сокращенно Java ME), используемая при создании приложений для всевозможных мобильных устройств. Мы будем использовать стандартную редакцию Java Standard Edition (или Java SE).

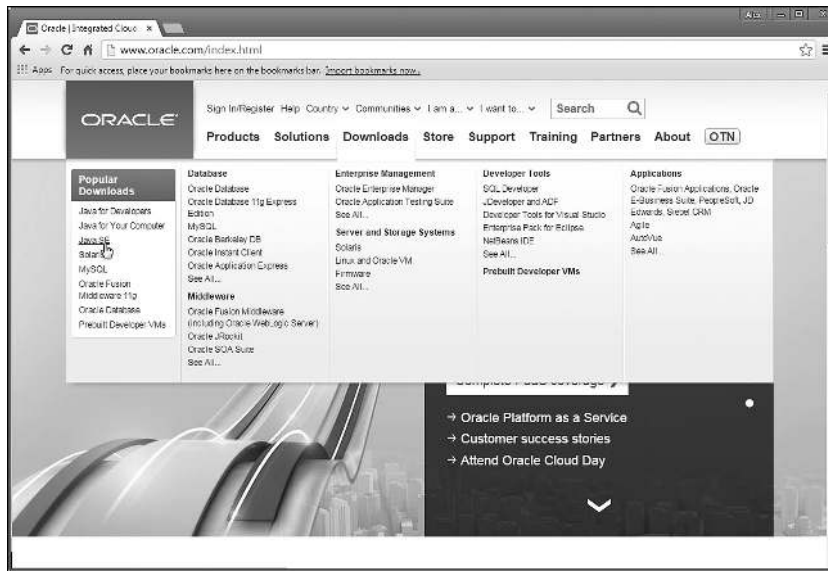


Рис. В.1. Окно браузера открыто на странице www.oracle.com корпорации Oracle

После щелчка по гиперссылке для загрузки программного обеспечения для работы с Java, переходим на еще одну страницу, с которой собственно и выполняется загрузка (рис. В.2).

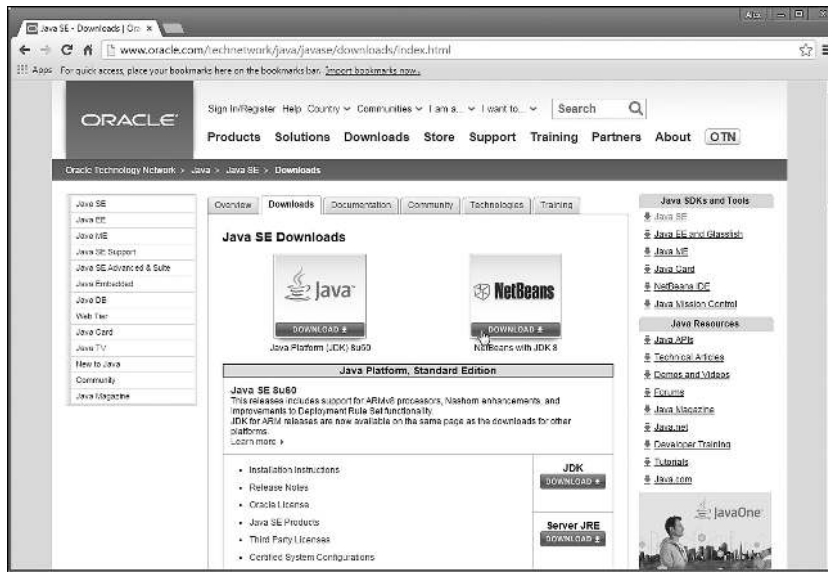


Рис. В.2. Окно браузера открыто на странице загрузки установочного файла пакета JDK и среды разработки NetBeans

В принципе здесь можно просто загрузить пакет JDK, но обычно предлагается еще и способ загрузки, при котором пакет JDK идет в комплекте со средой разработки NetBeans. Это, пожалуй, лучший вариант, который позволяет последовательно установить на компьютер JDK и NetBeans из одного установочного файла.

В процессе загрузки предлагается выбрать тип установочного файла в соответствии с используемой операционной системой. Ситуация проиллюстрирована на рис. В.3.

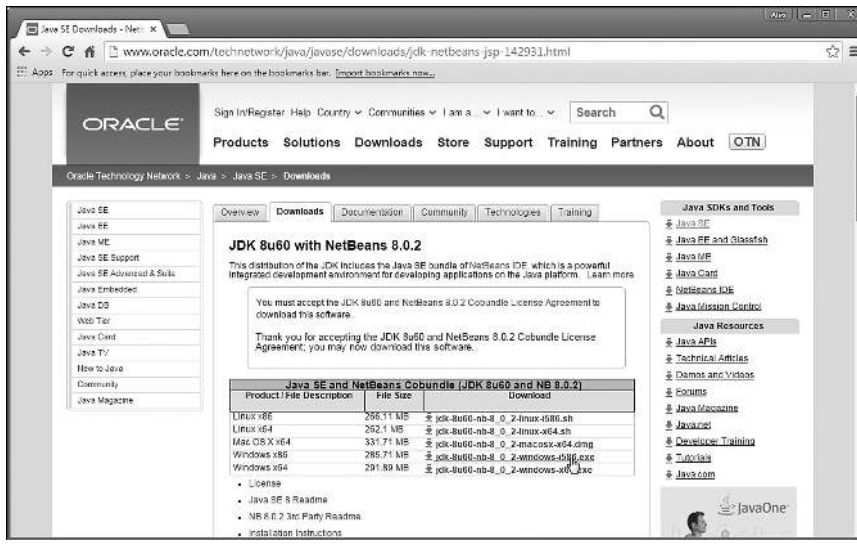


Рис. В.3. Выбор установочного файла в соответствии с используемой операционной системой

НА ЗАМЕТКУ

Внешний вид сайтов, в том числе и сайт корпорации Oracle, время от времени меняется, поэтому не исключено, что для поиска страницы загрузки программного обеспечения придется проявить некоторую изобретательность.

Если со страницы корпорации Oracle загружается установочный файл сразу для JDK и NetBeans, то все, что остается — выполнить установку. Выполняется она просто, так что комментировать здесь особо нечего (со всеми предлагаемыми в процессе установки настройками лучше согласиться). Если же по каким-то причинам загружается и устанавливается

только пакет JDK, то придется отдельно загрузить еще и установочный файл для среды разработки NetBeans. В этом случае переходим на страницу www.netbeans.org проекта NetBeans, как показано на рис. В.4.

Рис. В.4. Страница www.netbeans.org проекта NetBeans

Затем переходим к странице загрузки установочных файлов среды NetBeans, на которой следует выбрать версию среды для загрузки (рис. В.5).

Разные версии среды разработки отличаются, кроме языка интерфейса, поддерживаемыми технологиями (сюда включаются разные редакции платформы Java и еще несколько дополнительных языков программирования). Версия должна быть такой, чтобы в ней поддерживалась редакция Java SE. Если возможности аппаратного обеспечения позволяют, можно порекомендовать версию среды разработки с максимальным набором поддерживаемых технологий.

После выбора версии среды разработки NetBeans загружается установочный файл, после чего устанавливается среда разработки. На этом предварительная подготовка к написанию программ на Java завершается. Заметим лишь, что еще одна полезная страница находится по адресу www.java.com. На рис. В.6 показано окно браузера, открытое на данной странице.

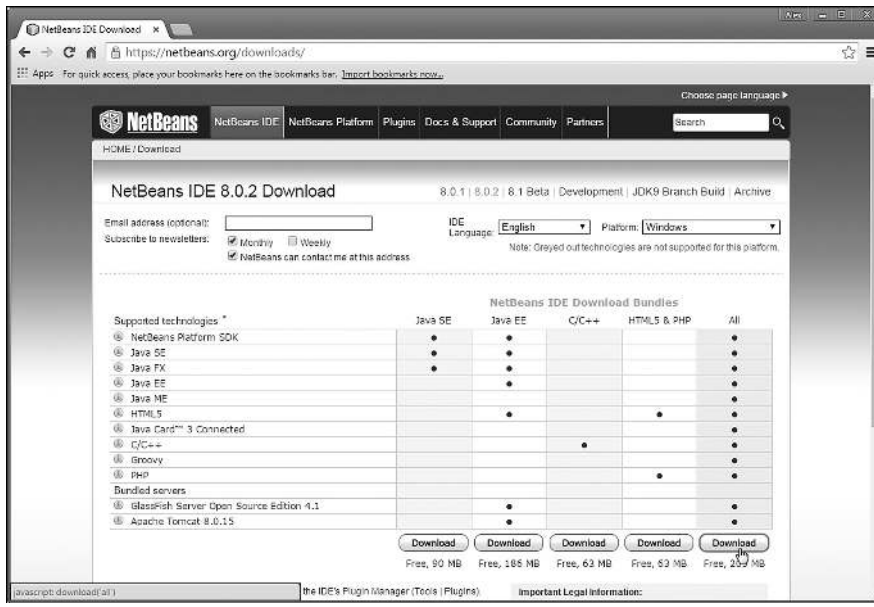


Рис. В.5. Выбор версии среды разработки NetBeans

Рис. В.6. Окно браузера открыто на странице www.java.com поддержки Java

На странице можно загружать обновления платформы Java, которые появляются достаточно часто.

Среда разработки NetBeans

Замечательная идея! Что ж она мне самому в голову не пришла?

Из к/ф «Ирония судьбы, или С легким паром!»

Все проекты (программы), рассматриваемые в книге, тестировались в среде разработки NetBeans. Читателю рекомендуется использовать эту же среду разработки (с другой стороны, не все рекомендации обязательны к исполнению).

В любом случае, уместно уделить хоть небольшое внимание приложению NetBeans.

На рис. В.7 показано, как выглядит пустое окно (без открытого в нем проекта) среды разработки NetBeans.

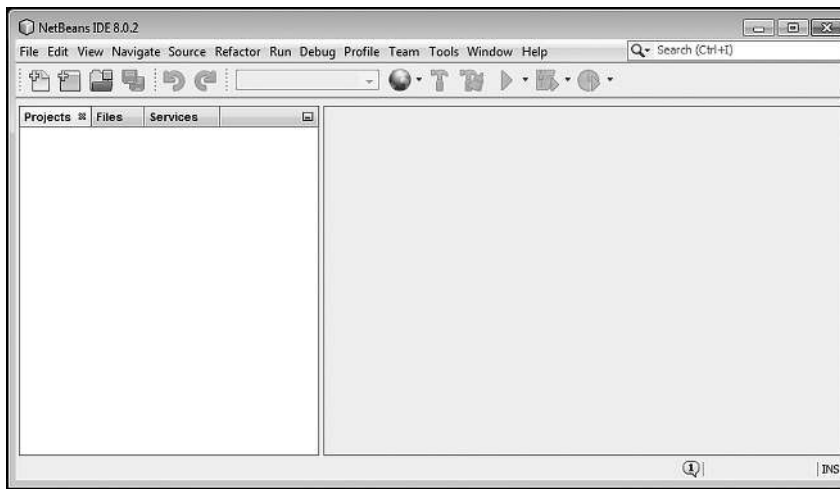


Рис. В.7. Внешний вид окна среды разработки NetBeans

Особенности работы с приложением NetBeans могут быть предметом отдельной книги. В наши планы это не входит. Мы рассмотрим лишь некоторые наиболее важные операции, которые читателю предстоит выполнять в процессе написания программных кодов.

Чтобы не отвлекаться в основной части книги на описание манипуляций с элементами интерфейса приложения NetBeans, сделаем это заблаговременно. А именно интерес представляют такие действия:

- создание нового проекта (выполняется при написании новой программы);
- компиляция и запуск на выполнение программы;
- закрытие открытого проекта (программы);
- открытие уже существующего проекта.

Теперь обо всем по порядку.

Создание нового проекта

Создание новой программы означает создание нового проекта. Чтобы создать новый проект, в меню **File** выбираем команду **New Project**, как показано на рис. В.8.

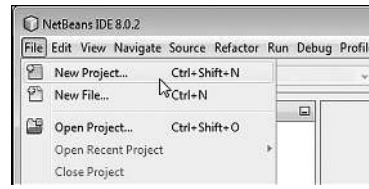


Рис. В.8. Создание проекта с помощью команды **New Project** из меню **File**

Альтернативный способ создания проекта — щелкнуть кнопку с желтой папкой и зеленым знаком «плюс» на панели инструментов, как показано на рис. В.9, или нажать комбинацию клавиш <Ctrl>+<Shift>+<N>.

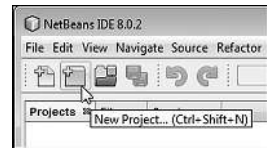


Рис. В.9. Для создания нового проекта выполняется щелчок на кнопке на панели инструментов

На следующем этапе открывается диалоговое окно **New Project**, в котором есть два раздела: **Categories** и **Projects** (рис. В.10).

В разделе **Categories** выбираем позицию **Java**, а в разделе **Projects** выбираем позицию **Java Application**, после чего щелкаем кнопку **Next**. В результате откроется диалоговое окно **New Java Application**, представленное на рис. В.11.

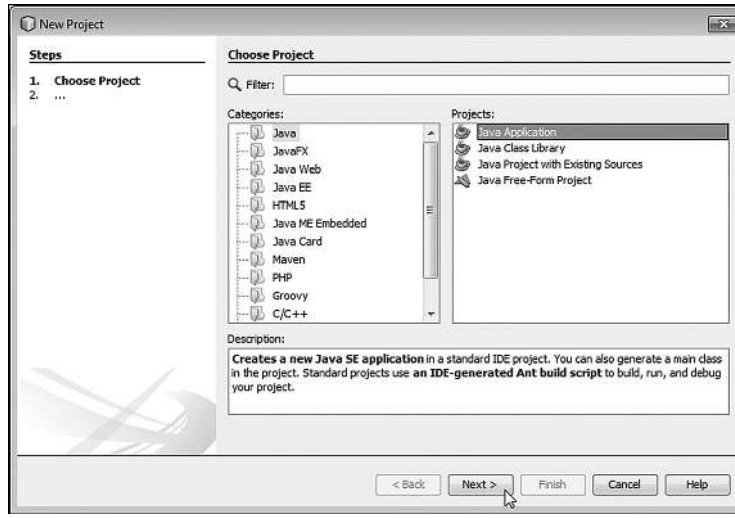


Рис. В.10. Выбор типа проекта в окне **New Project**

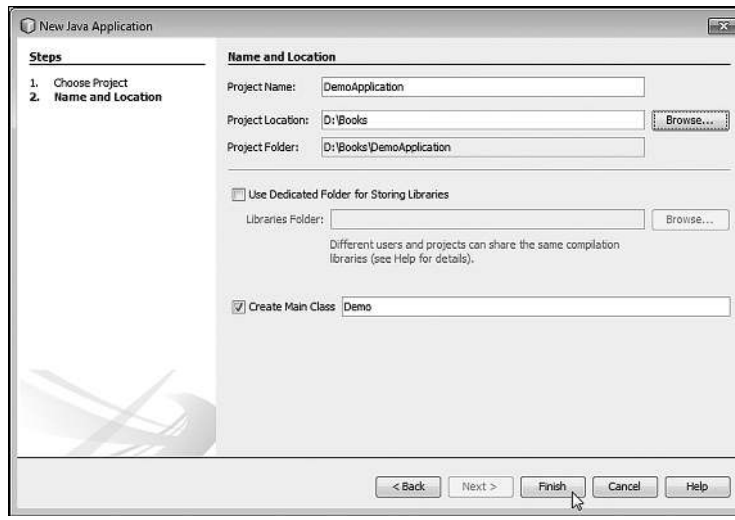


Рис. В.11. Определение основных параметров проекта в окне **New Java Application**

В окне **New Java Application** выполняются такие настройки приложения, как его название и место хранения файлов приложения. В частности, в поле **Project Location** указывается каталог, в котором хранятся файлы проекта. Для удобства выбора имеется кнопка **Browse**. Щелчок на кнопке приводит к отображению диалогового окна выбора каталога. В поле **Project Name** задается название проекта (здесь мы назвали

проект DemoApplication). Поле **Project Folder** заполняется автоматически, а вот поле **Create Main Class** лучше заполнить: поле активно при установленном флажке в опции слева от названия поля, а в самом поле вводится название для главного класса программы (в рассматриваемом примере главный класс называется Demo).

ДЕТАЛИ

Как отмечалось ранее, любая программа на языке Java содержит описание хотя бы одного класса. Если речь не идет об апплетах, то среди классов программы есть один, который называется главным классом программы. Класс содержит код главного метода, и этот код выполняется при выполнении программы. Проще говоря, главный класс программы содержит код, выполняемый при запуске программы. В поле **Create Main Class** указывается название для главного класса программы. Вообще, название этого класса задается непосредственно в программном коде, но для компиляции программы оно должно совпадать с именем, указанным для главного класса при создании проекта.

По умолчанию в поле **Create Main Class** предлагается текстовая строка в формате имя.имя (имя, точка, затем еще одно имя). Если оставить в поле такой «точечный» текст, то первое имя (до точки) является именем пакета, а второе имя (после точки) является именем главного класса. Если так, то код программы следует начинать с инструкции package, после которой указывается имя пакета (первое имя перед точкой). Например, если бы в поле **Create Main Class** было указано mypack.Demo, то код программы должен был бы начинаться инструкцией package mypack (заканчивается точкой с запятой). Мы будем обсуждать работу с пакетами, но немного позже. Пока же рекомендуется в поле **Create Main Class** просто указать имя главного класса.

После щелчка на кнопке **Finish** создание нового проекта завершено. На рис. В.12 показано окно среды разработки с открытым в нем вновь созданным проектом.

В правой части окна среды разработки отображается внутреннее окно редактора кодов с кодом главного класса (окно называется Demo.java). В новом проекте отображается шаблонный код, который на самом деле обычно удаляется, и вместо него вводится тот код, который нужен.

НА ЗАМЕТКУ

Если по каким-то причинам код главного класса не отображается, в левой верхней части окна среды разработки находим внутреннее

окно с названием **Projects**. В нем есть раскрывающийся пункт с названием приложения. Если раскрыть этот пункт, можно увидеть позицию с названием файла Demo.java для главного класса. Двойной щелчок на названии Demo.java приводит к отображению кода главного класса.

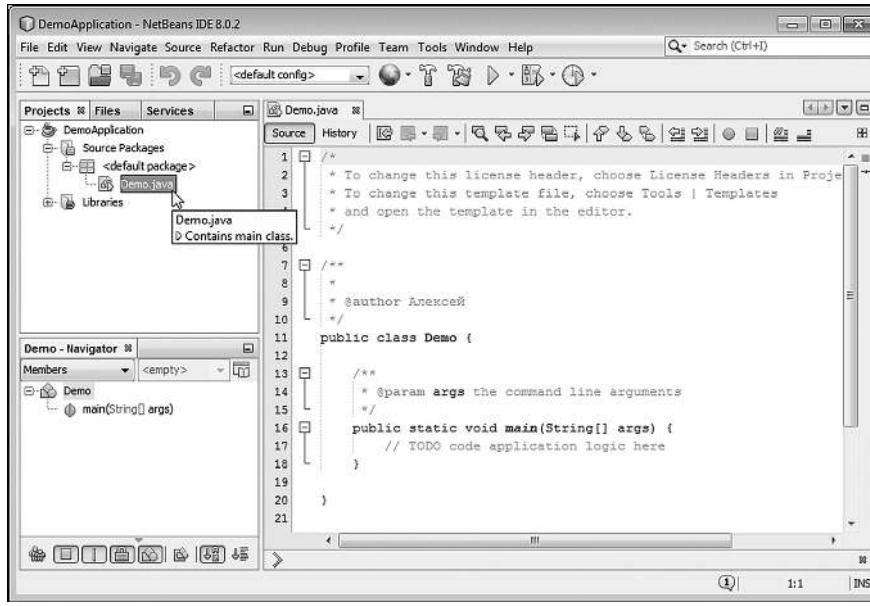


Рис. В.12. Окно среды разработки NetBeans с открытым новым проектом

Осталось набрать код программы, откомпилировать его и запустить на выполнение.

Компиляция и запуск программы на выполнение

Для тестирования функциональности приложения NetBeans в редакторе кодов вводим очень простой код, который есть в листинге В.1.

Листинг В.1. Программный код проекта DemoApplication

```
class Demo{
  public static void main(String[] args){
    System.out.println("Java & NetBeans");
  }
}
```

Как выглядит окно приложения NetBeans с введенным кодом в редакторе кодов, показано на рис. В.13.

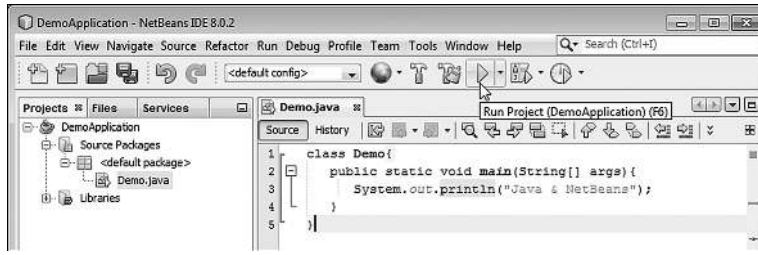


Рис. В.13. Компиляция программы и запуск ее на выполнение

ДЕТАЛИ

На данном этапе смысл команд из приведенного выше программного кода не так уж и важен. Анализировать коды мы будем в основной части книги. Пока же отметим, что инструкцией `class Demo` начинается описание класса. Само описание заключается в фигурные скобки { и }. В классе описывается главный метод программы, который называется `main()`. В теле метода выполняется всего одна команда `System.out.println("Java & NetBeans")`, которой в окне вывода отображается сообщение `Java & NetBeans`.

Для компиляции программы и запуска ее на выполнение щелкаем кнопку с зеленой стрелкой на панели инструментов (см. рис. В.13).

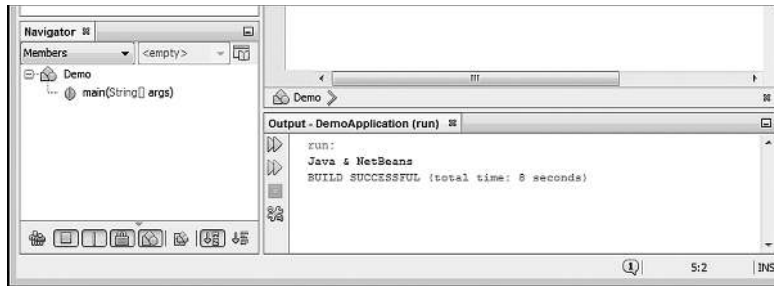


Рис. В.14. Результат выполнения программы отображается во внутреннем окне **Output**

Если компиляция пройдет удачно, в нижней части окна среды разработки во внутреннем окне **Output** появляется сообщение `Java & NetBeans`, как это показано на рис. В.14.

Проще говоря, результат выполнения программы такой:

Результат выполнения программы (из листинга В.1)

```
Java & NetBeans
```

Заметим, что откомпилировать и запустить программу на выполнение также можно с помощью команды **Run Project** из меню **Run** (рис. В.15) или нажав клавишу F6.

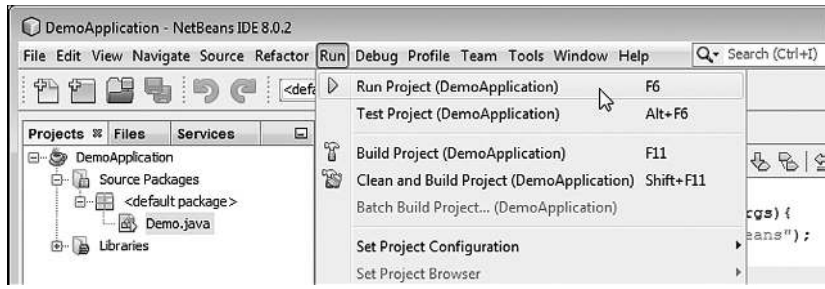


Рис. В.15. Альтернативный способ компиляции и запуска программы на выполнение с помощью команды **Run Project** из меню **Run**

ДЕТАЛИ

Файл с Java-программой имеет расширение .java. В общем случае программа, написанная на языке Java, может содержать несколько классов. При компиляции для каждого класса создается отдельный файл. Название таких файлов совпадает с названиями соответствующих классов, а расширения у файлов .class.

Закрытие проекта

Чтобы закрыть проект, достаточно его выделить во внутреннем окне **Projects**, щелкнуть правой кнопкой мыши и в открывшемся контекстном меню выбрать команду **Close**, как показано на рис. В.16.

Еще один способ состоит в том, чтобы при выделенном проекте (в окне **Projects**) в меню **File** выбрать команду **Close Project** (рис. В.17).

Если нужно закрыть все открытые проекты, можем воспользоваться командой **Close All Projects** (см. рис. В.17).

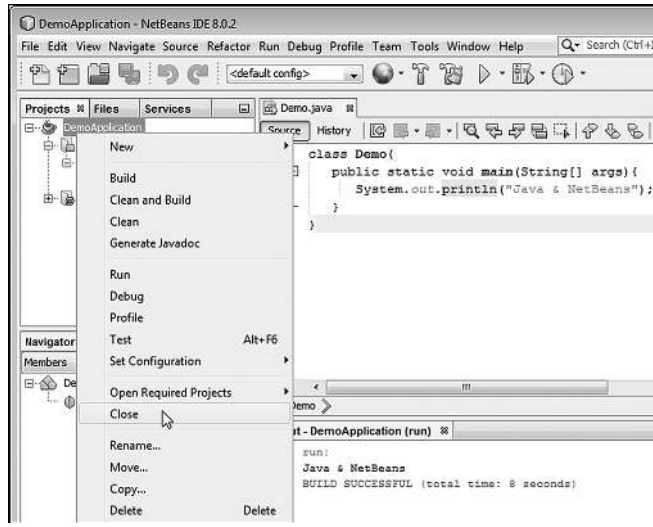


Рис. В.16. Выбор команды **Close** в контекстном меню проекта для его закрытия

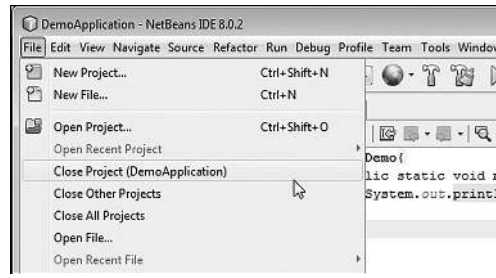


Рис. В.17. Закрытие проекта с помощью команды **Close Project** из меню **File**

Открытие существующего проекта

Иногда необходимо открыть проект, который был создан ранее. В таком случае полезным станет подменю **Open Recent Project** из меню **File**, которое содержит названия недавно открывавшихся проектов (рис. В.18).

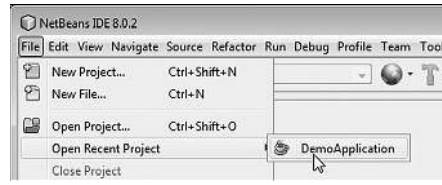


Рис. В.18. Выбор проекта для открытия в списке подменю **Open Recent Project** из меню **File**

Если в меню **File** выбрать команду **Open Project** (рис. В.19), откроется диалоговое окно, в котором выбирается папка, содержащая интересующий нас проект.

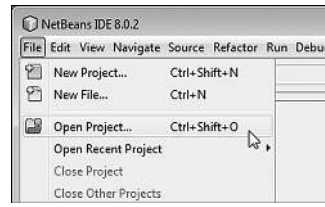


Рис. В.19. Открытие проекта с помощью команды **Open Project** из меню **File**

Наконец, можно воспользоваться специальной кнопкой на панели инструментов, как показано на рис. В.20.

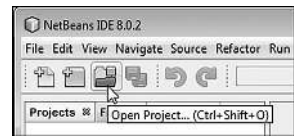


Рис. В.20. Для открытия проекта выполняется щелчок по кнопке на панели инструментов

Еще один способ открыть проект — нажать комбинацию клавиш Ctrl+Shift+O.

О книге

Бывают такие случаи, когда неплохо и соврать.
Из к/ф «Ирония судьбы, или С легким паром!»

Несколько слов хочется сказать собственно о книге. Как отмечалось в самом начале, книга о том, как программировать на Java. Мы начнем с самых простых вещей и постепенно рассмотрим практически все основные темы, которые так или иначе формируют парадигму программирования на Java. Далее приводится список некоторых тем, которые рассматриваются в книге:

- базовые приемы создания программ на Java;
- классы и объекты;

- перегрузка методов;
- лямбда-выражения;
- наследование и переопределение методов;
- использование интерфейсов;
- обработка исключительных ситуаций;
- многопоточное программирование;
- обобщенные классы;
- создание приложений с графическим интерфейсом;
- создание апплетов.

Список не является исчерпывающим, так что, помимо перечисленного выше, книга содержит обсуждение многих иных вопросов и механизмов.

Для удобства усвоения материала книга разбита на относительно небольшие главы. Каждая глава содержит примеры решения различных задач. В конце каждой главы есть краткое резюме, облегчающее усвоение материала главы. Хочется верить, что книга станет надежным помощником для всех, кто изучает язык программирования, в том числе и для тех, кто осваивает премудрости Java самостоятельно.

Обратная связь с автором

Если он явится еще раз, то подожжет дом.

Из к/ф «Ирония судьбы, или С легким паром!»

Автор книги — *Алексей Николаевич Васильев*, профессор кафедры теоретической физики Киевского национального университета имени Тараса Шевченко. Автор книг по программированию и математическому моделированию. Более подробную информацию можно найти на сайте www.vasilev.kiev.ua. Вопросы и замечания можно направлять по электронной почте alex@vasilev.kiev.ua.

Глава 1
ПРИСТУПАЕМ К ПРОГРАММИРОВАНИЮ

— Куда вы меня несете?
— Навстречу твоему счастью.
Из к/ф «Ирония судьбы, или С легким паром!»

Мы приступаем к изучению языка программирования Java. Первое, что мы сделаем, — напишем небольшую программу. И хотя мы еще практически ничего не узнали о языке Java, написать первую программу нам это не помешает.

Первая программа

— Ну ладно, куда ехать?
— Понятия не имею.
Из к/ф «Ирония судьбы, или С легким паром!»

Написание программы начинается с определения задачи, которая должна быть решена, или цели, которая должна быть достигнута при ее выполнении. Цель у нас простая: при выполнении программы должно появляться диалоговое окно с текстовым сообщением. Такая задача в Java решается исключительно просто. Понадобится всего несколько строчек кода.

Создание программы

Поскольку речь идет о первом проекте, мы поступим так: сначала создадим программу, посмотрим, как она выполняется, а уже затем проанализируем программный код.

Мы используем программный код, представленный в листинге 1.1.

Листинг 1.1. Программный код проекта ShowMeAWindowApplication

```
import javax.swing.JOptionPane;
class ShowMeAWindowDemo{
  public static void main(String[] args){
    JOptionPane.showMessageDialog(null,"Первая программа на Java!");
  }
}
```

Код совсем небольшой. Необходимо создать новый проект и ввести в окно редактора программный код из листинга 1.1. Как при этом может выглядеть окно среды разработки NetBeans с кодом программы, показано на рис. 1.1.

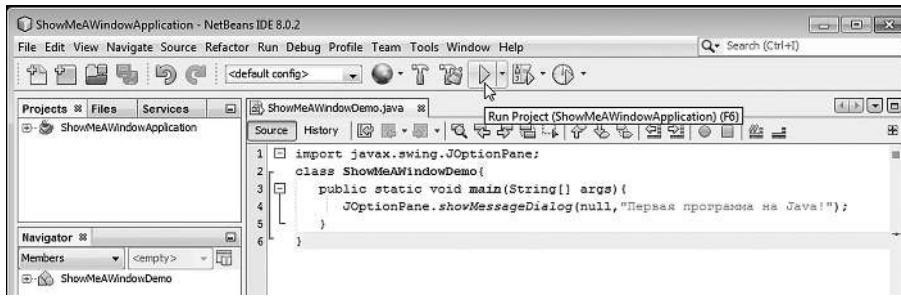


Рис. 1.1. Окно среды разработки NetBeans с кодом программы

ДЕТАЛИ

Напомним алгоритм создания нового приложения в среде разработки NetBeans. Для начала в меню **File** выбираем команду **New Project**. В окне **New Project** в разделе **Categories** выбираем пункт **Java**, а в разделе **Projects** выбираем пункт **Java Application**. В окне **New Java Application** в поле **Project Name** указываем название ShowMeAWindowApplication для имени проекта, в поле **Project Location** задаем место сохранения проекта, а в поле **Create Main Class** указываем название ShowMeAWindowDemo для главного класса (он же единственный), который создается в программе.

Для компиляции и запуска программы на выполнение щелкаем на панели инструментов по кнопке с зеленой стрелкой (см. рис. 1.1) или выбираем в меню **Run** команду **Run Project**. В результате начинает

выполняться программа и на экране появляется диалоговое окно, как на рис. 1.2.

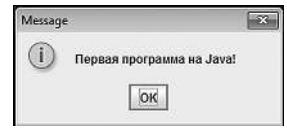


Рис. 1.2. В результате выполнения программы отображается диалоговое окно с сообщением

Окно называется **Message** (название окна отображается слева вверху в строке названия), содержит информационную пиктограмму (круг с буквой i внутри), текст Первая программа на Java!, под которым расположена кнопка **ОК**. При щелчке по кнопке **ОК** или системной пиктограмме (с крестиком) в правом верхнем углу окна оно закрывается, а программа прекращает выполнение.

Анализ программного кода

Теперь проанализируем код программы. Самая первая инструкция import javax.swing.JOptionPane необходима для использования в программе класса JOptionPane из библиотеки Swing.

НА ЗАМЕТКУ

Библиотека Swing содержит набор классов для разработки приложений с графическим интерфейсом. Является неотъемлемой частью платформы Java. С библиотекой Swing мы познакомимся поблизе немного позже, когда будем рассматривать создание приложений с графическим интерфейсом.

Дело в том, что диалоговое окно в нашей программе отображается с помощью метода showMessageDialog(). Это статический метод класса JOptionPane — проще говоря, метод описан в данном классе, и говорить о методе showMessageDialog() вне контекста класса JOptionPane просто нет смысла.

ДЕТАЛИ

При вызове метода выполняется определенный блок программного кода или набор команд. В данном случае неважно даже, какие конкретно команды выполняются при вызове метода showMessageDialog(),

а важен результат — создается и отображается диалоговое окно. Мы просто используем уже существующий метод, созданный не нами.

Методы бывают статическими и обычными, то есть не статическими. Если метод обычный (не статический), то он вызывается из объекта. Нас такие методы пока не интересуют. Статические методы вызываются из класса, в котором они описаны. Проще говоря, при вызове статического метода необходимо указать класс, в котором описан метод. Метод showMessageDialog() описан в классе JOptionPane. Класс JOptionPane в свою очередь входит в состав библиотеки Swing.

В общем случае инструкция import используется для импорта (то есть добавления) в программу всевозможных утилит — обычно классов. При этом после import-инструкции указывается не просто имя импортируемого класса, а полный «путь» к нему, отображающий иерархию пакетов, вплоть до того «места», в котором хранится класс.

ДЕТАЛИ

В Java используется система распределения классов по пакетам. Идея состоит в том, что в пределах пакета имена у всех входящих в пакет классов уникальные (разные), но если классы находятся в разных пакетах, то их имена могут и совпадать. Последствия такие: каждый класс хранится в каком-то пакете. Причем пакет может сам входить в другой пакет, и так далее (пакет, входящий в другой пакет, называется подпакетом). В итоге получается иерархия пакетов. Когда в import-инструкции указывается имя импортируемого класса, то вместе с именем класса указывается и вся иерархическая цепочка пакетов.

В частности, выражение javax.swing.JOptionPane означает, что класс JOptionPane входит в пакет swing, который является подпакетом пакета javax.

Весь остальной код программы — описание класса ShowMeAWindowDemo. Описание класса начинается с ключевого слова class, после которого следует имя класса. Выражение class ShowMeAWindowDemo, например, означает, что описывается класс с названием ShowMeAWindowDemo.

НА ЗАМЕТКУ

Название класса совпадает с именем, которое мы ввели в поле **Create Main Class** при создании проекта. В программе может описываться и использоваться несколько классов, но среди них

обязательно (если речь не идет об апплетах) есть главный класс (в нем описывается главный метод программы). Имя главного класса совпадает с именем, указанным в поле **Create Main Class**. В данном случае программа содержит описание (явное) лишь одного класса, он же является главным классом.

Собственно описание класса содержится в фигурных скобках (между открывающей скобкой { и закрывающей скобкой }). В классе содержится только главный метод программы. Метод называется main(). Перед названием метода указаны ключевые слова.

- Инструкция public означает, что метод открытый, и доступ к нему существует и вне пределов класса. Если учесть, что выполнение программы — это выполнение кода метода main(), то доступность метода вне класса вполне логична.
- Ключевое слово static означает, что метод статичный, и для его вызова нет необходимости создавать объект (метод вызывается из класса). Это тоже вполне логично, поскольку для создания объекта нужно запустить программу на выполнение, для чего следует вызвать метод main(). Если бы метод main() не был статичным, для его вызова пришлось бы создавать объект, для чего необходимо запустить программу, ну и так далее — получается замкнутый круг.
- Ключевое слово void означает, что метод main() не возвращает результат. Здесь тоже все разумно, поскольку результат просто некуда возвращать.

НА ЗАМЕТКУ

Методы могут возвращать результат. Если метод возвращает некоторое значение в качестве результата, то инструкцию с вызовом метода можно использовать в каком-то выражении, отождествляя ее с возвращаемым результатом. Есть методы, которые не возвращают результат. При вызове таких методов просто выполняется определенная последовательность команд (каких именно — зависит от описания метода).

После имени главного метода main() в круглых скобках указана инструкция String[] args. Данной инструкцией описывается аргумент args метода main(), представляющий собой текстовый массив. Мы аргумент метода main() в программном коде не используем, тем не менее по правилам языка Java аргумент главного метода все равно должен быть описан.

ДЕТАЛИ

У метода могут быть аргументы — значения, которые передаются методу при вызове и от которых зависит результат выполнения метода. При вызове метода аргументы через запятую указываются в круглых скобках после имени метода в том порядке, как они объявлены в описании метода. Если у метода нет аргументов, то при вызове метода после его имени указываются пустые круглые скобки.

Что касается главного метода программы, то у него есть аргументы (точнее, один аргумент, но он является текстовым массивом — набором текстовых значений). Дело в том, что при запуске программы на выполнение ей могут передаваться параметры. Именно такие параметры, которые передаются в программу при запуске, отождествляются с текстовым массивом, являющимся аргументом метода main(). Аргументы главного метода используются не очень часто. Мы тоже не планируем их использовать. Тем не менее стандарт описания метода main() предусматривает и описание аргумента метода.

Ключевое слово String является названием класса, через объекты которого в Java реализуются текстовые значения. Объекты мы обсудим позже, а сейчас отождествим идентификатор String со значением текстового типа. Пустые квадратные скобки [] после ключевого слова String свидетельствуют о том, что речь идет о массиве. Название аргумента главного метода произвольно (но традиционно его называют args).

Еще одна пара фигурных скобок (открывающая { и закрывающая }) определяет тело главного метода программы. Там всего одна команда JOptionPane.showMessageDialog(null,"Первая программа на Java!"), которой, собственно и отображается диалоговое окно.

НА ЗАМЕТКУ

В конце команды стоит точка с запятой. Все команды в языке Java заканчиваются точкой с запятой.

Команда JOptionPane.showMessageDialog(null,"Первая программа на Java!") представляет собой вызов метода showMessageDialog(). Метод вызывается из класса JOptionPane. Первым аргументом методу передается ключевое слово null, означающее пустую ссылку. В общем случае первым аргументом методу showMessageDialog() передается ссылка на родительское окно (окно верхнего

уровня по отношению к открывающемуся окну). Но поскольку в данном случае родительского окна попросту нет, то использовано стандартное ключевое null, означающее отсутствие родительского окна.

Вторым аргументом методу showMessageDialog() передается текст "Первая программа на Java!" Именно он отображается в диалоговом окне, появляющемся на экране при выполнении программы.

> **НА ЗАМЕТКУ**
>
> Текстовое значение в Java заключается в двойные кавычки. Выражение в двойных кавычках называется текстовым литералом.

Общие замечания

Таким образом, мы использовали общий подход, который кратко сводится к следующему.

- Программа состоит из описания класса.
- Класс, в свою очередь, содержит описание главного метода программы.
- При выполнении главного метода программы вызывается статический метод showMessageDialog() из класса JOptionPane.
- Для импорта в программу класса JOptionPane используем import-инструкцию.

Рассмотренный нами пример дает определенное формальное правило для написания программ на Java. В частности, базовым является следующий шаблон кода (ключевые элементы шаблона выделены жирным шрифтом):

```
class имя_класса{
  public static void main(String[] args){
    // Программный код
  }
}
```

При написании программы нам как минимум необходимо описать класс (главный класс программы, его название такое же, как имя, указанное

в поле **Create Main Class** диалогового окна **New Java Application** при создании проекта). В классе описывается метод main(), перед его именем указываются атрибуты public, static и void, а в круглых скобках после имени метода — выражение вида String[] args. Блоки программного кода (тело класса и тело метода) выделяются фигурными скобками. Это, так сказать, обязательная программа. Если в программе необходимо использовать библиотечные классы, наподобие класса JOptionPane, в начале программы используем import-инструкцию.

НА ЗАМЕТКУ

В программных кодах используются *комментарии*. Комментарии игнорируются при компиляции и предназначены для пояснения программного кода. Существует три вида комментариев.

Однострочные комментарии начинаются с двойной косой черты //. Все, что находится справа от двойной косой черты //, является комментарием. Однострочный комментарий располагается в одной строке.

Многострочный комментарий начинается с инструкции /* и заканчивается инструкцией */. Все, что находится между инструкциями /* и */, является комментарием. Такой комментарий может размещаться в нескольких строках.

Кроме однострочных и многострочных комментариев, есть еще и комментарии документирования. Они начинаются с инструкции /** и заканчиваются инструкцией */. Такие комментарии используются для автоматического генерирования содержимого в справочных HTML-документах.

Вариации на тему первой программы

Разве может быть запрограммированное, ожидаемое, запланированное счастье?

Из к/ф «Ирония судьбы, или С легким паром!»

В рассмотренной выше программе мы определяли только текст, отображаемый в диалоговом окне. Существует возможность задать, помимо текста, еще и название окна, а также тип пиктограммы. Поможет нам все тот же статический метод showMessageDialog() из класса JOptionPane. На этот раз мы просто передадим ему немного больше аргументов.

Рассмотрим программный код, представленный в листинге 1.2.

Листинг 1.2. Программный код проекта ShowMeNewWindowApplication

```
// Импортируется класс JOptionPane:
import javax.swing.JOptionPane;
// Описание класса ShowMeNewWindowDemo:
class ShowMeNewWindowDemo{
  // Описание главного метода программы:
  public static void main(String[] args){
    // Текст для названия окна:
    String title="Сообщение";
    // Текст сообщения:
    String text="Продолжаем изучать Java";
    // Отображение диалогового окна с сообщением:
    JOptionPane.showMessageDialog(null,text,title,JOptionPane.WARNING_MESSAGE);
  } // Завершение описания метода
}   // Завершение описания класса
```

Концептуально код в листинге 1.2 очень схож с кодом из листинга 1.1. Но, разумеется, имеются некоторые отличия. Одно из отличий декоративного, так сказать, характера, состоит в том, что в листинге 1.2 использованы комментарии. Еще один принципиально новый для нас момент связан с использованием текстовых переменных: командой `String title="Сообщение"` объявляется переменная `title` с текстовым значением `"Сообщение"`, а командой `String text="Продолжаем изучать Java"` объявляется переменная `text` со значением `"Продолжаем изучать Java"`.

ДЕТАЛИ

Переменная представляет собой имя, которое связано с некоторым объектом или обозначает область в памяти, в которую можно заносить значение и считывать значение оттуда. Объявляется переменная просто: указывается тип переменной и ее имя. Тип переменной определяется с помощью специального идентификатора типа.

При объявлении переменной под нее выделяется в памяти место. После объявления переменной ее можно использовать в программном коде. В рассматриваемом примере мы создаем две переменные (`title` и `text`), которым значениями присваиваются текстовые литералы, поэтому переменные объявляются как относящиеся к типу `String`. Справедливости ради следует отметить, что `String` — это имя

класса. Соответственно, переменные title и text являются так называемыми *объектными переменными*, которые в реальности ссылаются на объект. Но такие технические подробности пока что не существенны, и мы временно будем интерпретировать переменные title и text как содержащие текст.

Еще одно новшество связано с тем, что в команде JOptionPane.showMessageDialog(null,text,title,JOptionPane.WARNING_MESSAGE) методу showMessageDialog() передается не два, как ранее, а четыре аргумента.

- Первым аргументом методу передано значение null, указывающее на то, что у открываемого окна родительского окна нет.
- Вторым аргументом методу передается переменная text, значение которой определяет текст, отображаемый в диалоговом окне.
- Третий аргумент метода title (значение данной переменной) определяет название окна, отображаемое в строке названия.
- Четвертым аргументом методу передается выражение JOptionPane.WARNING_MESSAGE. Это статическая целочисленная константа, определяющая тип окна (а если более конкретно, то тип пиктограммы, отображаемой в области окна). Константа WARNING_MESSAGE означает, что в окне отображается «предупреждающая» пиктограмма (восклицательный знак внутри желтого треугольника), а окно, соответственно, называется окном предупреждения.

На рис. 1.3 показано окно, которое отображается при выполнении программы.

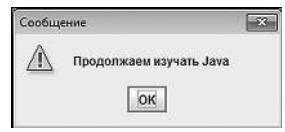


Рис. 1.3. При выполнении программы отображается окно предупреждения **Сообщение** (с соответствующей пиктограммой) и текстом в области окна

ДЕТАЛИ

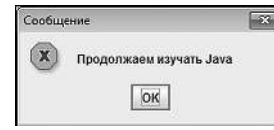
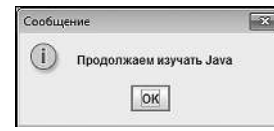
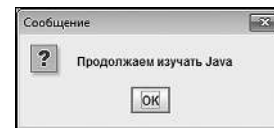
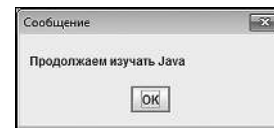
В классе JOptionPane существуют следующие статические константы, определяющие тип (и, соответственно, пиктограмму) окна: ERROR_MESSAGE (окно ошибки), INFORMATION_MESSAGE (информационное сообщение, данный тип используется по умолчанию), WARNING_MESSAGE (окно предупреждения), QUESTION_MESSAGE (окно с вопросом) и PLAIN_MESSAGE

(окно без пиктограммы). Каждая константа указывается вместе с именем класса JOptionPane: после имени класса через точку указывается имя константы.

Константа аналогична переменной, но с поправкой на то, что значение константы изменить нельзя. Статическая константа (статическое постоянное поле) описывается в классе и при обращении к такой константе вне программного кода класса перед именем переменной указывается имя класса. Имя класса и имя переменной разделяется точкой.

Если вместо константы WARNING_MESSAGE в команде JOptionPane.showMessageDialog (null,text,title,JOptionPane.WARNING_MESSAGE) указать другую константу, отображаемое окно изменит свой вид. Желающие могут проделать такой эксперимент. В табл. 1.1 представлены возможные варианты диалоговых окон, соответствующих значению константы, которая (вместе с именем класса JOptionPane) передается четвертым аргументом методу showMessageDialog().

Табл. 1.1. Примеры диалоговых окон разных типов

Константа	Окно
ERROR_MESSAGE	
INFORMATION_MESSAGE	
QUESTION_MESSAGE	
PLAIN_MESSAGE	

На будущее также заметим, что существует возможность не только использовать в диалоговом окне стандартные пиктограммы, но и определять пользовательские, создаваемые на основе файла с изображением. Этот вопрос мы рассмотрим немного позже.

Вывод в консольное окно

Я могу пронизать пространство и уйти в прошлое.

Из к/ф «Иван Васильевич меняет профессию»

В некоторых случаях информацию удобнее выводить не в диалоговое, а в консольное окно (если используется среда разработки NetBeans, то вывод осуществляется во внутреннее окно вывода). В таком случае нам метод showMessageDialog() из класса JOptionPane не нужен, а вместо него используем метод println().

Метод вызывается из объекта out, связанного со стандартным потоком вывода (по умолчанию — в консольное окно) и являющегося полем класса System. Фраза может быть не очень понятной, но здесь важно усвоить, что для вызова метода println() используется конструкция вида System.out.println(). Аргументом методу println() передается текстовое значение, которое отображается в консольном окне.

Как пример такого подхода рассмотрим листинг 1.3, в котором сообщение выводится не в диалоговое окно, а в консоль (точнее, внутреннее окно вывода среды разработки NetBeans).

Листинг 1.3. Программный код проекта ConsoleOutputApplication

```
// Описание класса ConsoleOutputDemo:
class ConsoleOutputDemo{
  // Главный метод программы:
  public static void main(String[] args){
    // Вывод сообщения в консольное окно (окно вывода):
    System.out.println("Мы изучаем язык Java!");
  }
}
```

Поскольку в программе класс JOptionPane не используется, то необходимость в импортировании класса отпадает, и поэтому в программе import-инструкции нет.

В главном методе всего одна команда System.out.println("Мы изучаем язык Java!"), которой в окне вывода отображается следующее сообщение:

Результат выполнения программы (из листинга 1.3)

```
Мы изучаем язык Java!
```

Как это выглядит на практике, показано на рис. 1.4.

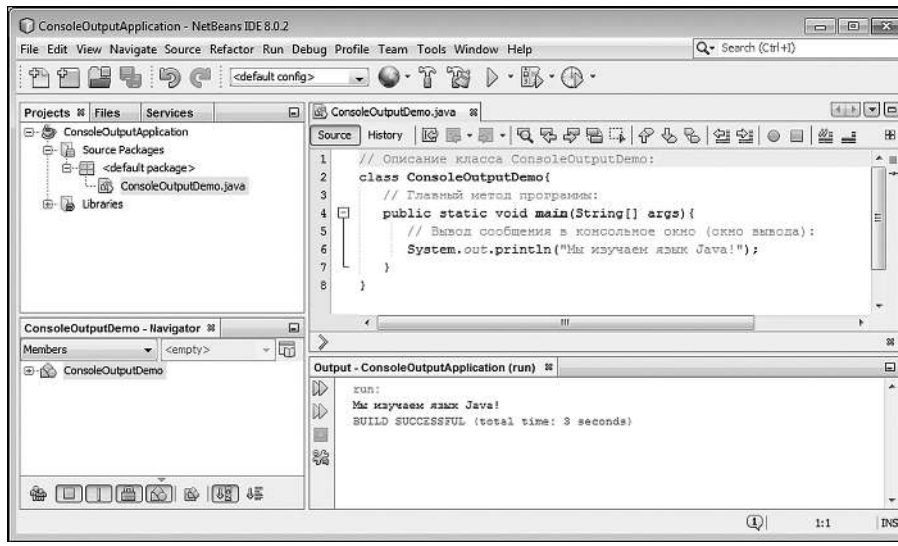


Рис. 1.4. Результат выполнения программы отображается во внутреннем окне вывода среды разработки NetBeans

Окно среды разработки NetBeans содержит внутреннее окно вывода **Output** (снизу под окном редактора с программным кодом), в котором отображается результат выполнения программы.

Окно с полем ввода

У меня вот тоже один такой был — крылья сделал.

Из к/ф «Иван Васильевич меняет профессию»

Информацию из программы приходится не только выводить — также важно иметь возможность передавать в программу различные данные.

Далее мы рассмотрим механизм передачи данных в программу с использованием диалогового окна, имеющего поле ввода.

Создание окна с полем ввода

Для отображения диалогового окна с текстовым полем ввода нам понадобится статический метод showInputDialog() из класса JOptionPane. Аргументом методу передается текстовое значение, отображаемое над полем ввода, а результатом метод возвращает текстовое значение, которое пользователь вводит в текстовое поле в окне. В листинге 1.5 представлен код программы, которой при запуске на выполнение сначала выводится запрос на ввод текстового значения, а затем введенный пользователем текст отображается в другом диалоговом окне.

Листинг 1.5. Программный код проекта InputDialogApplication

```
// Импорт класса JOptionPane:
import javax.swing.JOptionPane;
// Описание класса:
class InputDialogDemo{
  // Главный метод:
  public static void main(String[] args){
    // Переменная для записи текста:
    String text;
    // Отображения диалогового окна с полем вода:
    text=JOptionPane.showInputDialog("Введите текст:");
    // Отображение диалогового окна с сообщением:
    JOptionPane.showMessageDialog(null,"Вы ввели такой текст:\n"+text);
  }
}
```

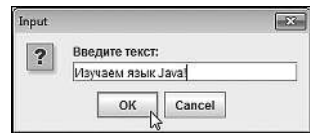


Рис. 1.5. Диалоговое окно с текстовым полем и введенным в поле текстом

При запуске программы на выполнение открывается диалоговое окно **Input** (название по умолчанию) с текстовым полем и надписью **Введите текст:** над ним. В окно вводим текст и щелкаем кнопку **ОК**. На рис. 1.5 показано окно с полем ввода и текстом "Изучаем язык Java!" в поле перед щелчком кнопки **ОК**.

После щелчка на кнопке **ОК**, окно с полем ввода закрывается, и вместо него появляется другое окно, представленное на рис. 1.6.

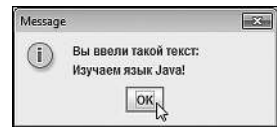


Рис. 1.6. Окно с сообщением содержит текст, который пользователь ввел в поле ввода в предыдущем окне

Это окно с сообщением, причем сообщение содержит и тот текст, который вводился в текстовое поле первого окна. Вкратце это результаты (или последствия) выполнения программы. Далее проанализируем программный код, благодаря которому все это происходит.

Анализ программного кода

Поскольку мы в программе используем методы из класса JOptionPane, то начинается программа инструкцией import javax.swing.JOptionPane, которой класс JOptionPane импортируется в программу. Все остальное в программном коде — описание класса InputDialogDemo. Класс, в свою очередь, состоит из главного метода программы. Там командой String text объявляется переменная text такая, что значением ей можно присваивать текст. Это и происходит при выполнении команды text=JOptionPane.showInputDialog("Введите текст:"). В правой части выражения инструкцией JOptionPane.showInputDialog("Введите текст:") из класса JOptionPane вызывается статический метод showInputDialog() текстовый аргумент "Введите текст:", переданный методу, определяет надпись над полем ввода в окне. Метод возвращает результат — текст, который пользователь вводит в текстовое поле. Поскольку результат вызова метода showInputDialog() присваивается значением переменной text, то данная переменная будет содержать текст, который пользователь ввел в текстовое поле (и подтвердил свой ввод щелчком на кнопке **ОК**). Переменная используется в следующей команде JOptionPane.showMessageDialog(null,"Вы ввели такой текст:\n"+text), которой отображается окно с сообщением. Текст сообщения «вычисляется» выражением "Вы ввели такой текст:\n"+text. Речь идет о «сумме» двух текстовых значений "Вы ввели такой текст:\n" и text. Если операция сложения применяется к текстовым строкам, то выполняется конкатенация (объединение) текстовых строк. Результатом выражения "Вы ввели такой текст:\n"+text является текстовая строка, получающаяся объединением текстового литерала "Вы ввели такой текст:\n" и значения текстовой переменной text.

Текстовый литерал содержит, кроме собственно текста, инструкцию \n. Инструкция \n используется для выполнения разрыва строки: в том месте, где размещена данная инструкция, выполняется переход к новой строке. Поэтому во втором окне с сообщением, отображаемом после окна с полем ввода, текст сообщения отображается в двух строках, причем переход к новой строке выполняется именно в том месте, где в текстовом литерале размещена инструкция \n (см. рис. 1.6).

ДЕТАЛИ

У диалогового окна с текстовым полем (см. рис. 1.5), помимо кнопки **OK**, имеется еще кнопка **Cancel**. Также в правом верхнем углу окна есть системная пиктограмма с крестиком. Если щелкнуть вместо кнопки **OK** кнопку **Cancel** или системную пиктограмму, окно будет закрыто. Но в этом случае, даже если поле ввода содержало текст, результатом метода showInputDialog() возвращается пустая ссылка null.

Управление видом окна с полем ввода

Для окна с полем ввода можно в явном виде задавать тип пиктограммы (по умолчанию используется пиктограмма со знаком вопроса) и название окна (отображается в строке названия, по умолчанию используется название **Input**). Существуют разные способы передачи аргументов методу showInputDialog(). Мы рассмотрим случай, когда при явном определении текста над полем ввода, названия окна и типа пиктограммы для окна аргументы методу showInputDialog() передаются в том же виде, как и методу showMessageDialog() в аналогичной ситуации. В листинге 1.6 представлена небольшая иллюстрация к вышесказанному.

Листинг 1.6. Программный код проекта ShowMeWindowsApplication

```
import javax.swing.JOptionPane;
class ShowMeWindowsDemo{
  public static void main(String[] args){
    // Название диалогового окна:
    String title;
    // Текст сообщения:
    String text;
    // Отображение первого окна с полем ввода:
```

```
    title=JOptionPane.showInputDialog(null,"Имя для окна:","Название",JOptionPane.
WARNING_MESSAGE);
    // Отображение второго окна с полем ввода:
    text=JOptionPane.showInputDialog(null,"Текст сообщения:","Содержание",JOptionPane.
PLAIN_MESSAGE);
    // Отображение окна с сообщением:
    JOptionPane.showMessageDialog(null,text,title,JOptionPane.INFORMATION_MESSAGE);
  }
}
```

Программа достаточно простая. В главном методе объявляются текстовые переменные title и text. Значения переменным присваиваются командами title=JOptionPane.showInputDialog(null,"Имя для окна:","Название",JOptionPane.WARNING_MESSAGE) и text=JOptionPane.showInputDialog(null,"Текст сообщения:","Содержание",JOptionPane.PLAIN_MESSAGE). В каждой из них метод showInputDialog() вызывается с четырьмя аргументами.

- Первым аргументом передается пустая ссылка null на родительское окно (которого, понятно, нет).
- Второй текстовый аргумент определяет текст, отображаемый над полем ввода.
- Третий текстовый аргумент задает название окна с полем ввода.
- Четвертый аргумент является статической константой класса JOptionPane и определяет пиктограмму, которая отображается в диалоговом окне. Речь о тех же константах, что используются при определении типа пиктограммы в окне с сообщением: ERROR_MESSAGE (пиктограмма «ошибка» — крест в кружке), INFORMATION_MESSAGE (информационная пиктограмма — буква i в кружке), WARNING_MESSAGE (пиктограмма «предупреждение» — восклицательный знак в треугольнике), QUESTION_MESSAGE (пиктограмма «вопрос» — знак вопроса в квадратной рамке, используется по умолчанию для окна с полем ввода) и PLAIN_MESSAGE (окно отображается без пиктограммы).

После того как значения переменных title и text определены, командой JOptionPane.showMessageDialog(null,text,title,JOptionPane.INFORMATION_MESSAGE) отображается окно с сообщением. Значение переменной title служит названием окна, а отображаемый в окне текст сообщения определяется значением переменной text.

На рис. 1.7 показано первое (из двух) окно с полем ввода, которое отображается при запуске программы.

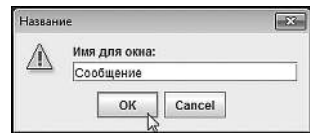


Рис. 1.7. Первое окно с полем ввода для определения названия окна с сообщением

После ввода текста в поле и подтверждения щелчком на кнопке **ОК**, отображается еще одно окно с полем ввода, представленное на рис. 1.8.

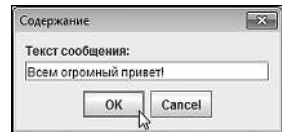


Рис. 1.8. Второе окно с полем ввода для определения текста сообщения

После щелчка на кнопке **ОК** в этом окне оно закрывается, а на экране появляется окно с сообщением, как показано на рис. 1.9.

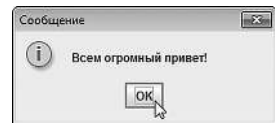


Рис. 1.9. Окно с сообщением. Название окна и содержание сообщения определяется на предыдущих этапах

Название окна определяется текстом, введенным в поле первого окна (см. рис. 1.7), а содержание сообщения — это текст, введенный в поле второго окна (см. рис. 1.8).

Консольный ввод

Какой паразит осмелился сломать двери в царское помещение?

Из к/ф «Иван Васильевич меняет профессию»

Выше мы познакомились со способом передачи в программу текстовых значений. В заключение главы кратко рассмотрим вопрос о том, как

данные (текстовые пока что) вводятся через консольное окно. Сразу отметим, что процедура эта не самая тривиальная, да и «теоретическая основа» под нее не подведена. Но если относиться к тому, что описывается далее, как к некоему алгоритму действий, которые следует выполнить при считывании информации с консоли, то ситуация представляется более-менее сносной.

> **НА ЗАМЕТКУ**
>
> Даже если что-то останется непонятным — впадать в отчаяние не стоит. К теме ввода и вывода данных мы еще будем возвращаться.

Система консольного ввода реализуется с использованием класса Scanner. Класс должен быть импортирован в программу, для чего используется инструкция import java.util.Scanner. На основе класса Scanner необходимо создать *объект*. Созданный объект, в свою очередь, должен быть связан с объектом стандартного потока ввода System.in. Вся эта жуткая и совершенно непонятная схема на практике реализуется одной-единственной командой следующего вида:

```
Scanner объект=new Scanner(System.in);
```

Все, что нужно сделать, — указать название для объекта (название выбираем сами, по своему усмотрению). После того как объект с указанными характеристиками создан, из него можно вызывать методы, среди которых имеется и метод nextLine(), считывающий текстовую строку, которую пользователь вводит в консольном окне. Строка возвращается результатом метода. Вот, собственно, и вся схема (алгоритм) реализации процедуры консольного ввода. Теперь посмотрим, как все описанное реализуется на практике. Поможет нам в этом программа из листинга 1.7.

Листинг 1.7. Программный код проекта ConsoleInputApplication

```
// Импортируется класс Scanner:
import java.util.Scanner;
// Описание класса:
class ConsoleInputDemo{
  // Главный метод:
```

```
  public static void main(String[] args){
    // Создание объекта input класса Scanner:
    Scanner input=new Scanner(System.in);
    // Переменные для считывания названия
    // дня недели и месяца:
    String day,month;
    // Отображается сообщение:
    System.out.print("Какой сегодня день? ");
    // Считывается текстовая строка:
    day=input.nextLine();
    // Отображается сообщение:
    System.out.print("Какой месяц? ");
    // Считывается текстовая строка:
    month=input.nextLine();
    // Текстовая переменная:
    String text;
    // Текстовое значение для отображения в консоли:
    text="Сегодня "+day+", месяц — "+month;
    // Отображается сообщение:
    System.out.println(text);
  }
}
```

Результат выполнения программы такой (жирным шрифтом выделены вводимые пользователем значения):

Результат выполнения программы (из листинга 1.7)

```
Какой сегодня день? понедельник, 21-е число
Какой месяц? сентябрь
Сегодня понедельник, 21-е число, месяц — сентябрь
```

При выполнении программы все самое интересное происходит в окне вывода среды разработки. Как все выглядит на начальном этапе, показано на рис. 1.10.

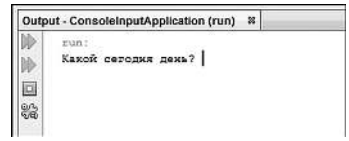


Рис. 1.10. Ввод первого текстового значения по запросу программы

После ввода первой текстовой фразы (день недели и число), нужно ввести вторую текстовую строку (название месяца), как показано на рис. 1.11.

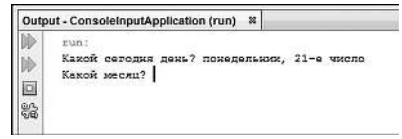


Рис. 1.11. Ввод второго текстового значения по запросу программы

Результат выполнения программы, как он выглядит в окне вывода, представлен на рис. 1.12.

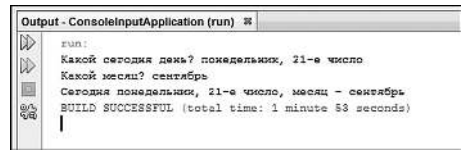


Рис. 1.12. Результат выполнения программы

Также следует учесть, что при работе с кириллическим текстом может понадобиться изменить используемую в приложении кодировку символов (в противном случае в окне вывода кириллический текст может отображаться некорректно). Для этого в окне **Projects** выделяем пункт с названием приложения и щелкаем правой кнопкой мыши. В раскрывшемся контекстном меню выбираем команду **Properties** (рис. 1.13).

Можно также воспользоваться командой **Project Properties** в меню **File**. Откроется окно свойств проекта, показанное на рис. 1.14.

В разделе **Categories** выбираем позицию **Source**, а в раскрывающемся списке **Encoding** выбираем кодировку символов (в данном случае используется стандартная кодировка **windows-1251**).

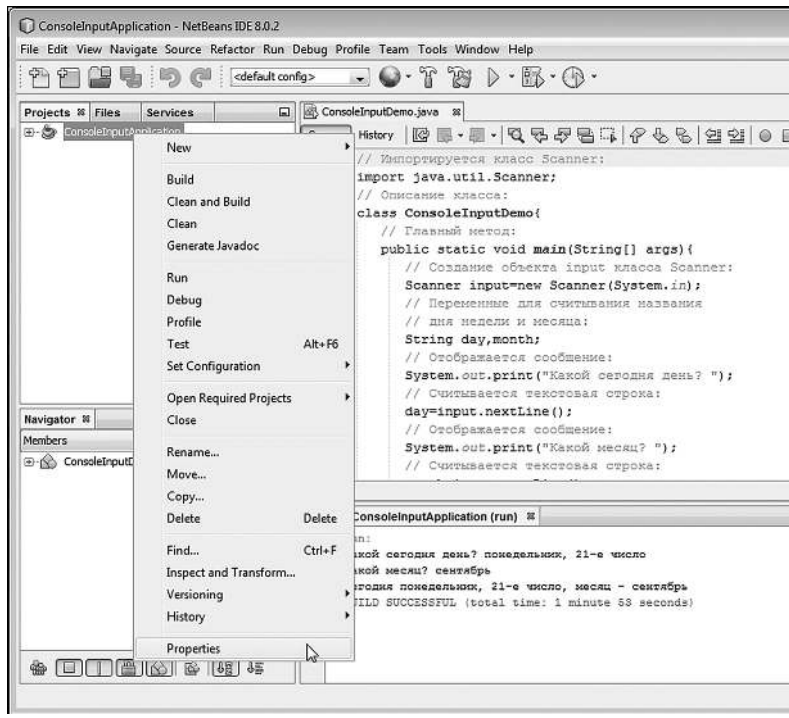


Рис. 1.13. В контекстном меню проекта выбирается команда **Properties**

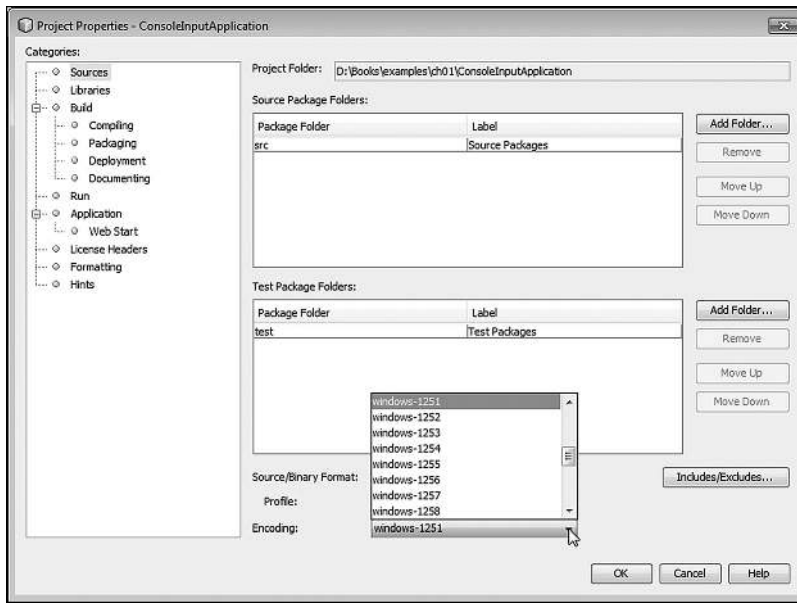


Рис. 1.14. В окне свойств проекта определяется тип кодировки

Теперь проанализируем программный код. Итак, программа начинается с инструкции import java.util.Scanner, которой в программу импортируется класс Scanner. В главном методе командой Scanner input=new Scanner(System.in) создается объект input класса Scanner. Также объявляются три текстовые переменные: day, month и text. Командой System.out.print("Какой сегодня день? ") в консольном окне отображается сообщение, а командой day=input.nextLine() считывается и записывается в переменную day текстовая строка, которую вводит пользователь (по окончании ввода строки следует нажать клавишу <Enter>).

НА ЗАМЕТКУ

Разница между методами println() и print() в том, что методом println() после отображения в консоли сообщения выполняется переход к новой строке. При отображении сообщения с помощью метода print() курсор остается в той же строке.

Командой System.out.print("Какой месяц? ") отображается еще одно сообщение, после чего командой month=input.nextLine() введенное пользователем значение записывается в переменную month. С помощью команды text="Сегодня "+day+", месяц — "+month значение переменной text вычисляется путем объединения нескольких текстовых строк. Результат отображается командой System.out.println(text).

Резюме

Ну, пошли дела кое-как.

Из к/ф «Иван Васильевич меняет профессию»

- Программа содержит, по крайней мере, описание хотя бы одного класса. Среди классов программы доложен быть один, содержащий описание главного метода программы main(). Выполнение программы отождествляется с выполнением метода main().
- Главный метод описывается с атрибутами public, static и void. Аргумент главного метода — текстовый массив. Тело метода (команды, выполняемые в программе) выделяется фигурными скобками.
- Описание класса начинается с ключевого слова class, после которого указывается имя класса. Описание класса выделяется блоком из фигурных скобок.

- Для вывода данных в консоль используют метод println(), который вызывается в формате System.out.println(). Аргументом методу передается значение (текст, например), которое отображается в консольном окне (или окне вывода среды разработки).
- Для отображения диалогового окна с сообщением используют статический метод showMessageDialog() из класса JOptionPane. Для использования класса JOptionPane в программу добавляется инструкция import javax.swing.JOptionPane, которой выполняется импортирование класса в программу.
- Для отображения диалогового окна с текстовым полем ввода используют статический метод showInputDialog() из класса JOptionPane.
- Для консольного ввода используют класс Scanner, который в программу импортируется инструкцией import java.util.Scanner. Командой вида Scanner объект=new Scanner(System.in) создается объект класса Scanner. Метод nextLine() этого объекта используется для считывания вводимых в консольном окне текстовых строк.

Глава 2
БАЗОВЫЕ ТИПЫ И ОСНОВНЫЕ ОПЕРАТОРЫ

Батюшка-царь, кто же это такой?
Из к/ф «Иван Васильевич меняет профессию»

В этой главе речь пойдет о переменных, базовых типах данных и основных операторах, которые есть в Java.

Переменные

Что ей надо, я тебе потом скажу.
Из к/ф «Бриллиантовая рука»

Переменная представляет собой идентификатор, связанный с областью памяти. Через этот идентификатор в память можно записывать значение, а также считывать значение из памяти. Далее мы обсудим переменные, объявляемые в методе `main()`. Здесь и далее, если явно не указано иное, под *переменной* подразумевается именно переменная, объявленная в главном методе программы.

> **НА ЗАМЕТКУ**
>
> Язык Java полностью объектно-ориентированный, поэтому с переменными сталкиваются обычно в описании методов. Такие переменные называются локальными и доступны только в теле метода. Далее речь идет о переменных, которые объявляются и используются в главном методе программы. Но принципиально такая ситуация мало отличается от описания переменных в прочих методах.
> Несколько позже мы узнаем, что в классах объявляются поля, которые, по сути, представляют собой переменные, но только связанные с объектом класса. Эти вопросы мы еще будем обсуждать.

Базовые типы

При объявлении переменной указывается тип переменной и ее имя. Тип переменной важен, поскольку от этого зависит область памяти,

выделяемая под переменную, а также способ интерпретации значения, считываемого из памяти. В Java, когда речь идет о типах данных, то обычно подразумеваются классы. Вместе с тем есть ограниченный набор так называемых *простых*, или *базовых* типов. С ними мы и познакомимся.

Каждый базовый тип данных определяется идентификатором — ключевым словом, обозначающим тип данных. Идентификаторы для базовых типов Java перечислены в табл. 2.1. Таблица также содержит информацию об объеме памяти, выделяемой для данных соответствующего типа и *классах-оболочках* (объяснение следует далее).

Табл. 2.1. Базовые типы Java

Тип	Описание	Размер памяти	Класс-оболочка
byte	Целочисленный тип. Переменная данного типа может принимать целочисленное значение в диапазоне от –128 до 127	8 бит	Byte
short	Целочисленный тип. Переменная данного типа может принимать целочисленное значение в диапазоне от –32768 до 32768	16 бит	Short
int	Целочисленный тип. Переменная данного типа может принимать целочисленное значение в диапазоне от –2147483647 до 2147483647	32 бита	Integer
long	Целочисленный тип. Переменная данного типа может принимать целочисленное значение в диапазоне от –9223372036854775807 до 9223372036854775807	64 бита	Long
float	Числовой тип. Переменная этого типа значением может принимать число в формате с плавающей точкой. Наибольшее (по модулю) значение равно $3{,}4\times10^{38}$, дискретность значений равна величине $3{,}4\times10^{-38}$	32 бита	Float
double	Числовой тип (двойной точности). Переменная этого типа значением может принимать число в формате с плавающей точкой. Наибольшее (по модулю) значение равно $1{,}7\times10^{308}$, дискретность значений равна величине $1{,}7\times10^{-308}$	64 бита	Double
char	Символьный тип. Переменная данного типа значением может принимать отдельный символ (букву)	16 бит	Character
boolean	Логический тип. Переменные этого типа могут принимать только два значения: true (истина) или false (ложь)	Зависит от реализации виртуальной машины	Boolean

Условно все типы можно разбить на четыре неравнозначные группы: целые числа, действительные числа, символы и логические значения. Для реализации целых чисел используется четыре типа (byte, short, int и long), отличающихся лишь объемом выделяемой памяти и, как следствие, допустимым диапазоном значений переменных данных типов.

НА ЗАМЕТКУ

Если для переменной с числовым значением выделяется n битов, то такая переменная может принимать 2^n различных значений: в один бит можно записать 0 или 1, а если битов n, то всего возможно 2^n различных комбинаций из нулей и единиц. Диапазон возможных значений целочисленной переменной, записанной с помощью n битов, лежит в диапазоне от -2^{n-1} до $2^{n-1} - 1$ (одно число уходит на кодирование нуля). Например, при $n = 8$ получаем всего $2^8 = 256$ разных чисел в диапазоне от $-2^7 = -128$ до $2^7 - 1 = 127$. Для $n = 32$ общее количество чисел, которые можно закодировать с помощью такого количества битов, равняется $2^{32} = 4294967296$, и диапазон возможных значений от $-2^{31} = -2147483648$ до $2^{31} - 1 = 2147483647$.

Стандартным типом при работе с целыми числами является тип int. Его по возможности и рекомендуется использовать. Главная причина в том, что целочисленные литералы (целые числа, которые используются в программном коде, например 5 или 123) по умолчанию интерпретируются как относящиеся к типу int. К тому же есть такая штука, как автоматическое приведение типов (обсуждается дальше), и оно может преподнести неожиданные и не всегда приятные сюрпризы. Так что если нет крайней необходимости прибегать к помощи иных целочисленных типов — используем тип int.

Числа с плавающей точкой (действительные числа) реализуются с помощью типов double и float. Для значений типа double памяти выделяется в два раза больше (по сравнению с данными типа float). Соответственно, для чисел типа double шире диапазон возможных значений и меньше величина «дискретности» чисел (фактически точность числа). Действительные числовые литералы (например, 12.3 или 5.0) по умолчанию интерпретируются как значения типа double. Вывод простой — если имеем дело с действительными числами, то лучше использовать переменные типа double.

Для работы с символами используют тип char. Значением переменной типа char может быть символ. Символьные литералы (например, 'A' или 'q') заключаются в одинарные кавычки.

НА ЗАМЕТКУ

Литерал заключается в одинарные кавычки. Именно одинарные кавычки являются признаком символа. Если символ (один) заключить в двойные кавычки, то это уже будет текст (объект класса String). Например, 'A' — символьный литерал, а "A" — текст, состоящий из одной буквы. И это абсолютно разные типы данных.

Переменная типа boolean (логический тип) может принимать всего два значения: true или false. Обычно такие переменные используются в управляющих инструкциях (условном операторе и операторах цикла) или операндами в логических выражениях. О переменной логического типа говорят, что она имеет *истинное* значение, если ее значение равно true, а если переменная имеет значение false, то говорят что это *ложное* значение.

Для каждого из базовых типов существует *класс-оболочка*. Класс-оболочка открывает альтернативный способ создания и работы с данными соответствующего типа. Преимущество классов-оболочек, кроме прочего, связано с тем, что такие классы содержат ряд статических методов, позволяющих выполнять различные полезные операции, наподобие преобразования текстового представления для числа в число (например, когда текст "123" необходимо преобразовать в целое число 123). Именно в таком аспекте нами будут использоваться классы-оболочки.

Объявление и инициализация переменных

Рассмотрим теперь вопрос собственно об объявлении переменных. Процедура эта очень простая. Как отмечалось ранее, при объявлении переменной указывается тип переменной (идентификатор типа) и имя переменной. Есть несколько простых правил.

- Переменная объявляется в любом месте — главное, чтобы до первого ее использования (использование — присваивание значения или считывание значения).
- Если объявляется несколько переменных одного типа, то для них можно использовать один идентификатор типа, а названия переменных указываются через запятую.
- При объявлении переменной ей сразу можно присвоить значение (такая процедура называется инициализацией переменной).

Ниже приведены примеры корректного объявления и инициализации переменных разных типов:

```
int num=10,a,b=5;
double x,y=12.5;
float z;
char s='B',symb;
```

В данном случае объявляются три целочисленные переменные (num, a и b) типа int, причем переменная num получает значение 10. Переменные x и y объявлены как относящиеся к типу double (переменной y присвоено значение 12.5). Переменная z относится к типу float. Символьные переменные s и symb объявлены одним выражением, при этом переменной s присвоено значение 'B'.

Ситуация с объявлением переменных может быть более замысловатой. Рассмотрим представленный ниже блок кода:

```
int one=1,two=2;
int three=one+two;
```

Командой int one=1,two=2 объявляются и инициализируются две целочисленные переменные one и two типа int. Затем командой int three=one+two объявляется переменная three типа int, и значением переменной присваивается выражение one+two, представляющее собой сумму двух других переменных. Такая ситуация допустима: переменная может инициализироваться со значением, определяемым на основе выражения, содержащего другие переменные.

Главное условие — переменные, входящие в инициализирующее выражение, должны иметь значения на момент вычисления выражения. Данный способ инициализации переменных называется *динамическим*.

НА ЗАМЕТКУ

Важно понимать, что при динамической инициализации переменной **не устанавливается** функциональной связи между переменными. Например, если переменная three инициализируется командой three=one+two, то она получит значением результат выражения one+two (на момент его вычисления). Если впоследствии изменятся значения переменных one и/или two, на значении переменной three это никак не скажется.

Еще один важный аспект — доступность переменной. Главное правило состоит в том, что переменная доступна в пределах блока, в котором она объявлена. Блок, в свою очередь, определяется парой фигурных скобок (открывающей { и закрывающей }). Поскольку мы обсуждаем объявление переменных внутри главного метода программы, то такие переменные доступны в главном методе (но не вне его).

Как небольшую иллюстрацию рассмотрим программу в листинге 2.1, в которой объявляется несколько переменных разных типов, переменным присваиваются значения, после чего значения этих переменных отображаются в окне сообщения.

Листинг 2.1. Программный код проекта UsingVariablesApplication

```
import javax.swing.JOptionPane;
class UsingVariablesDemo{
  public static void main(String[] args){
    // Целочисленная переменная:
    int number=123;
    // Действительная переменная:
    double x=32.1;
    // Символьная переменная:
    char symb='A';
    // Логическая переменная:
    boolean state=true;
    // Текстовая переменная для формирования
    // содержания сообщения:
    String text="Используемые переменные:\n";
    // Дописывается значение целочисленной переменной:
    text=text+"Целое число: "+number+"\n";
    // Дописывается значение числовой переменной:
    text=text+"Действительное число: "+x+"\n";
    // Дописывается значение символьной переменной:
    text=text+"Символ: "+symb+"\n";
    // Дописывается значение логической переменной:
    text=text+"Логическое значение: "+state;
```

```
    // Отображение сообщения:
    JOptionPane.showMessageDialog(null,text);
  }
}
```

При выполнении программы появляется диалоговое окно, представленное на рис. 2.1.

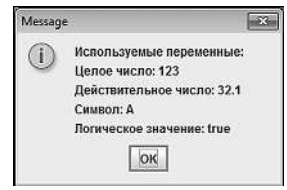


Рис. 2.1. В результате выполнения программы появляется окно с информацией о значении переменных

В данном случае в главном методе программы объявляется несколько переменных (переменная number типа int со значением 123, переменная x типа double со значением 32.1, переменная symb типа char со значением 'A' и переменная state типа boolean со значением true). Также в программе объявляется текстовая переменная text, в которую последовательно дописывается текст и значение каждой из перечисленных выше переменных. После того как значение переменной text сформировано, оно отображается в окне сообщения.

ДЕТАЛИ

Инструкция \n, которая несколько раз используется в текстовом значении переменной text, является инструкцией перехода к новой строке: там, где в тексте расположена данная инструкция, при отображении текста выполняется переход к новой строке. Также имеет смысл напомнить, что операция сложения для текстовых значений реализуется путем объединения текстовых строк. Например, команда text=text+"Целое число: "+number+"\n" выполняется так: формируется новая текстовая строка, которая получается объединением текущего значения переменной text, текста "Целое число: ", значения переменной number и текста "\n". Полученная строка присваивается значением переменной text.

Для сравнения в листинге 2.2 представлена аналогичная программа, но только результат (значения переменных) отображается не в окне сообщения, а в консольном окне.

Листинг 2.2. Программный код проекта UsingVarsConsoleApplication

```
class UsingVarsConsoleDemo{
  public static void main(String[] args){
    // Целочисленная переменная:
    int number=123;
    // Действительная переменная:
    double x=32.1;
    // Символьная переменная:
    char symb='A';
    // Логическая переменная:
    boolean state=true;
    // Отображение текста:
    System.out.println("Используемые переменные:");
    // Отображение значения целочисленной переменной:
    System.out.println("Целое число: "+number);
    // Отображение значения числовой переменной:
    System.out.println("Действительное число: "+x);
    // Отображение значения символьной переменной:
    System.out.println("Символ: "+symb);
    // Отображение значения логической переменной:
    System.out.println("Логическое значение: "+state);
  }
}
```

Результат выполнения программы такой, как показано ниже.

Результат выполнения программы (из листинга 2.2)

```
Используемые переменные:
Целое число: 123
Действительное число: 32.1
Символ: A
Логическое значение: true
```

Для вывода значения переменных в консольное окно использован метод println(), который вызывается в формате System.out.println(). Поскольку после окончания вывода данных в консоль методом println() переход к новой строке выполняется автоматически, инструкцию \n в тексте мы не используем. Также в этот раз мы обошлись без текстовой переменной, а процесс вывода данных разбили на несколько команд.

Считывание значения переменной

Часто в программах приходится не просто присваивать значения переменным, а вводить и считывать эти значения — через окно с полем ввода или через консольное окно.

> **НА ЗАМЕТКУ**
>
> Мы уже знакомы с ситуацией, когда значение переменной считывается с поля ввода. Но там, напомним, речь шла о текстовой переменной. Здесь мы собираемся рассмотреть способы считывания значений переменных разных типов.

Далее в программе из листинга 2.3 показано, как значения целочисленного типа могут вводиться в программу через диалоговое окно с полем ввода.

Листинг 2.3. Программный код проекта InputIntVariablesApplication

```
import javax.swing.JOptionPane;
class InputIntVariablesDemo{
  public static void main(String[] args){
    // Целочисленные переменные:
    int age,year,birth;
    // Переменная для записи значения в поле ввода:
    String result;
    // Считывание значения из поля ввода:
    result=JOptionPane.showInputDialog("Какой сейчас год?");
    // Преобразование текста в целое число:
    year=Integer.parseInt(result);
    // Считывание значения из поля ввода:
```

```
    result=JOptionPane.showInputDialog("Сколько Вам лет?");
    // Преобразование текста в целое число:
    age=Integer.parseInt(result);
    // Вычисление года рождения:
    birth=year-age;
    // Отображение окна с сообщением:
    JOptionPane.showMessageDialog(null,"Вы родились в "+birth+" году!");
  }
}
```

При выполнении программы сначала отображается окно с полем ввода, в котором следует указать текущий год, как показано на рис. 2.2.

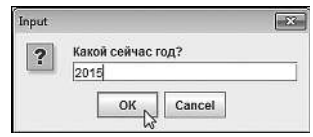


Рис. 2.2. В поле ввода указывается текущий год

После этого появляется еще одно окно с полем ввода, в котором указывается возраст. Ситуация проиллюстрирована на рис. 2.3.

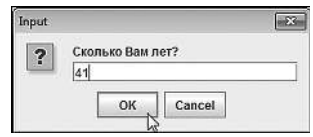


Рис. 2.3. В поле ввода указывается возраст

После этого появится окно с сообщением, в котором указан год рождения, как, например, показано на рис. 2.4.

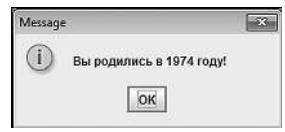


Рис. 2.4. Окно результатами вычисления года рождения

В программе объявляется несколько целочисленных переменных (тип int): age, year и birth. В переменную age мы планируем записывать возраст пользователя, в переменную year будет записываться текущий год, а значение переменной birth командой birth=year-age вычисляется как разность текущего года и возраста пользователя, что в результате дает год рождения.

Значения переменных age и year считываются из поля ввода диалогового окна. Проблема в том, что значения следует вводить целочисленные, в то время как методом showInputDialog() возвращается текстовое значение. Поэтому даже если мы в поле ввода вводим число 2017, методом showInputDialog() возвращается текст "2017". Это так называемое текстовое представление числа. Его следует трансформировать в число. Поможет нам статический метод parseInt() из класса-оболочки Integer (метод вызывается в формате Integer.parseInt()). Аргументом методу передается текстовое представление числа, а результатом возвращается число. Отсюда алгоритм считывания целочисленных значений через поле ввода такой: результат вызова метода showInputDialog() записывается в текстовую переменную result, после чего переменная result передается аргументом методу parseInt() (команды year=Integer.parseInt(result) и age=Integer.parseInt(result)).

Аналогично поступаем, если нужно, например, считать из поля ввода значение типа double. Теперь только следует использовать статический метод parseDouble() из класса-оболочки Double. Пример приведен в листинге 2.4.

Листинг 2.4. Программный код проекта InputDoubleVariablesApplication

```
import javax.swing.JOptionPane;
class InputDoubleVariablesDemo{
  public static void main(String[] args){
    // Числовые переменные:
    double mass,height,bmi;
    // Переменная для записи значения в поле ввода:
    String result;
    // Считывание значения из поля ввода:
    result=JOptionPane.showInputDialog("Ваш рост в метрах:");
    // Преобразование текста в число:
    height=Double.parseDouble(result);
```

```
    // Считывание значения из поля ввода:
    result=JOptionPane.showInputDialog("Ваш вес в килограммах:");
    // Преобразование текста в число:
    mass=Double.parseDouble(result);
    // Вычисление индекса массы тела:
    bmi=mass/height/height;
    // Округление полученного значения:
    bmi=Math.round(bmi*100)/100.0;
    // Отображение окна с сообщением:
    JOptionPane.showMessageDialog(null,"Индекс массы тела: "+bmi);
  }
}
```

В программе на основе значений для роста и массы тела вычисляется индекс массы тела.

НА ЗАМЕТКУ

Индекс массы тела вычисляется как отношение массы m (в килограммах) тела к квадрату роста h (в метрах), то есть как m/h^2. Другими словами, чтобы вычислить индекс массы тела, необходимо дважды поделить массу тела на рост. Для взрослых людей нормой считается значение индекса массы тела от 18,5 до 25.

Первое диалоговое окно, которое появляется на экране, предназначено для ввода значения роста (рис. 2.5).

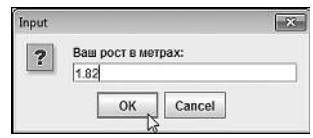


Рис. 2.5. В поле ввода указывается рост (в метрах)

Рост указывается в метрах, поэтому в принципе это нецелое число. Значение, указанное в поле ввода, запоминается в виде текста в переменную result, а затем командой height=Double.parseDouble(result) преобразуется в значение типа double и присваивается значением переменной height. Аналогично

поступаем при определении массы тела (записывается в переменную mass). На рис. 2.6 показан процесс ввода значения для массы тела.

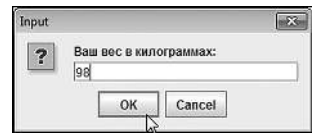


Рис. 2.6. В поле ввода указывается масса (в килограммах)

На рис. 2.7 показано окно с вычисленным значением индекса массы тела.

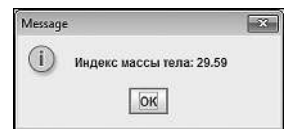


Рис. 2.7. Результат вычисления индекса массы тела

Индекс массы тела вычисляется командой bmi=mass/height/height. В общем случае такое значение будет содержать много цифр в десятичной части. Нам такая точность не нужна, поэтому мы выполняем округление до сотых. Для этого использована команда bmi=Math.round(bmi*100)/100.0. Выражение Math.round(bmi*100)/100.0 вычисляется так: текущее значение bmi умножается на 100, полученное значение округляется (с помощью статического метода round() класса Math) до целого числа, после чего оно делится на 100.0. Это и есть новое значение переменной bmi.

ДЕТАЛИ

В классе Math есть много статических методов, через которые реализуются всевозможные математические функции. В частности, статический метод round() используется для округления до целочисленных значений чисел в формате с плавающей точкой. Метод вызывается в формате Math.round(), а аргументом ему передается округляемое число (при этом само число остается неизменным, просто на его основе методом вычисляется и возвращается «округленный» результат). Выше мы использовали такой подход: умножили значение переменной на 100, округлили до целого значения, а затем поделили на 100. В результате получилось округление до сотых. Здесь есть один важный момент: оператор деления / в Java реализован так, что если делятся два целых числа, то деление выполняется нацело (результатом

является целое число, а остаток от деления отбрасывается). Поэтому после округления мы делили на число, записанное как 100.0, то есть использовали десятичную точку. Записанный в таком виде числовой литерал интерпретируется не как целое, а как действительное число и деление выполняется обычное, а не целочисленное.

Еще один небольшой пример консольного ввода числовых значений (целых и в формате с плавающей точкой) представлен в программе в листинге 2.5.

Листинг 2.5. Программный код проекта InputVariablesConsoleApplication

```
// Импортирование класса Scanner:
import java.util.Scanner;
class InputVariablesConsoleDemo{
  public static void main(String[] args){
    // Создание объекта input класса Scanner:
    Scanner input=new Scanner(System.in);
    // Текущий год:
    int now=2015;
    // Переменная для записи имени пользователя:
    String name;
    // Переменная для записи возраста:
    int age;
    // Переменная для записи значения роста (в метрах):
    double height;
    // Переменная для записи значения массы
    // (в килограммах):
    double mass;
    System.out.print("Ваше имя: ");
    // Считывание имени (текст):
    name=input.nextLine();
    System.out.print("Ваш возраст: ");
    // Считывание возраста (целое число):
    age=input.nextInt();
```

```
    System.out.print("Ваш рост (в метрах): ");
    // Считывание роста в метрах
    // (число в формате с плавающей точкой):
    height=input.nextDouble();
    System.out.print("Масса тела (в килограммах): ");
    // Считывание массы в килограммах
    // (число в формате с плавающей точкой):
    mass=input.nextDouble();
    // Вычисление индекса массы тела:
    double bmi=mass/height/height;
    // Отображение имени:
    System.out.println("Здравствуйте, "+name+"!");
    // Вычисление и отображение года рождения:
    System.out.println("Вы родились в "+(now-age)+" году.");
    // Отображение значения индекса массы тела:
    System.out.printf("Ваш индекс массы тела: %5.2f\n",bmi);
  }
}
```

Результат выполнения программы представлен ниже (жирным шрифтом выделены значения, вводимые пользователем):

Результат выполнения программы (из листинга 2.5)

Ваше имя: **Иван Петров**

Ваш возраст: **41**

Ваш рост (в метрах): **1,82**

Масса тела (в килограммах): **98**

Здравствуйте, Иван Петров!

Вы родились в 1974 году.

Ваш индекс массы тела: 29,59

В программе считываются следующие параметры: имя пользователя (переменная `name`), возраст (переменная `age`), рост (переменная `height`) и масса тела (переменная `mass`). В программе отображается приветствие

(с указанием имени пользователя), вычисляется и отображается год рождения, а также индекс массы тела. Для считывания текстового значения используем метод nextLine(), считывание значения типа int выполняется с помощью метода nextInt(), а для считывания значения типа double используем метод nextDouble().

НА ЗАМЕТКУ

В соответствии с локальными настройками среды разработки NetBeans при вводе десятичных чисел (чисел в формате числа с плавающей точкой) может оказаться, что вместо десятичной точки следует использовать десятичную запятую. Такая ситуация имеет место выше.

Год рождения вычисляется как разница значений переменной now (значением переменной указан текущий год) и переменой age (возраст пользователя, вводится с клавиатуры).

ДЕТАЛИ

В команде System.out.println("Вы родились в "+(now-age)+" году.") выражение, которым вычисляется год рождения, взято в круглы скобки. Если этого не сделать, разность чисел вычисляться не будет. Чтобы понять причину, рассмотрим выражение "2+2="+2+2. Результатом такого выражения является текстовая строка "2+2=22". Объяснение такое: при вычислении выражения "2+2="+2+2 к текстовой строке сначала дописывается число 2, а затем к получившемуся тексту "2+2=2" дописывается еще одно число 2. В итоге получается текст "2+2=22". Результатом выражения "2+2="+(2+2) является текстовая строка "2+2=4", поскольку в данном случае сначала вычисляется сумма 2+2 (получается значение 4), а результат дописывается к текстовой строке "2+2=", в результате чего получаем текстовую строку "2+2=4".

Для отображения значения переменной bmi (индекс массы тела) использован метод printf(). Метод вызывается в формате System.out.printf(), и ему в данном случае передается два аргумента: строка форматирования "Ваш индекс массы тела: %5.2f\n" и значение переменной bmi. Команда System.out.printf("Ваш индекс массы тела: %5.2f\n",bmi) выполняется так: отображается текстовая строка "Ваш индекс массы тела: %5.2f\n", но вместо выражения %5.2f (инструкция форматирования) отображается значение переменной bmi. В инструкции %5.2f символ % является идентификатором инструкции форматирования, символ f означает, что отображается числовое значение

в формате с плавающей точкой, а числовое выражение 5.2 означает, что для отображения числа используется не менее пяти позиций, причем после десятичной точки (запятой) отображается две цифры. Поскольку при вызове метода printf() автоматический переход к новой строке не выполняется, то в конце текстового литерала использована инструкция \n.

НА ЗАМЕТКУ

Благодаря методу printf() нам удалось избежать необходимости округлять вычисленное значение для индекса массы тела. Вместо округления мы явно задали формат вывода числовых данных (две цифры после десятичной точки), благо метод printf() позволяет это сделать.

Литералы и управляющие символы

Кроме собственно переменных, в программах используются *литералы*. Литералы — это константные значения, которые нельзя изменить в программе и которые представляют собой некоторые выражения или конструкции, понятные для программиста. Примерами литералов являются: 123 (целое число), 123.45 (число в формате с плавающей точкой), 'A' (символ), "текст" (текстовое значение), 1.2e-3 (число $1{,}2\times10^{-3}$, записанное в научной нотации), 0123 (восьмеричное представление числа 83), 0xA1B (число 2587 в шестнадцатеричном представлении).

ДЕТАЛИ

Для позиционного представления числа необходимо хотя бы две цифры. Количество используемых цифр определяет систему счисления. Если для представления чисел используется h различных цифр, то в позиционном представлении числа $\overline{b_m b_{m-1} \ldots b_2 b_1 b_0}$ параметры $b_k = 0, 1, 2, \ldots, h-1$ для всех индексов $k = 0, 1, 2, \ldots, m$, а значение такого числа в десятичной системе вычисляется как $\overline{b_m b_{m-1} \ldots b_2 b_1 b_0} = b_0 \cdot h^0 + b_1 \cdot h^1 + b_2 \cdot h^2 + \ldots + b_m \cdot h^m$. Например, для двоичной системы $h = 2$ и $\overline{b_m b_{m-1} \ldots b_2 b_1 b_0} = b_0 \cdot 2^0 + b_1 \cdot 2^1 + b_2 \cdot 2^2 + \ldots + b_m \cdot 2^m$, где параметры $b_k = 0,1$. Для восьмеричной системы $h = 8$ и $\overline{b_m b_{m-1} \ldots b_2 b_1 b_0} = b_0 \cdot 8^0 + b_1 \cdot 8^1 + b_2 \cdot 8^2 + \ldots + b_m \cdot 8^m$, причем параметры $b_k = 0, 1, 2, \ldots, 7$. В частности, если имеем дело с числом 123 в восьмеричной системе, то в десятичной получаем $1 \cdot 8^2 + 2 \cdot 8^1 + 3 \cdot 8^0 = 64 + 16 + 3 = 83$.

В шестнадцатеричной системе ($h = 16$) кроме десяти цифр от 0 до 9 еще используются буквы от A (обозначает число 10) до F (обозначает число 15). Перевод в десятичную систему выполняется в соответствии с соотношением $\overline{b_m b_{m-1} \ldots b_2 b_1 b_0} = b_0 \cdot 16^0 + b_1 \cdot 16^1 + b_2 \cdot 16^2$

+ ... + $b_m \cdot 16^m$, а параметры b_k = 0, 1, 2, ..., 15. Например, шестнадцатеричному числу *A*1*B* в десятичной системе соответствует значение $10 \cdot 16^2 + 1 \cdot 16^1 + 11 \cdot 16^0 = 2560 + 16 + 11 = 2587$ (напомним, что в шестнадцатеричном представлении символ *A* соответствует числу 10, а символ *B* соответствует числу 11).

При работе с литералами следует иметь в виду несколько простых правил.

- По умолчанию целочисленные литералы (например, `123`) интерпретируются как значения типа `int`.
- Числа в формате с плавающей точкой (например, `12.345`) по умолчанию относятся к типу `double`.
- Восьмеричное целое число начинается с нуля — например, `123` является десятичным числом, а `0123` есть число восьмеричное.
- Шестнадцатеричные литералы начинаются с комбинации символов `0x` или `0X` — например, число `0x123` является шестнадцатеричным.

С числовыми значениями можно использовать суффиксы `L` (или `l`) — для создания целочисленного литерала, относящегося к типу `long` (например, `123L`), и `F` (или `f`) — для создания действительного литерала относящегося к типу `float` (например, `12.345F`). Также действительные числа можно записывать в научной (или экспоненциальной) нотации: указывается мантисса числа и через символ `E` (или `e`) — показатель степени. Например, литерал `1.23E-5` соответствует числу $1{,}23 \cdot 10^{-5}$, а литерал `5.3E7` соответствует числу $5{,}3 \cdot 10^7$.

Нередко на практике используются специальные символы (или инструкции), которые называются *управляющими*. Начинаются такие инструкции с косой черты \, формально состоят из двух символов (включая косую черту \), но интерпретируются как один символ. С одной управляющей инструкцией мы знакомы — это инструкция `\n` перехода к новой строке. Помимо нее, можно выделить еще инструкцию `\t` для выполнения табуляции (вставки символа табуляции в текст).

Приведение типов

Есть очень важный и во многих случаях незаметный механизм, который называется *приведением типов*. На сцену он выходит в тех случаях, когда в некотором выражении присутствуют значения разных типов. Обычно

процесс приведения типов разделяют на *явное* и *неявное* (или *автоматические*). Неявное, или автоматическое, приведение типов выполняется автоматически и подчиняется определенной логике. Например.

- Если в числовом выражении имеется операнд типа double, то все прочие операнды (и результат выражения) автоматически расширяются до типа double.
- Если в числовом выражении имеется операнд типа float, и при этом нет операндов типа double, то расширение выполняется до типа float.
- Если в числовом выражении есть только целочисленные операнды и среди них имеется операнд типа long, то прочие операнды и результат выражения расширяется до типа long.
- Если в числовом выражении есть только целочисленные операнды, и среди них нет операндов типа long, то прочие операнды (в том числе и типов short и byte) и результат выражения расширяется до типа int.

При явном приведении типа перед выражением, тип которого необходимо «скорректировать», в круглых скобках указывается тип, к которому выполняется приведение. Ниже приведен небольшой пример программного кода, в котором объявляется и инициализируется переменная x типа byte, после чего ее начальное значение увеличивается в два раза:

```
byte x=1;
x=(byte)(x*2);
```

Если в команде x=(byte)(x*2) не использовать инструкцию явного приведения типа (byte), возникнет ошибка (программа с таким кодом не скомпилируется). Причина в том, что целочисленный литерал 2 интерпретируется как значение int и поэтому результатом выражения x*2 является значение типа int, которое мы пытаемся присвоить переменной типа byte. Поскольку для типа byte выделяется всего 8 бит, а для типа int выделяется 32 бита, может произойти потеря точности. Чтобы показать, что мы осознанно идем на такую процедуру, используем явное приведение значения типа int к значению типа byte.

НА ЗАМЕТКУ

Преобразование значения типа int к значению типа byte выполняется так: из 32 битов остаются только младшие 8, а остальные отбрасываются.

В приведенном далее блоке кода также необходимо выполнять приведение типа:

```
byte x=1,n=2;
x=(byte)(x*n);
```

Здесь мы не используем литерал 2, а вместо него использована переменная n типа byte со значением 2. Но при вычислении выражения x*n результат все равно расширяется до типа int, поэтому приходится явно выполнять привидение к типу byte.

Рассмотрим еще один пример корректных команд:

```
byte x=1,y=10;
x=(byte)(x+1);
y++;
```

Здесь объявляются две переменные x и y (с начальными значениями 1 и 10 соответственно) типа byte. Далее выполняются команды x=(byte)(x+1) и y++. В каждой из команд значение переменной увеличивается на единицу, но в первой из команд это делается через присваивание, а во второй команде использован оператор инкремента. Команда x=(byte)(x+1) содержит инструкцию (byte) явного приведения типа (причины те же, что и ранее — выражение в правой части от оператора присваивания автоматически расширяется до типа int). В команде y++ никаких инструкций приведения типа нет (да там их и негде размещать). Причина в том, что операция вида A++ на самом деле эквивалентна не команде A=A+1, а команде A=(тип A)(A+1) то есть к операнду прибавляется единица, после чего результат приводится к типу операнда. В этом смысле выражения вида A++ и A-- на основе операторов инкремента и декремента более предпочтительны к использованию по сравнению с операциями вида A=A+1 и A=A-1 соответственно.

НА ЗАМЕТКУ

В рассмотренных выше командах переменным типа byte при инициализации присваиваются числовые значения (например, команда byte x=1). Литерал 1, напомним, интерпретируется как значение типа int. Тем не менее в подобных случаях, когда переменной типа byte или short присваивается значением целочисленный литерал, явное приведение типов выполнять не обязательно — литерал автоматически «ужимается» (путем отбрасывания лишних старших битов) до нужного типа.

> Вообще, здесь еще раз хочется повторить рекомендацию: во избежание недоразумений целочисленные переменные разумно объявлять с типом int, а для работы с числами в формате с плавающей точкой использовать тип double.

Приведение типов не ограничивается только целочисленными типами. Так, можно привести к целочисленному значению число в формате с плавающей точкой (обратное преобразование тоже выполняется, но обычно это происходит автоматически). В таком случае у действительного числа просто отбрасывается дробная часть. Например, результатом выражения (int)5.8 является целое число 5: приведение числа 5.8 к типу int производится путем отбрасывания дробной части, остается только целая часть числа, причем интерпретируется она именно как целое число. А результатом выражения (double)5 является действительное значение 5.0 типа double.

Еще один характерный пример приведения типов — когда вычисляется выражение, представляющее собой сумму текста и числа. В этом случае число автоматически трансформируется в текстовое представление и выполняется конкатенация (объединение) строк. Мы с подобными ситуациями уже сталкивались. Вообще, тему автоматического и явного приведения типов можно обсуждать долго, но лучше делать это на конкретных примерах. Поэтому мы будем возвращаться к данному вопросу, но уже при рассмотрении прикладных задач.

Основные операторы

Ничего особенного. Обыкновенная контрабанда.

Из к/ф «Бриллиантовая рука»

Операторы, используемые в Java, условно можно разделить на четыре группы: арифметические, логические, побитовые и операторы сравнения.

Арифметические операторы

Арифметические операторы применяются в основном для выполнения арифметических операций. В табл. 2.2 перечислены все основные арифметические операторы, используемые в Java.

Табл. 2.2. Арифметические операторы Java

Оператор	Описание
+	Бинарный оператор *сложения*. Результатом выражения вида A+B является сумма значений числовых операндов A и B. Если один из операндов является текстовым значением, то второй операнд автоматически преобразуется (если необходимо) к текстовому типу и затем результат вычисляется путем конкатенации (объединение) текстовых строк
-	Бинарный оператор *вычитания*. Результатом выражения вида A-B является разность значений числовых операндов A и B
*	Оператор *умножения*. Результат выражения вида A*B вычисляется как произведение значений числовых операндов A и B
/	Бинарный оператор *деления*. Результатом выражения вида A/B является частное от деления значений числовых операндов A и B. Особенность оператора деления в том, что если операндами являются целые числа, то деление выполняется нацело. Например, результатом выражения 15/4 является целое число 3. Чтобы деление выполнялось обычное (не целочисленное), следует использовать инструкцию (double). Так, результатом выражения (double)15/4 является значение 3.75
%	Бинарный оператор *вычисления остатка* от целочисленного деления. Результатом выражения A%B является остаток от целочисленного деления значения операнда A на значение операнда B
++	Унарный оператор *инкремента*. У оператора есть префиксная и постфиксная форма. В результате вычисления выражения вида A++ (постфиксная форма) или ++A (префиксная форма) значение операнда A увеличивается на 1. Результатом выражения A++ возвращается исходное (до увеличения на единицу) значение операнда A, а результатом выражения ++A возвращается новое (после увеличения на единицу) значение операнда A
--	Унарный оператор *декремента*. Унарный оператор. У оператора есть префиксная и постфиксная форма. В результате вычисления выражения вида A-- (постфиксная форма) или --A (префиксная форма) значение операнда A уменьшается на 1. Результатом выражения A-- является исходное (до уменьшения на единицу) значение операнда A, а результатом выражения --A возвращается новое (после уменьшения на единицу) значение операнда A

При вычислении выражений на основе бинарных арифметических операторов значения операндов не изменяются.

Операторы сравнения

Операторы сравнения используются для сравнения на предмет равенства, неравенства, превышения или не превышения значений двух (обычно числовых) операндов. Результатом выражения на основе оператора сравнения является логическое значение true при истинности соответствующего соотношения, и false при его ложности. В табл. 2.3 перечислены операторы сравнения, используемые в языке Java.

Табл. 2.3. Операторы сравнения Java

Оператор	Описание
<	Оператор *меньше*. Результатом выражения вида A<B является логическое значение true, если значение операнда A меньше значения операнда B. В противном случае результатом выражения A<B является значение false
<=	Оператор *меньше или равно*. Результатом выражения вида A<=B является значение true, если значение операнда A не превышает значения операнда B. В противном случае результатом выражения A<=B является значение false
>	Оператор *больше*. Результатом выражения вида A>B является значение true, если значение операнда A больше значения операнда B. В противном случае результатом выражения A>B является значение false
>=	Оператор *больше или равно*. Результатом выражения вида A>=B является значение true, если значение операнда A не меньше значения операнда B. В противном случае результатом выражения A>=B является значение false
==	Оператор *равно*. Результатом выражения вида A==B является значение true, если значение операнда A равно значению операнда B. В противном случае результатом выражения A==B является значение false
!=	Оператор *не равно*. Результатом выражения вида A!=B является значение true, если значение операнда A отлично от значения операнда B. В противном случае результатом выражения A!=B является значение false

Про выражение на основе оператора сравнения говорят, что оно истинно, если значение выражения равно true, а если значение выражения равно false, то такое выражение называют ложным. Все операторы сравнения являются бинарными (у них два операнда).

Логические операторы

Логические операторы используются с операторами логического типа, возвращают результатом значения логического типа и предназначены для создания сложных условий. Логические операторы языка Java представлены в табл. 2.4.

Табл. 2.4. Логические операторы Java

Оператор	Описание
&	Бинарный оператор *логическое и*. Результатом выражения вида A&B является true, если значения обоих операндов A и B равны true. Если хотя бы один из операндов имеет значение false, то результатом выражения A&B является значение false
&&	Бинарный оператор *логическое и*, вычисляемый по *упрощенной схеме*. Результат выражения A&&B такой же, как и выражения A&B, но в случае с оператором && если при проверке значения операнда A оказывается, что оно равно false, то значение операнда B уже не вычисляется, а сразу результатом выражения A&&B возвращается значение false

Оператор	Описание
|	Бинарный оператор *логическое или*. Результатом выражения вида A|B является true, если значение хотя бы одного из операндов A или B равно true. Если значения обеих операндов равны false, то результатом выражения A|B возвращается значение false
||	Бинарный оператор *логическое или*, который вычисляется по *упрощенной схеме*. Результат выражения A||B совпадает со значением выражения A|B, но в случае с оператором || если при проверке значения операнда A оказывается, что оно равно true, то значение операнда B уже не вычисляется, а сразу результатом выражения A||B возвращается значение true
^	Бинарный оператор *логическое исключающее или*. Результатом выражения вида A^B является true, если один операнд имеет значение true, а другой операнд имеет значение false. Если значения обеих операндов одновременно равны false, или если их значения одновременно равны true, результатом выражения A^B возвращается значение false
!	Унарный оператор *логического отрицания*. Результатом выражения вида !A равняется значение true, если значение операнда A равно false. Если значение операнда A равно true, то результатом выражения !A является значение false

Обычно выражения на основе логических операторов используются в условном операторе или в оператора цикла, когда в программном коде создаются точки ветвления.

Побитовые операторы

Побитовые операторы предназначены для выполнения операций с целыми числами на уровне их битового (двоичного) представления.

ДЕТАЛИ

В двоичном представлении число записывается в виде последовательности нулей и единиц. Сначала определимся со способом двоичного кодирования положительных чисел. Допустим, что для записи числа используется n битов, каждый из которых может принимать значение 0 или 1. Бинарный код числа формально можно записать как $\overline{a_{n-1}a_{n-2}\ldots a_2a_1a_0}$, где параметры $a_k = 0,1$ при $k = 0, 1, 2, \ldots, n-1$. В десятичной системе счисления такое число определяется как $a_0 \cdot 2^0 + a_1 \cdot 2^1 + a_2 \cdot 2^2 + \ldots + a_{n-2} \cdot 2^{n-2} + a_{n-1} \cdot 2^{n-1}$.

Операции с числами в двоичной системе выполняются так же просто, как и в десятичной. Например, правила сложения чисел базируются на следующих соотношениях: 0 + 0 = 0, 1 + 0 = 1, 0 + 1 = 1 и 1 + 1 = 10. Отрицательные числа кодируются по принципу дополнения до нуля. Чтобы понять главную идею, предположим, что имеется некоторое положительное число x, а нам необходимо сгенерировать код для

числа $-x$. Мы будем исходить из того, что число $-x$ по определению такое, что в сумме с числом x дает ноль. Другими словами, должно иметь место соотношение $-x + x = 0$. Теперь вернемся к числу x и проделаем над ним такую операцию, как побитовая инверсия: в двоичном представлении числа x единицы поменяем на нули, а нули поменяем на единицы. Результат обозначим как $\tilde{x}$. Если теперь вычислить сумму $\tilde{x} + x$, то получим бинарный код из единиц: там, где в коде числа x единица, в числе $\tilde{x}$ ноль, и наоборот. Итак, если числа запоминаются кодом из n битов, то результатом выражения $\tilde{x} + x$ является код $\underbrace{111\ldots11}_{n}$ из n единиц. Прибавим к этому коду число 1, то есть вычислим сумму $\tilde{x} + x + 1$. Результат будет код $\underbrace{1000\ldots00}_{n}$, в котором n нулей и в старшем бите 1. Но, поскольку речь идет о компьютере и запоминании числа с помощью n битов, то старший единичный бит теряется. В результате получается код из n нулей, что соответствует нулю. Другими словами, имеет место соотношение $\tilde{x} + x + 1 = 0$. Следовательно, выражение $\tilde{x} + 1$ дает код отрицательного числа $-x$.

В двоичном побитовом представлении числа старший бит называется знаковым и определяет знак числа: у положительных чисел старший бит нулевой, а у отрицательных чисел старший бит единичный.

Чтобы по коду положительного числа получить код противоположного ему отрицательного числа, необходимо инвертировать код положительного числа и прибавить к результату единицу.

Чтобы по коду отрицательного числа определить это число в десятичной системе счисления, следует выполнить инверсию кода отрицательного числа, прибавить к коду единицу, перевести полученный код в десятичное значение (как это делается для положительных чисел) и «дописать» перед числом знак «минус».

Побитовые операторы, используемые в языке Java, перечислены в табл. 2.5.

Табл. 2.5. Побитовые операторы Java

Оператор	Описание
`&`	Бинарный оператор *побитовое и*. Результатом выражения вида `A&B` является число, бинарный код которого вычисляется так: сравниваются биты в кодах чисел `A` и `B`. Если оба бита единичные, то в соответствующем месте числа-результата будет единица. В противном случае (если хотя бы один из сравниваемых битов нулевой) в число-результат записывается нулевой бит
`|`	Бинарный оператор *побитовое или*. Результатом выражения вида `A|B` является число, бинарный код которого вычисляется сравнением битов в кодах чисел `A` и `B`. Если хотя бы один из сравниваемых битоединичный, то в соответствующем месте числа-результата будет единица.

Оператор	Описание
	В противном случае (если оба сравниваемых бита нулевые) в число-результат записывается нулевой бит
^	Бинарный оператор *побитовое исключающее или*. Результатом выражения вида A^B является число, бинарный код которого вычисляется сравнением битов в кодах чисел A и B. Если из двух сравниваемых битов один единичный, а другой нулевой, то соответствующий бит у числа-результата будет единичным. Если оба сравниваемых бита нулевые или оба они единичные, то в число-результат записывается нулевой бит
~	Унарный оператор *побитовой инверсии*. Результатом выражения ~A является число с кодом, который получается заменой в коде числа A единиц на нули и нулей на единицы
>>	Оператор *побитового сдвига вправо*. Результатом выражения вида A>>n является число с кодом, который получается смещением вправо на n позиций кода числа A. При этом младшие биты теряются, а старшие биты заполняются значением знакового бита (0 для положительных чисел и 1 для отрицательных чисел)
>>>	Оператор *беззнакового побитового сдвига вправо*. Результатом выражения вида A>>n является число с кодом, который получается смещением вправо на n позиций кода числа A. При этом младшие биты теряются, а старшие биты заполняются нулями
<<	Оператор *побитового сдвига влево*. Результатом выражения вида A<<n является число с кодом, который получается смещением влево на n позиций кода числа A. При этом младшие биты заполняются нулями, а старшие биты теряются

Некоторые побитовые операторы формально совпадают с логическими операторами. Неоднозначность разрешается исходя из типа операндов: в логических выражениях операндами являются логические величины, а в побитовых выражениях операндами являются целые числа.

Тернарный оператор

В языке Java есть один оператор, у которого три операнда. Такой оператор называется *тернарным* и формально обозначается как ?:. Синтаксис у оператора следующий (жирным шрифтом выделены ключевые элементы шаблона):

```
условие?значение:значение;
```

Сначала указывается некоторое условие (выражение с логическим значением — первый операнд), затем знак вопроса ? и второй операнд, затем двоеточие : и третий операнд. Результат вычисляется так: проверяется условие, и если оно истинно, то результатом всего выражения возвращается значение, указанное после вопросительного знака. Если условие ложно, то результатом возвращается значение, указанное после двоеточия. Например,

результатом выражения (x>0)?100:200 является значение 100 при положительном значении переменной x (если истинно условие x>0), и значение 200 при ложном значении переменной x (если ложно условие x>0).

Тернарный оператор представляет собой упрощенную форму *условного оператора*, с которым мы познакомимся немного позже.

Оператор присваивания

Оператор присваивания = в языке Java возвращает значение. Так, при выполнении команды вида A=B значение переменной B или выражения, указанного справа от оператора присваивания, присваивается переменной A, указанной слева от оператора присваивания. При этом выражение A=B имеет значение (совпадает со значением, присваиваемым переменной A) и поэтому само может быть частью более сложного выражения. Например, если предварительно переменные x, y и z объявлены с типом int, корректной является следующая команда:

```
x=y=z=100;
```

В результате выполнения этой команды переменные x, y и z получают значение 100. Обрабатывается команда так. Переменной x значением присваивается выражение y=z=100, результатом которого является значение выражения z=100, присваиваемое переменной y. Значением выражения z=100, в свою очередь является число 100, которое присваивается переменной z.

Ситуация может быть и более «запутанной». Рассмотрим следующую команду:

```
x=(x=(y=3)-(z=7))+(x=x+(y=y+1)*(z=z-1));
```

В результате выполнения этой команды переменная x получает значение 16, переменная y получает значение 4, а переменная z получает значение 6. Объяснение такое. Переменной x присваивается значение выражения (x=(y=3)-(z=7))+(x=x+(y=y+1)*(z=z-1)), представляющее собой сумму двух слагаемых (x=(y=3)-(z=7)) и (x=x+(y=y+1)*(z=z-1)). Первым вычисляется слагаемое (x=(y=3)-(z=7)). Данным выражением переменной x присваивается значение выражения (y=3)-(z=7). При вычислении выражения переменная y получает значение 3, переменная z получает значение 7, а переменная x получает значение -4 (поскольку 3 – 7 = –4). Затем вычисляется выражение (x=x+(y=y+1)*(z=z-1)). В нем переменной x присваивается значение выражения x+(y=y+1)*(z=z-1). Выражение вычисляется так:

- значение переменной y увеличивается на единицу (становится равным 4, и это значение выражения y=y+1);
- значение переменной z уменьшается на единицу (становится равным 6, и это значение выражения z=z-1);
- вычисляется результат произведения (y=y+1)*(z=z-1) (значение 24);
- к полученному значению 24 прибавляется текущее значение -4 переменной x — получаем значение 20;
- переменная x получает значение 20;
- вычисляется сумма слагаемых (x=(y=3)-(z=7)) и (x=x+(y=y+1)*(z=z-1)), что означает сумму чисел -4 и 20, и в итоге получаем число 16, которое и присваивается переменной x в качестве значения.

Общая рекомендация такая: если результат команды не совсем очевиден, то лучше такую команду разбить на несколько простых и понятных.

Сокращенные формы оператора присваивания

В Java существуют специальные сокращенные формы оператора присваивания. Идея состоит в том, что вместо команд вида A=A operator B, где через operator обозначен один из арифметических или побитовых бинарных операторов, можно использовать команду вида A operator =B. Например, вместо команды A=A+B используют команду вида A+=B. Вместо команды A=A>>n можно использовать команду A>>=n, и так далее.

НА ЗАМЕТКУ

На самом деле команда вида A operator =B эквивалентна команде A=(тип A) A operator B. Другими словами, команда вида A operator =B выполняется так: вычисляется результат выражения A operator B, полученное значение приводится к типу операнда A, после чего данное значение присваивается переменной A.

Резюме

Не надо меня щадить: пусть самое страшное, но правда.

Из к/ф «Бриллиантовая рука»

- В Java существует несколько базовых типов. Каждый тип данных определяется с помощью специального идентификатора. При работе

с целыми числами используют типы int, long, short и byte. При работе с числами в формате с плавающей точкой используются типы double и float. Символьные значения относятся к типу char, а логические значения относятся к типу boolean.

- При объявлении переменной указывается ее тип и название. Если несколько переменных относятся к одному типу, они могут объявляться одной командой с общим идентификатором типа, а имена переменных разделяются запятыми. Для переменных при объявлении можно указывать значения. Значение переменной может определяться на основе выражения. Доступность переменной определяется блоком, в котором она объявлена.
- Значения переменных можно определять путем считывания через поле ввода или через консольное окно. Для преобразования текстовых значений в числовые используют такие статические методы, как parseInt() или parseDouble() соответственно из классов-оболочек Integer и Double. При считывании значений с консоли используют такие методы, как nextLine(), nextInt() и nextDouble(), которые вызываются из объекта класса Scanner.
- В Java есть несколько групп операторов (арифметические, логические, побитовые и операторы сравнения), которые позволяют выполнять всевозможные операции с переменными разных типов. Оператор присваивания возвращает значение, а также существуют сокращенные формы оператора присваивания. Помимо этого, в Java существует механизм приведения типов, который следует учитывать при составлении программных кодов.

Глава 3
ЗНАКОМСТВО С КЛАССАМИ И ОБЪЕКТАМИ

Он начинает новую жизнь. Дайте ему возможность вспомнить все лучшее.

Из к/ф «Покровские ворота»

С терминами *класс* и *объект* мы уже неоднократно сталкивались. Теперь пришло время познакомиться с классами и объектами более основательно.

Классы и объекты

История, леденящая кровь. Под маской овцы скрывался лев!

Из к/ф «Покровские ворота»

В первую очередь нам предстоит разобраться с тем, что такое класс, что представляет собой объект, чем они отличаются и что их связывает между собой. Для этого мы прибегнем к аналогии.

Представим себе фабрику, на которой производится деревянная мебель: столы, тумбочки, стулья. В общих чертах процесс выглядит так: имеется некоторый чертеж или эскиз деревянного изделия, и по этому эскизу, собственно, и собирается вся «конструкция». Для большей конкретики пускай речь идет о сборке компьютерного стола. Чертеж такого «изделия» содержит всю необходимую информацию: размер стола, его высота, наличие, количество и расположение внутренних выдвижных ящиков, и другие характеристики. Если у нас имеется чертеж, то по этому чертежу можно сделать сколько угодно столов: один, десять, тридцать или вообще ни одного. Все столы, выполненные по одному эскизу, будут однотипными. И при этом каждый из них физически уникален. Мы можем покрасить столы в разный цвет, поставить их в разных комнатах, положить в их внутренние ящики разные предметы.

Аналогом чертежа, на основе которого создается реальная конструкция, выступает *класс*. На основе класса, как на основе эскиза, создаются *объекты*. Что же такое объект? Объект, если кратко, представляет собой набор переменных (разных типов, в том числе это могут быть и объекты других классов), а также код, предназначенный для обработки этих переменных. Переменные, входящие в состав объекта, называются его *полями*. Код, предназначенный для обработки полей объекта (и не только), реализуется в виде *методов* объекта. Метод объекта представляет собой именованный блок команд, которые выполняются при вызове метода.

Описание класса с полями

Судить о том, какие у объекта есть поля и методы, следует по описанию класса, на основе которого создается объект. Другими словами, в описании класса содержится информация о том, какие у объектов, создаваемых на основе данного класса, есть поля и методы. По большому счету, описание класса представляет собой описание полей и методов. Впоследствии, когда на основе класса будут созданы объекты, чтобы понять, какие у них есть поля и методы, достаточно взглянуть на описание класса.

Описание класса начинается с ключевого слова class, после которого указывается имя класса. Описание класса выполняется в блоке из фигурных скобок. Общий шаблон описания класса такой (жирным шрифтом выделены ключевые элементы шаблона):

```
class имя_класса{
  // Описание класса
}
```

Для начала мы рассмотрим классы, которые содержат только поля (и не содержат методов). Поэтому при описании класса речь пока что идет об описании полей. А поля описываются очень просто: указывается тип поля и его имя. То есть фактически поля описываются точно так же, как и переменные. Разница лишь в том, что поле жестко привязано к объекту, который создается на основе класса.

> **НА ЗАМЕТКУ**
>
> Существуют статические поля (и методы), которые существуют вне контекста объектов и «привязаны» непосредственно к классу. Такие поля и методы мы пока не рассматриваем, а обсудим их чуть позже.

Ниже приведен пример описания класса, который называется MyClass. В классе описываются два поля: целочисленное (тип int) поле number и символьное (тип char) поле symbol. Класс описывается следующим образом:

```
class MyClass{
  int number;
  char symbol;
}
```

При создании объекта для каждого из его полей в памяти выделяется место, так что у каждого объекта поля свои, так сказать, «персональные». Поля, как отмечалось, играют роль переменных. Но поскольку поле является «переменной внутри объекта», то, во-первых, нет смысла говорить о поле без указания объекта и, во-вторых, поля с одинаковыми названиями, но для разных объектов, являются совершенно независимыми.

Любой объект, созданный на основе класса MyClass, будет иметь поле number и поле symbol. Осталось только выяснить, как создать объект на основе класса.

Создание объекта

Процесс создания объекта можно условно разделить на два этапа:

- объявление *объектной переменной*, через которую будет осуществляться доступ к объекту;
- собственно создание *объекта* и «связывание» объекта с объектной переменной.

Объявление объектной переменной аналогично объявлению переменной базового типа, но только для объектной переменной в качестве идентификатора типа указывается название класса. В частности, если мы хотим объявить объектную переменную класса MyClass, то соответствующая команда выглядит следующим образом:

```
MyClass obj;
```

Здесь объявляется объектная переменная с именем obj класса MyClass. Принадлежность объектной переменной к определенному классу означает,

что данная переменная может ссылаться на объект соответствующего класса. Но важно понимать, что объявление объектной переменной не означает создания объекта. То есть переменная существует, под нее выделяется место в памяти. Но это не объект. Объект создается с помощью оператора new, после которого указывается имя класса, на основе которого создается объект, и круглые скобки — в нашем случае пустые. Более конкретно, для создания объекта класса MyClass, используем инструкцию new MyClass().

> **НА ЗАМЕТКУ**
>
> На самом деле после оператора new указывается конструктор создания объекта. Это специальный метод, который вызывается как раз при создании объекта. Название конструктора совпадает с названием класса. В круглых скобках передаются аргументы конструктору (если они нужны). Если аргументов у конструктора нет, то круглые скобки пустые. Это как раз наш случай. Но, поскольку мы с конструкторами еще не познакомились, то временно можем просто полагать, что после оператора new указывается имя класса и пустые круглые скобки.

При выполнении данной инструкции создается новый объект класса MyClass, а значением выражения new MyClass() является *ссылка* на созданный объект. Обычно такую ссылку присваивают значением объектной переменной. Поэтому в нашем случае, после создания объектной переменной obj, объект мог бы создаваться с помощью такой команды:

```
obj=new MyClass();
```

Нередко команду объявления объектной переменной и команду создания объекта объединяют в одну инструкцию. Так, вместо команд MyClass obj и obj=new MyClass() мы могли бы использовать следующую команду:

```
MyClass obj=new MyClass();
```

Теперь инициализация объектной переменной происходит одновременно с ее объявлением.

Некоторое представление о роли объектной переменной в получении доступа к объекту дает рис. 3.1.

Разницу между объектной переменной и объектом легче понять, если исходить из представления, что значением объектной переменной

является «адрес» объекта. Как альтернативу к «адресу» мы будем использовать выражение «ссылка на объект». Или, попросту говоря, объектная переменная «знает», где в памяти хранится объект и имеет к нему доступ.

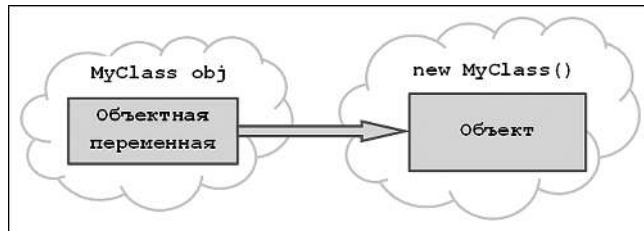


Рис. 3.1. Объектная переменная ссылается на объект

НА ЗАМЕТКУ

Еще раз подчеркнем: значение объектной переменной — ссылка на объект (адрес объекта). При объявлении объектной переменной под нее выделяется место в памяти. Но туда, в переменную, следует еще записать ссылку. Ссылку получаем, когда создаем объект.

Если в объектную переменную записана ссылка на некоторый объект, мы будем говорить, что переменная ссылается на этот объект. Во многих случаях, если это не будет приводить к недоразумениям, объектную переменную будем отождествлять с объектом. Однако для понимания принципов работы с объектами все же важно помнить, что объектная переменная является лишь «посредником» при получении доступа к объекту.

Использование объектов

Далее рассмотрим небольшой пример, в котором описывается класс и на основе этого класса в главном методе программы создается объект. В процессе работы с объектом выполняются обращения к полям объекта (для присваивания значений и считывания значений полей). Правило обращения к полям объекта простое: используется так называемый точечный синтаксис, когда при обращении к полю объекта сначала указывается имя объекта и, через точку, имя поля. Это, так сказать, общий подход. Теперь рассмотрим программный код в листинге 3.1.

Листинг 3.1. Программный код проекта UsingObjectApplication

```
import javax.swing.JOptionPane;
// Описание класса:
class MyClass{
  // Поля класса:
  int number;
  char symbol;
}
// Описание класса с главным методом программы:
class UsingObjectDemo{
  // Главный метод программы:
  public static void main(String[] args){
    // Создание объекта:
    MyClass obj=new MyClass();
    // Присваивание значений полям объекта:
    obj.number=100;
    obj.symbol='A';
    // Текст для отображения в диалоговом окне:
    String text="Число:  "+obj.number+"\n";
    text+="Символ:  "+obj.symbol;
    // Отображение диалогового окна:
    JOptionPane.showMessageDialog(null,text);
  }
}
```

В результате выполнения программы отображается диалоговое окно, представленное на рис. 3.2.

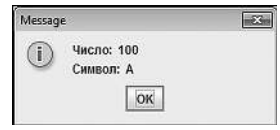


Рис. 3.2. В диалоговом окне отображаются значения полей объекта

Программа очень простая. Сначала описывается класс MyClass с двумя полями: полем number типа int и полем symbol типа char. Метод main() описан в другом классе, который называется UsingObjectDemo. В теле метода командой MyClass obj=new MyClass() создается объект класса MyClass. Ссылка на объект записывается в объектную переменную obj, которую мы отождествляем с объектом. У объекта obj, поскольку он создан на основе класса MyClass, есть поля number и symbol. Этим полям следует присвоить значения, что мы и делаем с помощью команд obj.number=100 и obj.symbol='A'. Здесь при обращении к полям мы сначала указываем объект, а имя поля указывается через точку после имени объекта. После того, как значения полям объекта obj присвоены, формируется значение текстовой переменной text, и данное значение определяет текст сообщения, отображаемого в диалоговом окне. В частности, там отображаются значения полей number и symbol объекта obj.

Класс с методами

Кроме полей, в классе могут описываться *методы*. Метод представляет собой блок команд, которые выполняются при вызове метода. При описании метода, кроме собственно команд, указывается тип возвращаемого методом результата, имя метода и список аргументов. Тип возвращаемого методом результата определяется идентификатором типа (например, int, double или char). Если метод не возвращает результат, то идентификатором типа указывается ключевое слово void.

Название метода выбираем на свое усмотрение, но оно по возможности должно быть информативным и не совпадать с зарезервированными ключевыми словами.

Список аргументов метода перечисляется в круглых скобках после имени метода. Для каждого аргумента указывается тип и название. Тело метода (команды, выполняемые при вызове метода) заключается в фигурные скобки.

НА ЗАМЕТКУ

Идентификатор типа результата метода, имя метода и список его аргументов формируют *сигнатуру метода*.

В самом методе могут использоваться *локальные переменные*. Объявляются локальные переменные метода точно так же, как и переменные

в методе main(), но с поправкой на то, что теперь речь идет о другом методе и, как следствие, такие переменные доступны только в теле метода (в котором они объявлены). Создаются локальные переменные при вызове метода, а по его завершении удаляются из памяти. В этом смысле они принципиально отличаются от полей объекта, которые существуют, пока существует объект.

К полям объекта метод может обращаться «напрямую» так, как если бы это были переменные, объявленные в методе. Однако следует четко понимать, что описанное в классе поле доступно всем методам из этого класса, а локальная переменная доступна только в теле метода, в котором она объявлена.

Небольшой пример, в котором показано, как описываются классы с методами и как затем эти методы вызываются из объектов, представлен в листинге 3.2.

Листинг 3.2. Программный код проекта UsingObjectWithMethodsApplication

```
import javax.swing.JOptionPane;
// Описание класса:
class MyClass{
  // Поля класса:
  int number;
  char symbol;
  // Метод для присваивания значений полям:
  void set(int n,char s){
    number=n;
    symbol=s;
  }
  // Методом возвращается текстовая строка
  // с описанием объекта:
  String getInfo(){
    // Текст, который возвращается
    // результатом метода:
    String text="Число:  "+number+"\n";
    text+="Символ:  "+symbol;
```

```
    // Результат метода:
    return text;
  }
}
// Описание класса с главным методом программы:
class UsingObjectWithMethodsDemo{
  // Главный метод программы:
  public static void main(String[] args){
    // Создание первого объекта:
    MyClass objA=new MyClass();
    // Создание второго объекта:
    MyClass objB=new MyClass();
    // Присваивание значений полям первого объекта:
    objA.set(100,'A');
    // Присваивание значений полям второго объекта:
    objB.set(200,'B');
    // Отображение первого диалогового окна:
    JOptionPane.showMessageDialog(null,
      objA.getInfo(),  // Отображаемый текст
      "Первый объект", // Заголовок окна
      JOptionPane.INFORMATION_MESSAGE); // Тип окна
    // Отображение второго диалогового окна:
    JOptionPane.showMessageDialog(null,
      objB.getInfo(),  // Отображаемый текст
      "Второй объект", // Заголовок окна
      JOptionPane.INFORMATION_MESSAGE); // Тип окна
  }
}
```

При запуске программы на выполнение последовательно появляются два диалоговых окна. Каждое из окон содержит информацию о значениях полей объектов, создаваемых на основе класса MyClass в программе. На рис. 3.2 представлено первое окно.

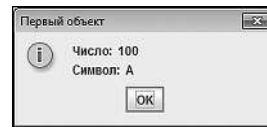


Рис. 3.2. Диалоговое окно со значениями полей первого объекта

После того, как первое окно закрывается, отображается второе окно, как показано на рис. 3.3.

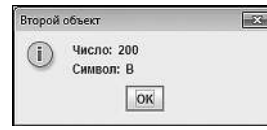


Рис. 3.3. Диалоговое окно со значениями полей второго объекта

Что касается непосредственно программного кода, то в нем описаны два класса. В классе MyClass есть два поля (поле number типа int и поле symbol типа char). Еще в классе MyClass описано два метода. Метод set() предназначен для присваивания значений полям объекта, из которого метод вызывается.

У метода два аргумента: первый аргумент обозначен как n и относится к типу int, а второй аргумент называется s и относится к типу char. Метод не возвращает результат. Поэтому в качестве идентификатора типа для метода указано ключевое слово void. В теле метода командами number=n и symbol=s полям number и symbol присваиваются значения. Таким образом, аргументы метода set() определяют значения полей объекта, из которого вызывается метод.

Методом getInfo() результатом возвращается текстовая строка, содержащая значения полей number и symbol объекта, из которого вызывается метод. Аргументов у метода getInfo() нет.

Для формирования текстовой строки, возвращаемой результатом метода, в теле метода объявляется локальная переменная text типа String (на самом деле это объектная переменная text класса String). Начальное значение переменной text равно "Число: "+number+"\n".

Данная текстовая строка содержит значение поля number. Затем с помощью команды text+="Символ: "+symbol к текущему текстовому значению добавляется текст со значением поля symbol.

НА ЗАМЕТКУ

В теле методов getInfo() и set() инструкции number и symbol означают обращение к полям того объекта, из которого вызывается метод. Таким образом, метод при вызове «оперирует» с полями «своего» объекта. Метод имеет «прямой» и «непосредственный» доступ к полям объекта.

Командой return text значение переменной text возвращается результатом метода getInfo().

ДЕТАЛИ

Процесс формирования текстового значения и возвращения его результатом метода не такой тривиальный, как может показаться на первый взгляд. При объявлении локальной переменной text в теле метода getInfo() на самом деле создается объектная переменная. Текстовое значение "Число: "+number+"\n", которым инициализируется переменная text, реализуется в виде объекта класса String и ссылка на него записывается в переменную text. При выполнении команды text+="Символ: "+symbol происходит следующее: на основе текущего текстового значения объекта, на который ссылается переменная text, и текстового значения "Символ: "+symbol путем их объединения создается новый текстовый объект, ссылка на этот объект записывается в переменную text. Таким образом, при выполнении операций с текстом каждый раз создается новый объект, но поскольку доступ к конечному текстовому объекту выполняется через одну и ту же объектную ссылку text, создается иллюзия, что мы «изменяем» текстовое значение.

Нечто похожее происходит при возвращении результата метода getInfo(). Результатом метода, напомним, возвращается переменная text. Если вспомнить, что на самом деле значением переменной text является ссылка на текстовый объект, то становится понятно, что по факту результатом метод возвращает ссылку на тот текстовый объект, на который ссылается переменная text.

На этом описание класса MyClass заканчивается. В классе UsingObjectWithMethodsDemo описывается метод main(). Там командами MyClass objA=new MyClass() и MyClass objB=new MyClass() создаются два объекта objA и objB класса MyClass. После создания объектов их полям значения не присвоены. Присваивание значений полям объектов выполняется командами objA.set(100,'A') и objB.set(200,'B'). Аргументы, переданные методу set() при вызове, присваиваются значениями полям того объекта, из которого вызывается метод. Поэтому поля number и symbol объекта objA получают

значения соответственно 100 и 'A', а поля number и symbol объекта objB получают значения соответственно 200 и 'B'.

Для отображения значений полей каждого из объектов отображается диалоговое окно. Окна отображаются последовательным вызовом статического метода showMessageDialog() из класса JOptionPane. Команды вызова метода значением второго и третьего аргументов, которые определяют текст сообщения и название окно. Текст для отображения в диалоговом окне получаем с помощью выражений objA.getInfo() (для первого объекта) и objB.getInfo() (для второго объекта). Здесь мы используем метод getInfo(), результатом которого является текст со значениями полей объекта, из которого вызывается метод.

Методы и конструкторы

— Мастера не мудрствуют.
— А могильщики в «Гамлете»?
— Ремесленники!

Из к/ф «Покровские ворота»

Далее мы рассмотрим некоторые аспекты, связанные с использованием методов, а также познакомимся с *конструкторами*.

Перегрузка методов

В Java в одном и том же классе можно описывать несколько методов с одинаковыми названиями. Различаются такие методы типом аргументов и/или их количеством. Такой механизм называется *перегрузкой методов*. Перегрузка методов — очень мощный инструмент, позволяющий создавать эффективные и гибкие программные коды.

ДЕТАЛИ

Создавая методы с одинаковыми названиями, мы создаем реально разные методы. При вызове метода с данным названием решение о том, какой из методов на самом деле вызывается, принимается исходя из контекста команды вызова метода — например, по количеству переданных методу аргументов или их типу. Вместе с тем, с формальной точки зрения, при вызове таких методов создается иллюзия, что вызывается один и тот же метод, но с разными аргументами. В этом смысле можно говорить о разных версиях одного метода.

Как иллюстрацию, рассмотрим программу, представленную в листинге 3.3. Программа представляет собой вариацию на тему предыдущего примера (из листинга 3.2), но на этот раз в программе используется перегрузка методов.

Листинг 3.3. Программный код проекта MethodOverloadingApplication

```
// Описание класса:
class MyClass{
  // Поля класса:
  int number;
  char symbol;
  // Метод с одним аргументом для присваивания
  // значения полю number:
  void set(int n){
    number=n;
  }
  // Метод с одним аргументом для присваивания
  // значения полю symbol:
  void set(char s){
    symbol=s;
  }
  // Метод с двумя аргументами для присваивания
  // значений полям number и symbol:
  void set(int n,char s){
    // Присваивание значения полю number:
    set(n);
    // Присваивание значения полю symbol:
    set(s);
  }
  // Метод без аргументов для присваивания значений
  // обоим полям:
  void set(){
    // Присваивание значения 0 полю number
```

```
    // и значения 'Z' полю symbol:
    set(0,'Z');
  }
  // Метод без аргументов для отображения
  // значений полей объекта:
  void show(){
    System.out.println("Значения полей "+number+" и "+symbol);
  }
  // Метод с одним аргументом для отображения
  // значений полей объекта:
  void show(String txt){
    System.out.println(txt+": значения полей "+number+" и "+symbol);
  }
  // Метод с двумя аргументами для отображения
  // значений полей объекта:
  void show(String txt1,String txt2){
    System.out.println(txt1+": "+number);
    System.out.println(txt2+": "+symbol);
  }
}
// Описание класса с главным методом программы:
class MethodOverloadingDemo{
  // Главный метод программы:
  public static void main(String[] args){
    // Объявление объектных переменных:
    MyClass objA,objB;
    // Создание объектов:
    objA=new MyClass();
    objB=new MyClass();
    // Присваивание значений полям первого объекта:
    objA.set(100);
    objA.set('A');
```

```
    // Отображение значений полей первого объекта:
    System.out.println("Объект objA:");
    objA.show();
    // Присваивание значений полям второго объекта:
    objB.set();
    // Отображение значений полей второго объекта:
    objB.show("Объект objB");
    // Изменение значения полей второго объекта:
    objB.set(200,'B');
    // Проверка значений полей второго объекта:
    System.out.println("Объект objB после изменения:");
    objB.show("Число","Символ");
  }
}
```

В результате выполнения программы в окне вывода появляются следующие сообщения:

Результат выполнения программы (из листинга 3.3)

```
Объект objA:
Значения полей 100 и A
Объект objB: значения полей 0 и Z
Объект objB после изменения:
Число: 200
Символ: B
```

Проанализируем особенности программного кода. В программе описываются два класса: в классе MethodOverloadingDemo описывается главный метод программы, а в классе MyClass, помимо полей number и symbol, есть перегруженные методы set() и show(). Именно перегруженные методы представляют наибольший интерес.

В классе MyClass описано четыре версии метода set(). Каждая из них не возвращает результат, а отличие сводится к количеству и типу аргументов. В частности:

- предусмотрена версия метода без аргументов;
- версия с одним аргументом типа int;
- версия с одним аргументом типа char;
- имеется версия метода с двумя аргументами (первый типа int, а второй типа char).

> **НА ЗАМЕТКУ**
>
> Еще раз подчеркнем, что технически каждая версия метода является отдельным методом. Мы, тем не менее, если это не приводит к недоразумениям, будем называть их версиями одного метода.

В версии метода set() с одним целочисленным (тип int) аргументом (обозначен как n) выполняется присваивание значения полю number (команда number=n в теле метода). Аналогично «устроена» версия метода set() с одним символьным (тип char) аргументом (обозначен как s): в теле метода командой symbol=s присваивается значение полю symbol.

Метод set() с двумя аргументами описывается одновременно и просто, и на первый взгляд немного странно: в теле метода set() с двумя аргументами (обозначены как n и s) последовательно выполняются команды set(n) и set(s). Другими словами, при вызове метода set() с двумя аргументами последовательно вызывается метод set() с каждым из этих аргументов. Ситуация может показаться запутанной, но если вспомнить, что каждая версия перегруженного метода на самом деле является отдельным методом, то очевидно, что ничего сложного или необычного здесь нет: в теле одного метода вызываются два других метода. Просто в данном случае эти методы имеют одинаковые названия.

У метода set() есть еще одна версия, которая не подразумевает передачу аргументов методу при вызове. В теле метода командой set(0,'Z') вызывается метод set() с аргументами 0 и 'Z' соответственно. То есть ситуация повторяется: в теле одной версии метода вызывается другая версия метода. Таким образом, если метод set() вызывается без аргументов, то это эквивалентно вызову метода set() с аргументами 0 и 'Z'.

> **НА ЗАМЕТКУ**
>
> На ситуацию с определением версии метода set() без аргументов можно посмотреть несколько под иным углом зрения. Так, если метод

set() вызывается с двумя аргументами, то переданные методу аргументы определяют значения полей объекта, из которого вызывается метод. Если метод set() вызывается без аргументов, то полям объекта присваиваются определенные значения (в данном случае 0 и 'Z'), которые можно интерпретировать как значения полей по умолчанию.

Для отображения значений полей объекта в консольном окне (или окне вывода) предназначен метод show(), описанный в классе MyClass в трех вариантах: без аргументов, с одним текстовым аргументом и с двумя текстовыми аргументами. В каждом из этих случаев результат методом не возвращается (поэтому в описании метода перед именем метода указано ключевое слово void).

В теле версии метода show() без аргументов выполняется всего одна команда System.out.println("Значения полей "+number+" и "+symbol), которой в окно вывода выводится сообщение со значениями полей объекта (из которого вызывается метод show()).

Если метод show() вызывается с одним текстовым аргументом (обозначен в описании метода как txt), то командой System.out.println(txt+": значения полей "+number+" и "+symbol) в окно вывода выводится содержимое текстового аргумента и, через двоеточие, текст, содержащий значения полей объекта.

Наконец, версия метода show() с двумя текстовыми аргументами (обозначены как txt1 и txt2) предполагает, что при вызове метода командами System.out.println(txt1+": "+number) и System.out.println(txt2+": "+symbol) в двух строках отображаются значения аргументов и, через двоеточие, значения полей объекта.

В главном методе программы в классе MethodOverloadingDemo командой MyClass objA,objB объявляются две объектные переменные objA и objB класса MyClass. Объекты класса MyClass создаются командами objA=new MyClass() и objB=new MyClass(), а ссылки на объекты записываются в объектные переменные objA и objB. Дальнейшие операции связаны с присваиванием значений полям объектов и проверкой значений полей. В частности, командой objA.set(100) полю number объекта objA присваивается значение 100, а командой objA.set('A') полю symbol этого же объекта присваивается значение 'A'.

ДЕТАЛИ

При выполнении команды с вызовом метода set(), версия метода, которая должна выполняться, идентифицируется по количеству и типу

аргументов, переданных методу. Например, в команде objA.set(100) вызывается метод set() с одним аргументом, и этот аргумент типа int. Поэтому выполняется версия метода set(), описанная с одним целочисленным аргументом. Далее, в команде objA.set('A') методу set() передается один аргумент типа char, поэтому выполняется та версия метода set(), что описана с одним символьным аргументом. Прочие ситуации обрабатываются подобным же образом.

Для проверки значений полей объекта objA из данного объекта вызываем метод show() без аргументов (команда objA.show()).

Значения полям объекта objB присваиваются при выполнении команды objB.set(). В результате поле number объекта objB получает значение 0, а поле symbol получает значение 'Z'. В справедливости данного утверждения убеждаемся с помощью команды objB.show("Объект objB"), в которой из объекта objB вызывается версия метода show() с одним текстовым аргументом. Далее выполняется команда objB.set(200,'B'). Здесь метод set() вызывается с двумя аргументами, в результате чего number объекта objB получает значение 200, а поле symbol получает значение 'A'. Значения полей проверяем посредством команды objB.show("Число","Символ"), в которой метод show() вызывается с двумя текстовыми аргументами.

Конструктор

В рассмотренном выше примере мы для присваивания значений полям объекта в классе описали специальный метод (даже несколько версий). Использование такого метода значительно упрощает процесс присваивания значений полям. Однако существует еще более простой и удобный механизм, позволяющий выполнять всевозможные операции (включая присваивание значений полям объекта) непосредственно при создании объекта. Базируется данный механизм на использовании *конструктора*.

Конструктором называется метод, вызываемый автоматически при создании объекта класса. В конструкторе определяются дополнительные (помимо выделения памяти под объект) действия, которые следует выполнить при создании объекта. Если конструктор в классе явно не описан (как это было в рассмотренных ранее примерах), то используется так называемый *конструктор по умолчанию*. Действие конструктора по умолчанию состоит в том, что никаких дополнительных действий не выполняется.

С формальной точки зрения конструктор описывается практически так же, как и любой другой метод, но есть некоторые «особенности». Вот они.

- Имя конструктора совпадает с именем класса.
- Конструктор не возвращает результат, но при этом ключевое слово void в сигнатуре конструктора не указывается.
- У конструктора могут быть аргументы и конструктор можно перегружать — другими словами, в классе может быть описано несколько конструкторов.

Если конструктор описан с аргументами, то при создании объекта их нужно передать конструктору. В таком случае в инструкции создания объекта после ключевого слова new и имени класса в круглых скобках указываются аргументы конструктора. В свете этого обстоятельства шаблон команды создания объекта выглядит таким образом:

```
Имя_класса переменная=new Конструктор(аргументы);
```

Мы использовали такую конструкцию и ранее, но круглые скобки в инструкции создания объекта были пустыми (использовался конструктор по умолчанию, у которого нет аргументов).

НА ЗАМЕТКУ

Таким образом, то, что мы ранее называли «имя класса и пустые круглые скобки» после инструкции new на самом деле является командой вызова конструктора.

Если в классе описано несколько конструкторов, то какой из них следует вызвать при создании объекта, определяется по типу и количеству переданных конструктору аргументов — то есть точно так же, как и при использовании перегруженного метода. Как иллюстрацию к использованию конструкторов рассмотрим пример в листинге 3.4.

Листинг 3.4. Программный код проекта UsingConstructorApplication

```
// Описание класса:
class MyClass{
  // Поля класса:
  int number;
  char symbol;
```

```
  // Конструктор класса без аргументов:
  MyClass(){
    // Присваивание значений полям:
    number=100;
    symbol='A';
  }
  // Конструктор класса с двумя аргументами:
  MyClass(int n,char s){
    // Присваивание значений полям:
    number=n;
    symbol=s;
  }
  // Метод для отображения значений полей объекта:
  void show(){
    System.out.println("Значения полей "+number+" и "+symbol);
  }
}
// Описание класса с главным методом программы:
class UsingConstructorDemo{
  // Главный метод программы:
  public static void main(String[] args){
    // Создание первого объекта
    // (вызывается конструктор без аргументов):
    MyClass objA=new MyClass();
    // Создание второго объекта
    // (вызывается конструктор с двумя аргументами):
    MyClass objB=new MyClass(200,'B');
    // Отображение значений полей первого объекта:
    System.out.println("Объект objA:");
    objA.show();
    // Отображение значений полей второго объекта:
    System.out.println("Объект objB:");
    objB.show();
  }
}
```

В результате выполнения программы получаем следующее:

Результат выполнения программы (из листинга 3.4)

```
Объект objA:
Значения полей 100 и A
Объект objB:
Значения полей 200 и B
```

В программе описывается класс MyClass с двумя полями (поле number типа int и поле symbol типа char), методом show() без аргументов, предназначенным для отображения в консоли значений полей объекта, а также двумя версиями конструктора — без аргументов и с двумя аргументами.

Метод show() не возвращает результат и при вызове метода в консольное окно выводится сообщение с информацией о значениях полей объекта.

Конструктор класса MyClass без аргументов описывается так (комментарии удалены):

```
MyClass(){
  number=100;
  symbol='A';
}
```

Название конструктора совпадает с названием класса, идентификатор типа результата не указывается, аргументов у конструктора нет. В теле конструктора есть две команды number=100 и symbol='A'. Командами присваиваются значения полям объекта. Следовательно, если объект создается с вызовом конструктора без аргументов, то поле number такого объекта получит значение 100, а поле symbol будет иметь значение 'A'.

Конструктор с двумя аргументами имеет такой код (комментарии удалены):

```
MyClass(int n,char s){
  number=n;
  symbol=s;
}
```

В соответствии с кодом, аргументы конструктора определяют значения полей создаваемого объекта. Примеры использования

конструкторов находим в главном методе программы, описанном в классе UsingConstructorDemo.

Первый объект создается командой MyClass objA=new MyClass(). Здесь при создании объекта вызывается конструктор без аргументов. Поэтому у объекта, на который ссылается объектная переменная objA, поле number получает значение 100, а поле symbol получает значение 'A'. Второй объект создается командой MyClass objB=new MyClass(200,'B'). При создании объекта, ссылка на который записывается в объектную переменную objB, вызывается конструктор с двумя аргументами. Первый аргумент конструктора 200 определяет значение поля number объекта objB, а второй аргумент конструктора 'B' задает значение поля symbol объекта objB. Для проверки значений полей объектов из каждого объекта вызывается метод show().

ДЕТАЛИ

Как отмечалось выше, если в классе не описан конструктор, то используется конструктор по умолчанию, у которого нет аргументов, и которым никакие дополнительные действия при создании объекта не выполняются. Но если только в классе описан хотя бы один конструктор, то конструктор по умолчанию перестает быть доступен. Поэтому, например, если в классе не описан конструктор, то при создании объекта вызывается конструктор по умолчанию без аргументов. Если же в классе описан конструктор с аргументом (или аргументами), то при создании объекта придется передавать аргументы конструктору. Создать объект без передачи аргументов конструктору не получится. Чтобы такой режим был доступен, в классе должен быть описан конструктор без аргументов. Более конкретно, если бы в рассмотренном выше примере в классе MyClass не был описан конструктор без аргументов, то каждый раз при создании объекту приходилось бы передавать два аргумента.

Статические и закрытые члены класса

У папы там совсем не то, что он всем говорит.

Из к/ф «Бриллиантовая рука»

Поля и методы, описанные в классе, называются *членами класса*. Мы, хотя этого явно и не констатировали, рассматривали *открытые* и *нестатические* члены класса. Но члены класса могут быть еще и *закрытыми* и/или *статическими*. Это и будет предметом дальнейшего обсуждения.

Статические поля и методы

Выше мы отмечали, что описанные в классе поля при создании объекта фактически становятся переменными «внутри» объекта. Методы же «прикреплены» к объекту, из которого вызываются, и имеют непосредственный доступ к полям объекта. При этом неявно подразумевалось, что речь идет о *нестатических* полях и методах. Поля и методы могут быть *статическими*.

Основная идея статических членов класса состоит в том, что статический член класса «общий» для всех объектов этого класса и существует независимо от наличия или отсутствия в программе объектов класса. Поэтому обращаться к статическому члену можно не только через объект класса, но и непосредственно через класс. Причем последний способ является предпочтительным.

При описании статических членов (полей и методов) используют ключевое слово static. Если речь идет о статическом поле, то его в описании класса можно инициализировать — присвоить значение поля непосредственно в классе. Обращение к статическим полям и методам может выполняться так же, как и к обычным полям и методам, но обычно вместо имени объекта используется имя класса. В частности, при обращении к статическому полю указывают имя класса и через точку — имя статического поля. При вызове статического метода указываем имя класса и, через точку, имя метода с аргументами (или без) в круглых скобках.

Статические поля в некотором смысле играют роль глобальных переменных. Правда, при обращении к такой переменной приходится указывать имя класса (с другой стороны, и объект создавать нет необходимости). Аналогично, статические методы нередко служат «заменителем» глобальных функций — с поправкой на то, что статический метод вызывается с указанием класса, в котором метод описан.

НА ЗАМЕТКУ

Мы уже имели дело и со статическими полями, и со статическими методами. Так, метод main() является статическим. Статические методы showMessageDialog() и showInputDialog() класса JOptionPane использовались нами для отображения диалоговых окон. Статические константы, WARNING_MESSAGE, ERROR_MESSAGE, QUESTION_MESSAGE, INFORMATION_MESSAGE и PLAIN_MESSAGE класса JOptionPane использовались для определения типа диалогового окна.

Небольшой пример, в котором иллюстрируется использование статических полей и методов, представлен в листинге 3.5.

Листинг 3.5. Программный код проекта UsingStaticMembersApplication

```
// Класс со статическими членами:
class MyClass{
  // Статическое поле:
  static int count=0;
  // Конструктор без аргументов:
  MyClass(){
    // Увеличение значения статического поля:
    count++;
    // Отображение сообщения:
    System.out.println("Создан объект номер "+count);
  }
  // Статический метод:
  static void show(){
    System.out.println("Количество объектов: "+count);
  }
}
// Класс с главным методом программы:
class UsingStaticMembersDemo{
  // Главный метод программы:
  public static void main(String[] args){
    // Вызов статического метода:
    MyClass.show();
    // Создание объектов:
    MyClass objA=new MyClass();
    MyClass objB=new MyClass();
    MyClass objC=new MyClass();
    // Вызов статического метода через ссылку на класс:
    MyClass.show();
    // Вызов статического метода через ссылку на объект:
```

```
    objC.show();
    objB.show();
  }
}
```

В программе описывается класс MyClass, у которого есть статическое целочисленное поле count, конструктор без аргументов и статический метод show(). Поле count инициализируется с начальным нулевым значением. При вызове статического метода show() командой System.out.println("Количество объектов: "+count) в консольном окне отображается сообщение с указанием значения статического поля count. Статическое поле count изменяется при вызове конструктора. В теле конструктора командой count++ на единицу увеличивается значение статического поля, после чего командой System.out.println("Создан объект номер "+count) отображается сообщение в окне вывода.

Таким образом, статическое поле count используется для подсчета созданных в программе объектов класса MyClass.

НА ЗАМЕТКУ

Правда, никто не запрещает нам в программе (в главном методе программы) изменить значение поля count так, что оно перестанет соответствовать количеству созданных объектов. Но если таких «исключительных» действий не предпринимать, то каждый раз при создании объекта класса MyClass значение статического поля count увеличивается на единицу. В результате значение поля count определяет количество созданных в программе объектов.

При создании объекта каждый раз появляется сообщение о том, что создан новый объект с указанием номера объекта (совпадает с количеством созданных объектов). Узнать значение поля count можно обратиться непосредственно к полю или вызвать статический метод show().

В главном методе программы командой MyClass.show() вызывается статический метод show(). В результате вызова метода в консольном окне появляется сообщение о том, что количество объектов нулевое. На момент вызова метода еще ни один объект не создан. Тем не менее статический метод show(), равно как и статическое поле count, «существуют», и их можно использовать, даже не создавая объект. При обращении к методу show() мы перед именем метода указываем имя класса.

Далее последовательно создаются три объекта objA, objB и objC. Каждый раз при создании объекта вызывается конструктор. При вызове конструктора появляется сообщение о создании объекта и номере объекта (соответствует количеству созданных объектов). После создания трех объектов командой MyClass.show() снова вызывается метод show(). Метод show() можно вызвать и «обычным» образом, через ссылку на объект. Примером тому служат команды objC.show() и objB.show(). Ниже представлен результат выполнения программы:

Результат выполнения программы (из листинга 3.5)

```
Количество объектов: 0
Создан объект номер 1
Создан объект номер 2
Создан объект номер 3
Количество объектов: 3
Количество объектов: 3
Количество объектов: 3
```

Важно еще раз подчеркнуть, что статическое поле или метод — общие для всех объектов и существуют независимо от того, создан в программе хотя бы один объект или нет. С обычными (нестатическими) членами класса ситуация принципиально иная — каждый нестатический член существует только в привязке к конкретному объекту. Нет объекта — нет смысла говорить о нестатическом поле или методе (этого объекта).

Закрытые и открытые члены класса

Рассмотренный выше пример выявил одну потенциальную проблему: мы условились использовать статическое поле count (см. листинг 3.5) для подсчета количества созданных объектов, но при этом существует возможность изменять значение статического поля не только при вызове конструктора, но и непосредственным присваиванием полю значения. Поэтому нет гарантии, что значение поля не будет изменено «случайно».

Данная проблема имеет общий характер и связана с необходимостью ограничения доступа к некоторым членам класса. Такие члены класса называются *закрытыми* и описываются с ключевым словом private.

Закрытые члены класса доступны только в программном коде в теле класса, и к ним нет доступа вне тела класса.

НА ЗАМЕТКУ

Открытые члены класса могут описываться без указания идентификатора уровня доступа или с указанием идентификатора уровня доступа public. И в том, и в другом случае член класса будет открытым и доступным вне программного кода класса. Если член класса описан без идентификатора уровня доступа, то область его доступности ограничивается текущим *пакетом* (пакетом, в котором описан класс). Если член класса описан с ключевым словом public, то член класса доступен и в других пакетах.
Пакеты обсуждаются в книге, но немного позже.

Небольшая иллюстрация, являющаяся вариацией на тему предыдущего примера, представлена в листинге 3.6.

Листинг 3.6. Программный код проекта UsingPrivateMembersApplication

```
// Класс с закрытыми членами:
class MyClass{
  // Закрытое статическое поле:
  private static int count=0;
  // Закрытые нестатические поля:
  private int number;
  private String name;
  // Конструктор без аргументов:
  MyClass(String n){
    // Увеличение значения статического поля:
    count++;
    // Присваивание значений нестатическим полям:
    name=n;
    number=count;
    // Отображение сообщения:
    System.out.println("Создан объект с именем "+name);
  }
```

```
  // Метод для отображения сообщения:
  public void show(){
    System.out.println("Название объекта: "+name);
    System.out.println("Номер объекта: "+number);
    System.out.println("Количество объектов: "+count);
  }
  // Метод для присваивания значения закрытому
  // текстовому полю:
  public void set(String n){
    name=n;
  }
}
// Класс с главным методом программы:
class UsingPrivateMembersDemo{
  // Главный метод программы:
  public static void main(String[] args){
    // Создание объектов:
    MyClass objA=new MyClass("Alpha");
    MyClass objB=new MyClass("Bravo");
    MyClass objC=new MyClass("Charlie");
    // Вызов метода для каждого из объектов:
    objA.show();
    objB.show();
    objC.show();
    // Изменение поля второго объекта:
    objB.set("Второй Объект");
    // Отображение значений полей объекта:
    objB.show();
  }
}
```

Результат выполнения программы будет таким:

Результат выполнения программы (из листинга 3.6)

```
Создан объект с именем Alpha
Создан объект с именем Bravo
Создан объект с именем Charlie
Название объекта: Alpha
Номер объекта: 1
Количество объектов: 3
Название объекта: Bravo
Номер объекта: 2
Количество объектов: 3
Название объекта: Charlie
Номер объекта: 3
Количество объектов: 3
Название объекта: Второй Объект
Номер объекта: 2
Количество объектов: 3
```

В классе MyClass описано закрытое статическое поле count с начальным нулевым значением, два закрытых нестатических поля (целочисленное number и текстовое name). Также в классе описаны нестатические открытые методы:

- методом show() отображаются значения полей name и number объекта, из которого вызывается метод, а также значение статического поля count;
- метод set() предназначен для присваивания значения закрытому полю name.

Также в классе MyClass есть конструктор с одним текстовым аргументом. При вызове конструктора переданный ему аргумент присваивается значением полю name, значение статического поля count увеличивается на единицу, и это значение присваивается закрытому полю number. При создании объекта отображается сообщение с указанием значения поля name объекта.

Открытые методы show() и set() описаны с ключевым словом public. Мы их используем для того, чтобы получить доступ к закрытым полям. Общая ситуация такая.

- Статическое поле count закрытое и его значение изменяется (увеличивается на единицу) при создании объекта. Других способов (кроме создания объекта) изменить значение поля count нет.
- Закрытое поле number получает значение при создании объекта, после этого нет возможности изменить значение поля.
- Закрытое текстовое поле name получает значение на основе аргумента конструктора. Впоследствии изменить значение поля можно с помощью метода set().

В главном методе программы последовательно создается три объекта objA, objB и objC. Из каждого из них вызывается метод show(). Далее для объекта objB вызывается метод set() с аргументом "Второй Объект" (новое значение поля name объекта objB). Проверка с помощью команды objB.show() показывает, что значение поля name объекта objB действительно изменилось.

НА ЗАМЕТКУ

Важно отметить, что в главном методе программы обратиться к полям name, number и count через объекты objA, objB и objC (а в случае поля count еще и через класс MyClass) с использованием «точечного» синтаксиса не получится — поля закрытые. Мы использовали подход, который имеет достаточно общий характер: в классе описываются закрытые поля, доступ к которым осуществляется с помощью открытых методов. Таким образом, мы ограничиваем «спектр» возможных действий, выполняемых с полями.

Достаточно часто используются не только закрытые поля, но и закрытые методы. Обычно закрытые методы предназначены для выполнения определенных «промежуточных» или «вспомогательных» операций.

Резюме

Я задумался, а еда помогает мыслить.

Из к/ф «Гостья из будущего»

- Класс представляет собой некоторый шаблон, на основе которого создаются объекты. Описание класса начинается с ключевого слова class, после которого указывается имя класса и, в фигурных скобках, описание класса. В классе описываются поля и методы.

- Поле аналогично переменной, связанной с объектом, и описывается как переменная: указывается тип поля и его название.
- Метод представляет собой именованный блок команд, которые выполняются при вызове метода. При описании метода указывается тип возвращаемого методом результата, название метода, список аргументов (в круглых скобках) и собственно программный код метода (в фигурных скобках).
- Методы можно перегружать — описывать несколько методов с одинаковыми названиями. Такие методы должны отличаться списком аргументов (типом и/или количеством аргументов). Решение о том, какая версия метода используется при вызове, принимается исходя из контекста команды, в которой вызывается метод.
- Для создания объекта используют оператор `new`, после которого указывается имя класса и список аргументов, которые передаются конструктору. При выполнении такой команды создается объект, а результатом возвращается ссылка на созданный объект. Ссылка обычно присваивается значением объектной переменной, которую и отождествляют с объектом. Объектная переменная объявляется так же, как и обычная переменная, но в качестве идентификатора типа переменной указывается имя класса.
- Конструктор представляет собой метод, который автоматически вызывается при создании объекта. Имя конструктора совпадает с именем класса. Конструктор не возвращает результата, идентификатор типа результата для конструктора не указывается. У конструктора могут быть аргументы. В классе допускается описывать несколько версий конструктора. Если конструктор явно в классе не описан, используется конструктор по умолчанию, который не имеет аргументов и не выполняет никаких дополнительных действий при создании объекта. Если в классе описан хотя бы один конструктор, конструктор по умолчанию больше не доступен.
- При обращении к обычным (нестатическим) полям и методам объекта используется «точечный» синтаксис: указывается имя объекта, точка, и затем имя поля или имя метода с аргументами в круглых скобках (если аргументов нет, указываются пустые круглые скобки).
- Помимо обычных (нестатических) членов, в классе могут быть статические поля и методы. Статические члены класса описываются с ключевым словом `static`. Статические поля и методы являются

общими для всех объектов класса. Более того, статические члены класса и существуют независимо от наличия или отсутствия объектов класса. При обращении к статическим членам вместо имени объекта обычно указывается имя класса.

- По умолчанию описанные в классе поля и методы являются открытыми: к ним можно обращаться как в теле класса, так и вне класса. Открытые члены класса могут быть описаны с ключевым словом public. Если член класса описан с ключевым словом private, то он становится закрытым. Обращение к такому члену класса может быть выполнено только в программном коде в теле класса.

Глава 4
УПРАВЛЯЮЩИЕ ИНСТРУКЦИИ

Это великая победа дедуктивного метода.

Из к/ф «Гостья из будущего»

Далее речь пойдет об *управляющих инструкциях*. Под управляющими инструкциями мы подразумеваем условный оператор, оператор выбора и операторы цикла — синтаксические конструкции, которые позволяют создавать в программе точки ветвления, многократно выполнять определенные блоки команд и решать много других полезных задач.

Условный оператор

Ну, борода многогрешная, ежели за тобой что худое проведаю...

Из к/ф «Иван Васильевич меняет профессию»

Условный оператор позволяет выполнять разные блоки команд в зависимости от истинности или ложности определенного условия (под условием подразумевается любое выражение, имеющее значение логического типа). Общая схема использования условного оператора такая: проверяется условие — то есть вычисляется его значение, которое может быть true (истина) или false (ложь). Если значение условия равно true, выполняется определенный блок команд. Если значение условия равно false, выполняется другой блок команд.

Синтаксис условного оператора

В Java используется условный оператор if. Описание оператора начинается с ключевого слова if, после которого в круглых скобках указывается условие. После if-инструкции в фигурных скобках указывается блок команд, которые выполняются при истинности условия.

На случай, если условие ложно, предусмотрен другой блок команд, которые в фигурных скобках указываются после ключевого слова else. Таким образом, условный оператор if описывается в соответствии со следующим шаблоном (жирным шрифтом выделены ключевые элементы шаблона):

```
if(условие){
  // Команды выполняются при истинном условии
}
else{
  // Команды выполняются при ложном условии
}
```

Структурная схема на рис. 4.1 иллюстрирует общие принципы выполнения условного оператора.

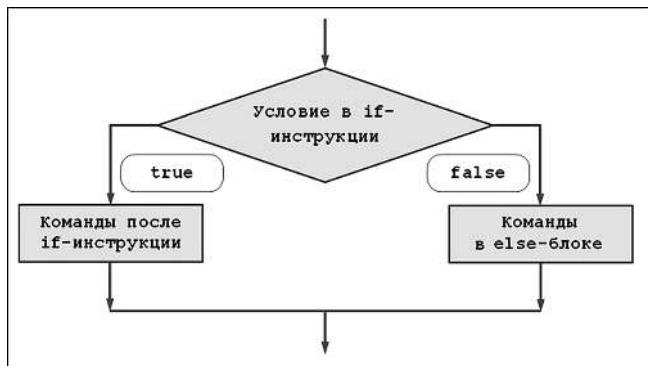


Рис. 4.1. Общая схема выполнения условного оператора

Существует также упрощенная форма условного оператора, в которой отсутствует else-блок. Шаблон описания условного оператора в упрощенной форме имеет такой вид (жирным шрифтом выделены основные элементы шаблона):

```
if(условие){
  // Команды выполняются при истинном условии
}
```

Схема выполнения условного оператора в упрощенной форме проиллюстрирована на рис. 4.2.

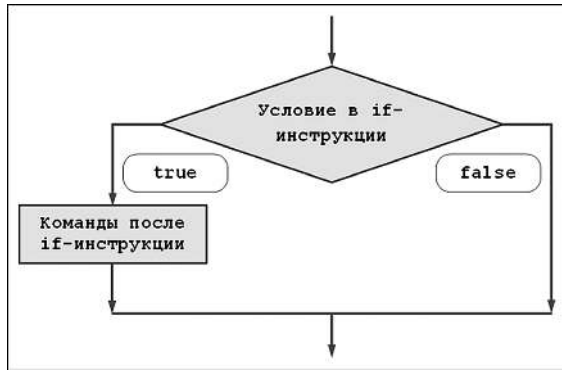


Рис. 4.2. Схема выполнения условного оператора в упрощенной форме

При использовании условного оператора в упрощенной форме проверяется условие, и при его истинности — выполняются команды после if-инструкции. Если условие ложно, то ничего не происходит, и начинает выполняться команда после условного оператора.

НА ЗАМЕТКУ

Если блок команд после if-инструкции или в else-блоке состоит из одной команды, то фигурные скобки можно не использовать. Тем не менее хороший стиль программирования подразумевает использование фигурных скобок в любых ситуациях. Во-первых, это помогает структурировать программный код. Во-вторых, позволяет избежать ненужных ошибок: если в упрощенной форме условного оператора не указать фигурные скобки для блока команд после if-инструкции, то из всех команд блока к условному оператору будет относиться только первая команда. При этом формально (с точки зрения синтаксиса) ошибки не будет.

Далее рассмотрим небольшие примеры использования условного оператора.

Использование условного оператора

Совсем небольшой пример, в котором используется условный оператор, представлен в листинге 4.1. В процессе выполнения программы появляется диалоговое окно с полем ввода, в которое следует ввести целое число. Число, с помощью условного оператора, проверяется на четность/нечетность, после чего появляется соответствующее сообщение.

НА ЗАМЕТКУ

Число является *четным*, если оно делится на два без остатка. Если число не делится на два без остатка, оно называется *нечетным*.

Теперь рассмотрим программный код примера.

Листинг 4.1. Программный код проекта UsingIfApplication

```
// Импорт классов:
import javax.swing.*;
// Класс с главным методом программы:
class UsingIfDemo{
  // Главный метод программы:
  public static void main(String[] args){
    // Текстовые переменные:
    String input,txt,title;
    // Целочисленная переменная:
    int number;
    // Объектная переменная для записи ссылки
    // на объект пиктограммы:
    ImageIcon img;
    // Отображение окна с полем ввода:
    input=JOptionPane.showInputDialog(null,
      "Введите целое число", // Надпись над полем ввода
      "Проверка числа",     // Заголовок окна
      JOptionPane.QUESTION_MESSAGE  // Тип пиктограммы
    );
    // Проверка, выполнен ли ввод:
    if(input==null){ // Если ввод не выполнен
      // Отображение диалогового окна:
      JOptionPane.showMessageDialog(null,
        "Вы не ввели число!",      // Сообщение
        "Ошибка ввода",           // Заголовок окна
        JOptionPane.ERROR_MESSAGE  // Тип окна
```

```
      );
      // Завершение выполнения программы:
      System.exit(0);
    }
    // Преобразование текста в число:
    number=Integer.parseInt(input);
    // Проверка числа на четность/нечетность:
    if(number%2==0){ // Если число четное
      // Создание объекта пиктограммы:
      img=new ImageIcon("d:/books/pictures/even.png");
      // Текст сообщения:
      txt="Число "+number+" — четное!";
      // Заголовок окна:
      title="Четное число";
    }
    else{ // Если число нечетное
      // Создание объекта пиктограммы:
      img=new ImageIcon("d:/books/pictures/odd.png");
      // Текст сообщения:
      txt="Число "+number+" — нечетное!";
      // Заголовок окна:
      title="Нечетное число";
    }
    // Отображение диалогового окна:
    JOptionPane.showMessageDialog(null,
      txt,                  // Текст сообщения
      title,                // Заголовок окна
      JOptionPane.PLAIN_MESSAGE, // Тип сообщения
      img                   // Пиктограмма
    );
  }
}
```

Перед анализом программного кода рассмотрим результат выполнения программы. Здесь возможны два сценария. Но в любом случае при запуске программы на выполнение сначала появляется окно с полем ввода, как показано на рис. 4.3.

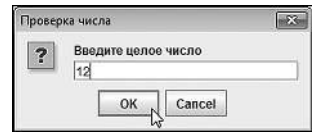


Рис. 4.3. В поле ввода указано четное число

Если мы вводим в поле четное число (см. рис. 4.3), то после щелчка на кнопке **ОК** появляется новое диалоговое окно с пиктограммой пользовательского типа (изображение пиктограммы определяется пользователем), в котором содержится сообщение о том, что число четное. Пример такой ситуации показан на рис. 4.4.

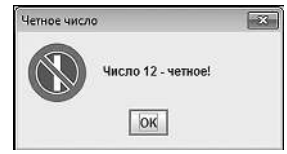


Рис. 4.4. Диалоговое окно, отображаемое при вводе четного числа

На рис. 4.5 показано диалоговое окно с полем ввода, в котором указывается нечетное целое число.

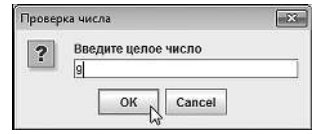


Рис. 4.5. В поле ввода указано нечетное число

Вид окна, которое отображается при вводе нечетного числа, показано на рис. 4.6.

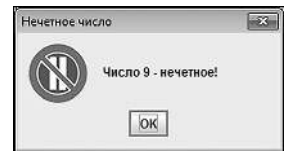


Рис. 4.6. Диалоговое окно, отображаемое при вводе нечетного числа

В данном случае используется иная пользовательская пиктограмма, включая иное название окна. Наконец, если в первом диалоговом окне (см. рис. 4.3 или рис. 4.5) пользователь вместо ввода числа щелкает кнопку **Cancel** или системную пиктограмму закрытия окна в правом верхнем углу, появляется диалоговое окно, как на рис. 4.7.

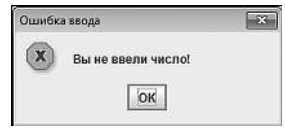


Рис. 4.7. Диалоговое окно появляется, если пользователь не вводит число

После этого выполнение программы завершается.

НА ЗАМЕТКУ

Приведенное на рис. 4.7 диалоговое окно появляется, если пользователь в окне с полем ввода щелкает кнопку **Cancel** или системную пиктограмму. Если пользователь вводит в поле ввода некорректное значение и щелкает кнопку **OK**, возникает ошибка, связанная с невозможностью преобразовать введенное текстовое значение в целое число.

Теперь проанализируем программный код, благодаря которому реализуются перечисленные выше возможности. Сразу отметим, что в программном коде, помимо условного оператора, встречаются и другие «новшества».

Программный код начинается инструкцией `import javax.swing.*`. От знакомой нам инструкции импортирования класса `JOptionPane` эта инструкция отличается наличием звездочки * в том месте, где раньше мы указывали имя импортируемого класса. Звездочка означает, что импортируются все классы из соответствующего пакета (в данном случае из пакета `swing`, являющегося подпакетом пакета `javax`).

Программа состоит из описания одного класса `UsingIfDemo` с главным методом программы, и все самое интересное происходит именно там. В теле метода `main()` объявляются три текстовые переменные (объектные переменные класса `String`) `input` (для записи результата считывания значения из поля ввода), `txt` (для записи текста, определяющего сообщение в диалоговом окне) и `title` (для записи названия диалогового окна).

Также объявляется переменная number типа int, в которую мы планируем записывать числовое значение, вводимое пользователем в поле ввода. Также нам понадобится объектная переменная, в которую мы запишем ссылку на объект *пиктограммы*.

НА ЗАМЕТКУ

Мы используем при отображении диалоговых окон пользовательские пиктограммы, которые создаются на основе графических изображений. Пиктограмма реализуется в виде объекта класса ImageIcon, который импортируется наравне с классом JOptionPane из подпакета swing пакета javax. Для работы с объектом пиктограммы нужна объектная переменная класса ImageIcon.

Объект пиктограммы создается на основе класса ImageIcon, поэтому в программе объявляется объектная переменная img класса ImageIcon.

После объявления переменных, командой input=JOptionPane.showInputDialog(null,"Введите целое число","Проверка числа",JOptionPane.QUESTION_MESSAGE) отображается диалоговое окно с полем ввода, в которое пользователь должен ввести целое число и подтвердить свой ввод щелчком на кнопке **ОК**. Результат в текстовом виде записывается в переменную input.

ДЕТАЛИ

Напомним, что аргументы методу showInputDialog() могут передаваться по-разному. В данном случае первый аргумент null представляет собой пустую ссылку на родительское окно (проще говоря, такого окна нет). Второй аргумент "Введите целое число" определяет надпись над полем ввода. Третий аргумент "Проверка числа" задает название окна. А четвертый аргумент JOptionPane.QUESTION_MESSAGE определяет тип пиктограммы, отображаемой в области окна (пиктограмма со знаком вопроса).

В программном коде в листинге 4.1 для удобства восприятия аргументы методов showInputDialog() и showMessageDialog() отображаются в отдельных строках и поясняются комментариями. Так можно поступать — главное, чтобы при этом корректно были указаны запятые, разделяющие аргументы, и закрывающая круглая скобка, а количество пробелов между инструкциями не принципиально.

Результатом метод showInputDialog() возвращает текстовое значение, даже если в поле ввода указано число. Поэтому переменная input, в которую

записывается результат, объявлена как текстовая. Но если быть более точным, то переменная input является объектной переменой класса String. Если пользователь щелкает в диалоговом окне с полем ввода кнопку **ОК**, то на основе значения в поле ввода создается текстовый объект, и ссылка на него возвращается результатом метода showInputDialog() (и, следовательно, записывается в переменную input). Если же пользователь щелкает кнопку **Cancel**, то результатом метода showInputDialog() возвращается пустая ссылка null. Это же происходит, если пользователь закрывает окно щелчком по системной пиктограмме. Итог следующий: если пользователь щелкает кнопку **Cancel** или системную пиктограмму, значение переменной input будет равно null. Именно такая ситуация проверяется в условном операторе (использована упрощенная форма). Проверяется условие input==null. Если оно истинно, то командой JOptionPane.showMessageDialog(null,"Вы не ввели число!","Ошибка ввода",JOptionPane.ERROR_MESSAGE) отображается диалоговое окно с пиктограммой ошибки и сообщением о том, что пользователь не ввел число. Затем командой System.exit(0) завершается выполнение программы.

ДЕТАЛИ

Из класса System вызывается статический метод exit(), в результате чего выполнение программы завершается. Нулевое значение, которое передается методу exit() при вызове, говорит, что программа завершила выполнение в нормальном режиме, без ошибок.

Если условие input==null ложно, то до выполнения команды завершения программы дело не доходит, а вместо этого командой number=Integer.parseInt(input) на основе считанного текстового значения определяется целое число (предполагается, что в переменную input записано текстовое представление для целого числа). Затем с помощью еще одного условного оператора число тестируется на четность/нечетность. В частности, в условном операторе проверяется условие number%2==0, заключающееся в том, что остаток от деления значения переменной number на число два равно нулю (то есть значение переменной number делится на два без остатка, то есть число четное). В таком случае выполняются следующие команды.

- Командой img=new ImageIcon("d:/books/pictures/even.png") на основе изображения, записанного в файл even.png, расположенный в каталоге d:\books\pictures, создается объект класса ImageIcon пиктограммы, и ссылка на объект записывается в переменную img.

- Командой txt="Число "+number+" — четное!" создается текст для сообщения, отображаемого в диалоговом окне.
- Командой title="Четное число" создается текстовое значение для заголовка окна.

Нечто похожее происходит и в случае, если условие number%2==0 ложно (в переменную number записано нечетное число), но только значения используются другие.

- Объект для пиктограммы создается командой img=new ImageIcon("d:/books/pictures/odd.png") на основе изображения из файла odd.png из каталога d:\books\pictures.
- Текст для отображения в виде сообщения в диалоговом окне определяется командой txt="Число "+number+" — нечетное!".
- Заголовок для окна определяется командой title="Нечетное число".

ДЕТАЛИ

Предварительно в каталог d:\books\pictures помещаются два файла (even.png и odd.png) с изображениями, которые мы используем в качестве пиктограмм. Размеры изображений — 60 пикселов в ширину и 60 пикселов в высоту (параметры изображения важны, поскольку в данном случае пиктограмма при отображении не масштабируется). Мы используем формат png, поскольку в таком формате можно сделать прозрачным фон изображения (поэтому в диалоговом окне отображается только «круглый» рисунок, в то время как сама пиктограмма квадратная). Но «напрямую» изображения использовать нельзя. На основе изображений создается объект класса ImageIcon. Аргументом конструктору класса передается текстовая строка с полным путем к рисунку. Путь может быть абсолютным или относительным (определяется по отношению к каталогу, в котором находятся файлы программы).

После того как все необходимые параметры определены, командой JOptionPane.showMessageDialog(null,txt,title,JOptionPane.PLAIN_MESSAGE,img) отображается диалоговое окно с соответствующим сообщением, пиктограммой и заголовком. Здесь мы сталкиваемся с новым способом передачи аргументов методу showMessageDialog(). Пустая ссылка null (первый аргумент метода) означает отсутствие родительского окна. Переменная txt (второй аргумент метода) определяет сообщение, отображаемое в окне. Переменная title (третий аргумент метода) задает заголовок окна. Статическая

константа PLAIN_MESSAGE из класса JOptionPane, переданная четвертым аргументом метода, определяет тип сообщения. Формально данная константа означает, что сообщение должно быть без пиктограммы. Но поскольку методу передан и пятый аргумент, которым является объектная переменная img, ссылающаяся на объект пиктограммы, то предыдущий аргумент «игнорируется» и в окне отображается пиктограмма, определяемая переменной img.

НА ЗАМЕТКУ

Поскольку при наличии пятого аргумента четвертый аргумент метода showMessageDialog() игнорируется, то с таким же самым успехом вместо константы PLAIN_MESSAGE мы могли бы использовать другое значение.

Вложенные условные операторы

Один условный оператор может размещаться внутри другого условного оператора. Синтаксическая конструкция, которая получается в результате, обычно называется *вложенными условными операторами*. В общем случае существует много разных способов взаимного размещения условных операторов, но на практике важным является случай, когда необходимо последовательно проверять несколько условий. Ниже представлен формальный шаблон синтаксической конструкции, в которой else-блок условного оператора сам состоит из условного оператора, в котором else-блок состоит из условного оператора, и так далее (жирным шрифтом выделены ключевые элементы шаблона):

```
if(условие){
  // Команды
}
else if(условие){
  // Команды
}
else if(условие){
  // Команды
}
// Прочие блоки
else if(условие){
```

```
  // Команды
}
else{
  // Команды
}
```

Речь идет о самом обычном выражении из условных операторов. Мы лишь использовали немного специфический синтаксис: поскольку все «внутренние» else-блоки состоят лишь из одного оператора (условного), то мы не задействовали фигурные скобки и ключевое слово if для внутреннего условного оператора разместили в той же строке, что и ключевое слово else для внешнего условного оператора. Как бы там ни было, при использовании вложенных условных операторов логика выполнения программы всецело определяется правилами выполнения отдельного условного оператора. На рис. 4.8 показана общая схема выполнения блока из вложенных условных операторов, организованных в соответствии с приведенным выше шаблоном.

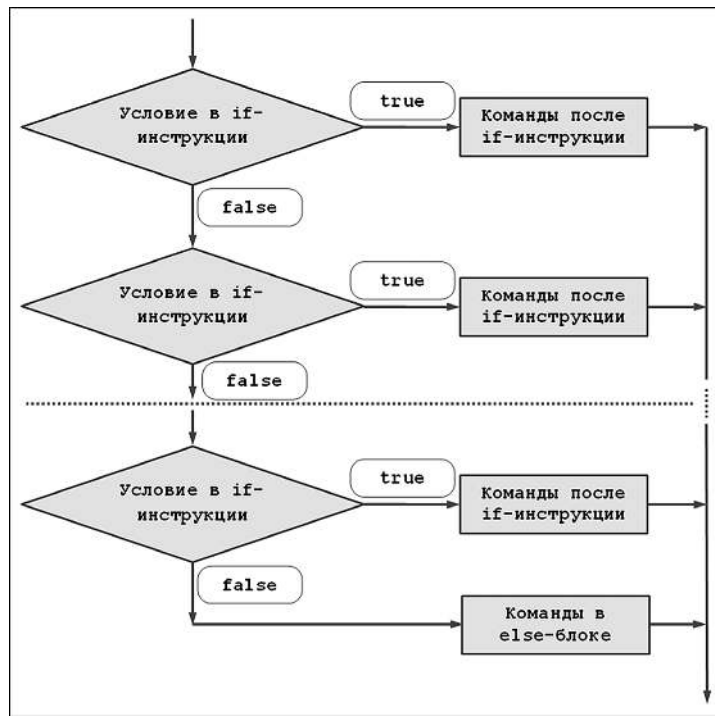


Рис. 4.8. Схема выполнения вложенных условных операторов

Принцип выполнения блока из вложенных условных операторов простой.

- Проверяется условие во внешнем условном операторе, и если оно истинно — выполняются команды после соответствующей if-инструкции.
- При ложном условии выполняется else-блок, состоящий из условного оператора. В этом операторе проверяется условие, и если оно истинно, начинается выполнение команд после if-инструкции.
- Если условие во внутреннем условном операторе ложно, то в игру вступает else-блок внутреннего условного оператора. Этот блок состоит из еще одного условного оператора, в котором проверяется условие, ну и так далее.
- Последний else-блок выполняется только в случае, если все проверяемые до этого условия оказались ложными.

Небольшая иллюстрация к использованию вложенных условных операторов представлена в листинге 4.2. В этой программе пользователю предлагается в поле ввода ввести название животного (*волк*, *лиса* или *медведь* — на выбор). После ввода названия животного появляется диалоговое окно с изображением зверя. В программе предусмотрена обработка ситуации, когда пользователь вводит некорректное название для животного или отменяет ввод щелчком на кнопке **Cancel**.

НА ЗАМЕТКУ

Для нормального функционирования программы в каталог d:\books\pictures\ помещаются четыре графических файла (wolf.jpg, fox.jpg, bear.jpg и raccoon.jpg) с изображениями животных (соответственно *волк*, *лиса*, *медведь* и *енот*). Каждое изображение в ширину имеет размер в 150 пикселов и в высоту имеет размер в 100 пикселов.

Рассмотрим представленный далее программный код.

Листинг 4.2. Программный код проекта UsingMultiIfApplication

```
// Импорт классов:
import javax.swing.*;
// Класс с главным методом программы:
class UsingMultiIfDemo{
```

```
// Главный метод программы:
public static void main(String[] args){
  // Объектная переменная для пиктограммы:
  ImageIcon img;
  // Текстовая переменная для записи
  // названия животного:
  String animal;
  // Текстовая переменная с начальным значением,
  // определяющим путь к файлу с изображением:
  String file="d:/books/pictures/";
  // Текстовые переменные с названиями животных:
  String wolf="Волк";
  String fox="Лиса";
  String bear="Медведь";
  String raccoon="Енот";
  // Считывание названия животного:
  animal=JOptionPane.showInputDialog(null,
    // Текст над полем ввода:
    wolf+", "+fox+" или "+bear+"?",
    // Название окна:
    "Укажите название животного",
    // Тип пиктограммы:
    JOptionPane.QUESTION_MESSAGE
    );
  // Вложенные условные операторы:
  if(animal==null){ // Если пользователь отменил ввод
    System.exit(0); // Завершение выполнения программы
  }
  // Если животное "Волк":
  else if(animal.equalsIgnoreCase(wolf)){
    file+="wolf.jpg";
    animal=wolf;
```

```
    }
    // Если животное "Лиса":
    else if(animal.equalsIgnoreCase(fox)){
      file+="fox.jpg";
      animal=fox;
    }
    // Если животное "Медведь":
    else if(animal.equalsIgnoreCase(bear)){
      file+="bear.jpg";
      animal=bear;
    }
    // Неизвестное животное:
    else{
      file+="raccoon.jpg";
      animal=raccoon;
    }
    // Создание объекта для пиктограммы:
    img=new ImageIcon(file);
    // Отображение диалогового окна:
    JOptionPane.showMessageDialog(null,
      img,   // В окне отображается изображение
      animal, // Название окна
      // Тип диалогового окна:
      JOptionPane.PLAIN_MESSAGE
    );
  }
}
```

Проанализируем программный код. В программе (в главном методе) объявляется ряд переменных. Объектная переменная img класса ImageIcon предназначена для записи ссылки на объект пиктограммы (с изображением зверушки), которая будет отображаться в диалоговом окне. Название животного будет записываться (после считывания из поля ввода)

в текстовую переменную animal. Текстовой переменной file в качестве начального значения присваивается литерал "d:/books/pictures/", содержащий полный путь к каталогу с графическими файлами, содержащими изображения, на основе которых будут создаваться пиктограммы. Еще четыре текстовые переменные (wolf, fox, bear и raccoon) содержат названия животных. Мы будем использовать эти переменные для определения названия диалогового окна. Также со значениями этих переменных будет сравниваться значение, вводимое пользователем в текстовое поле диалогового окна.

Значение переменной animal определяется как результат вызова метода showInputDialog(). Аргументами методу при вызове передаются:

- первый аргумент — пустая ссылка null на родительское окно (пустая ссылка означает, что такого окна нет);
- второй аргумент — текстовое значение wolf+", "+fox+" или "+bear+"?", представляющее собой результат объединения текстовых значений переменных wolf, fox и bear с разными разделителями;
- третий аргумент — текстовый литерал "Укажите название животного", определяющий название диалогового окна;
- последний, четвертый аргумент — статическая константа QUESTION_MESSAGE класса JOptionPane, которая задает тип пиктограммы, отображаемой в окне (стандартная пиктограмма со знаком вопроса).

Далее в программном коде следует блок из вложенных условных операторов. Первым условием проверяется animal==null. Истинность данного условия означает, что пользователь в окне с полем ввода щелкнул кнопку **Cancel** или закрыл окно с помощью системной пиктограммы (пиктограмма с крестиком в правом верхнем углу окна). В таком случае (если условие animal==null) выполняется команда System.exit(0), которой завершается выполнение программы.

НА ЗАМЕТКУ

Таким образом, если пользователь в окне с полем ввода щелкает кнопку **Cancel** или закрывает окно щелчком на системной пиктограмме, то программа завершает выполнение и второе диалоговое окно не появляется.

Если условие animal==null ложно, начинается проверка значения переменной animal на предмет совпадения со значением одной из текстовых

переменных wolf, fox или bear. Для сравнения текстовых значений используем метод equalsIgnoreCase(), который вызывается из одного текстового объекта, а аргументом методу передается другой текстовый объект. Результатом метод возвращает значение true, если в обоих текстовых объектах записан один и тот же текст, причем регистр букв в данном случае в расчет не принимается. Так, результатом выражения animal.equalsIgnoreCase(wolf) является значение true, если переменные animal и wolf ссылаются на текстовые объекты с одинаковыми (без учета состояния регистра) текстовыми значениями. Поскольку переменная wolf ссылается на объект с текстовым значением "Волк", то истинным выражение animal.equalsIgnoreCase(wolf) будет, если переменная animal ссылается на объект с таким же текстом. То, что регистр букв в расчет не принимается, означает, что, например, текстовые значения "Волк", "волк" и "ВОЛК" интерпретируются как совпадающие (поэтому в поле ввода название животного можем вводить как строчными, так и прописными буквами).

ДЕТАЛИ

Для сравнения текстовых значений логические операторы == (равно) и != (не равно) обычно не используют. Дело в том, что значением объектной переменной класса String является не текст, а ссылка на объект, содержащий текст. Поэтому если сравнивать значения двух объектных переменных класса String с помощью оператора ==, то значение true получим в том случае, если переменные ссылаются на *один и тот же* объект. Объяснение простое: сравниваются значения переменных, а значениями являются ссылки на объекты — то есть фактически адреса объектов. Если адреса одинаковые, то речь, очевидно, идет об одном и том же объекте. Но нас интересует текст, записанный в текстовых объектах. То есть объекты могут быть физически разными, но текст в них может быть записан одинаковый. Именно это обстоятельство проверяется. Поэтому нам понадобился метод equalsIgnoreCase(). Такой метод есть у каждого объекта класса String. Кроме метода equalsIgnoreCase() имеется еще метод equals(), который также используют для сравнения текстовых значений. В отличие от метода equalsIgnoreCase(), которым сравнение выполняется без учета состояния регистра букв в тексте, при сравнении текстов методом equals() регистр букв принимается в расчет, и в таком случае, например, текстовые значения "Волк" и "волк" интерпретируются как различные.

Итак, если при проверке условия animal.equalsIgnoreCase(wolf) оказывается, что оно истинно, командой file+="wolf.jpg" к значению текстовой

переменной file дописывается название файла с изображением, а командой animal=wolf переменной animal присваивается значение переменной wolf. Последняя операция выполняется для того, чтобы во втором окне название окна, совпадающее с названием животного, отображалось так, как записано значение переменной wolf, а не так, как его ввел пользователь в поле ввода (возможные различия сводятся лишь к состоянию регистра букв).

ДЕТАЛИ

После выполнения команды animal=wolf переменная animal ссылается на тот же самый текстовый объект, что и переменная wolf. Причина в том, что и animal, и wolf являются объектными переменными, поэтому ссылаются на объекты, а значения переменных — адреса объектов. Когда выполняется команда animal=wolf, переменной animal присваивается значение переменной wolf. Значением переменной wolf является адрес объекта (ссылка на объект). В итоге переменная animal будет содержать значением ту же ссылку, что и переменная wolf. Следовательно, переменная animal будет ссылаться на тот же объект, что и переменная wolf.

Если условие animal.equalsIgnoreCase(wolf) ложно, проверяется условие animal.equalsIgnoreCase(fox). При истинности данного условия выполняются команды, аналогичные предыдущему случаю, но с поправкой на название файла и переменную с названием животного.

При ложности условия animal.equalsIgnoreCase(fox) проверяется условие animal.equalsIgnoreCase(bear). Наконец, если все из перечисленных условий оказались ложными, в последнем else-блоке выполняются команды file+="raccoon.jpg" и animal=raccoon. Таким образом, если пользователь вводит в поле ввода любое значение, кроме названия *волка*, *лисы* или *медведя*, появится окно с изображением *енота*.

НА ЗАМЕТКУ

Фактически любая нестандартная ситуация интерпретируется в пользу енота. Другими словами, енот — зверь по умолчанию.

После выполнения блока команд из условных операторов командой img=new ImageIcon(file) создается объект для пиктограммы, а с помощью статического метода showMessageDialog() из класса JOptionPane изображение

отображается в диалоговом окне. Аргументы методу передаются такие.

- Пустая ссылка null, переданная первым аргументом, означает отсутствие родительского окна.
- Вторым аргументом передана переменная img, ссылающаяся на объект для пиктограммы. Ранее вторым аргументом мы передавали текст, который следует отобразить в диалоговом окне. Здесь же мы передаем вторым аргументом объект пиктограммы, которая отображается в диалоговом окне. Другими словами, теперь вместо текста мы используем изображение.
- Третьим аргументом передается переменная animal, которая ссылается на один из литералов, определяемых переменными wolf, fox, bear и raccoon. Соответствующий текст будет отображаться в качестве названия окна.
- Четвертым аргументом указана инструкция JOptionPane.PLAIN_MESSAGE, означающая отсутствие пиктограммы в левой части в области диалогового окна.

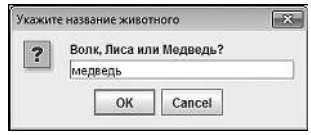
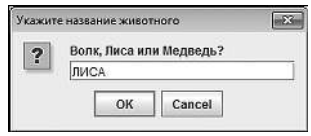
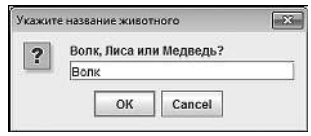
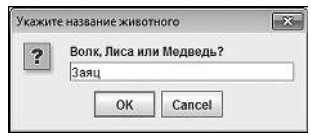
ДЕТАЛИ

Диалоговое окно, отображаемое с помощью метода showMessageDialog() в левой части может содержать пиктограмму: например, информационную, со знаком вопроса, предупреждающую, пиктограмму ошибки или определенную пользователем. Если речь идет о стандартных пиктограммах, то для определения типа пиктограммы используется одна из статических констант INFORMATION_MESSAGE, QUESTION_MESSAGE, WARNING_MESSAGE или ERROR_MESSAGE класса JOptionPane. Константа PLAIN_MESSAGE означает, что такая пиктограмма не используется. Изображение, отображаемое в данном примере в диалоговом окне, размещается в том месте, где обычно находится текстовое сообщение.

Результат выполнения программы иллюстрирует табл. 4.1. Там показано, как может заполняться поле ввода в первом окне, и какой вид при этом будет иметь второе окно.

Напомним, что если в первом диалоговом окне пользователь не щелкает кнопку **ОК**, то программа просто завершает выполнение без отображения второго окна.

Табл. 4.1. Возможные результаты выполнения программы

Первое окно	Второе окно	Пояснение
		В первом окне в поле ввода вводится слово медведь. После подтверждения ввода (щелчок на кнопке **OK**) появляется диалоговое окно с изображением *медведя*
		В поле ввода первого окна вводится слово ЛИСА. После щелчка на кнопке **OK** появляется диалоговое окно с изображением *лисы*
		В поле ввода в первом диалоговом окне вводится слово Волк. Во втором диалоговом окне содержится изображение *волка*
		В поле ввода первого окна вводится слово Заяц (хотя конкретное значение не принципиально — важно, что это не название *волка*, *лисы* или *медведя*). Во втором окне появляется изображение *енота*

Операторы цикла

Передайте Зинаиде Михайловне, что Розалия Францевна говорила Анне Ивановне: «Капитолина Никифоровна дубленку предлагает».

Из к/ф «Иван Васильевич меняет профессию»

Операторы цикла позволяют многократно выполнять определенные блоки команд. В языке Java существует несколько операторов цикла, которые мы и обсудим далее.

Оператор цикла while

Оператор цикла while имеет, пожалуй, самый простой синтаксис среди операторов цикла. Начинается описание оператора цикла с ключевого

слова while, после которого в круглых скобках указывается условие, представляющее собой выражение со значением логического типа. После условия в фигурных скобках размещается блок команд — тело оператора цикла. Общий шаблон описания оператора цикла while представлен ниже (основные элементы шаблона выделены жирным шрифтом):

```
while(условие){
  // Команды
}
```

При выполнении оператора цикла проверяется условие. Если условие истинно (значение true), то выполняются команды в теле оператора цикла. Затем снова проверяется условие. Если оно истинно, выполняются команды, после чего проверяется условие, и так далее. Если при очередной «плановой» проверке условия окажется, что оно ложно (значение false), выполнение оператора цикла прекращается. На рис. 4.9 процесс выполнения оператора цикла while иллюстрируется с помощью схемы.

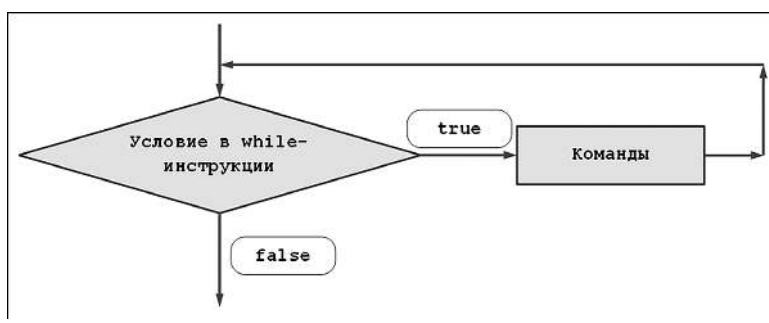


Рис. 4.9. Схема выполнения оператора цикла while

Пример, в котором используется оператор цикла while, представлен в листинге 4.3. Вниманию читателя предлагается простая программа, предназначенная для определения целого положительного числа по его бинарному коду.

НА ЗАМЕТКУ

Если число имеет бинарный код $\overline{a_n a_{n-1} \ldots a_2 a_1 a_0}$ (параметры a_k принимают значения 0 или 1 при всех $k = 0, 1, 2, \ldots, n$), то значение этого числа вычисляется как $a_0 \cdot 2^0 + a_1 \cdot 2^1 + a_2 \cdot 2^2 + \ldots + a_{n-1} \cdot 2^{n-1} + a_n \cdot 2^n$.

Бинарный код числа вводится пользователем в поле ввода диалогового окна, которое появляется после запуска программы на выполнение. После ввода кода появляется второе диалоговое окно с сообщением, в котором указан исходный код и число, которому этот код соответствует. А теперь рассмотрим программу.

Листинг 4.3. Программный код проекта UsingWhileApplicaion

```
import javax.swing.JOptionPane;
// Класс с главным методом программы:
class UsingWhileDemo{
  // Статический метод для определения числа
  // (результат метода) на основе
  // бинарного кода (аргумент метода):
  static int decoder(String code){
    // Количество символов в тексте:
    int n=code.length();
    // Целочисленные переменные:
    int s=0,k=1,q=1,a;
    // Оператор цикла для определения числа:
    while(k<=n){
      // Проверяется символ в бинарном коде:
      if(code.charAt(n-k)=='1'){
        a=1;
      }
      else{
        a=0;
      }
      // Вычисление суммы, определяющей число:
      s+=q*a;
      // Степень двойки:
      q*=2;
      // Новый индекс для определения символа в коде:
      k++;
    }
```

```
    // Результат метода:
    return s;
  }
  // Главный метод программы:
  public static void main(String[] args){
    // Текстовая переменная для записи бинарного кода:
    String input;
    // Название для диалоговых окон:
    String title="Расшифровка бинарного кода";
    // Отображение диалогового окна с полем для ввода
    // бинарного кода:
    input=JOptionPane.showInputDialog(null,
      "Введите бинарный код", // Надпись над полем ввода
      title,                  // Название окна
      JOptionPane.QUESTION_MESSAGE // Тип пиктограммы
    );
    // Если пользователь не ввел код:
    if(input==null){
      // Завершение выполнения программы:
      System.exit(0);
    }
    // Определение числа на основе бинарного кода:
    int num=decoder(input);
    // Текст для отображения в диалоговом окне:
    String text="Вы ввели бинарный код:  "+input;
    text+="\nКод соответствует числу:  "+num;
    // Отображение диалогового окна:
    JOptionPane.showMessageDialog(null,
      text,  // Текст сообщения
      title, // Название окна
      JOptionPane.INFORMATION_MESSAGE // Тип пиктограммы
    );
  }
}
```

В программе описывается класс UsingWhileDemo, в котором, кроме главного метода программы, описан статический метод decoder(). Аргументом методу передается текстовая строка с бинарным кодом, а результатом метод возвращает целое число, соответствующее данному бинарному коду. Фактически все основные вычисления выполняются в методе decoder(). Проанализируем более детально программный код данного метода.

Аргумент метода decoder() обозначен как code. Это объектная переменная класса String и, как мы предполагаем, она содержит ссылку на текстовый объект, содержащий бинарный код числа.

В теле метода объявляется целочисленная переменная n, значением которой присваивается выражение code.length(). Здесь используется метод length(), возвращающий результатом количество символов в том текстовом объекте (объект класса String), из которого вызывается метод. В результате в переменную n будет записано количество символов в текстовом значении (бинарный код), на которое ссылается переменная code. Также в методе объявляются и другие целочисленные переменные:

- переменная s с начальным нулевым значением используется для записи суммы, через которую определяется искомое число;
- переменная k с начальным единичным значением используется при переборе символов в бинарном коде;
- переменная q с начальным единичным значением нужна для записи в нее значений степеней двойки;
- в переменную a записывается значение (число 1 или 0) параметра в позиционном представлении числа в двоичной системе.

Основные вычисления выполняются в операторе цикла while. Проверяемым условием в операторе цикла указано выражение k<=n. Пока условие истинно, оператор цикла выполняется.

В самом начале в теле оператора цикла с помощью условного оператора if вычисляется значение для переменной a. В условном операторе проверяется условие code.charAt(n-k)=='1'. В данном выражении из текстового объекта code вызывается метод charAt(). Результатом метод возвращает символ (значение типа char), индекс которого передан аргументом методу. Индексация символов в текстовой строке начинается с нуля. Поэтому самый первый элемент имеет нулевой индекс. Если в текстовой строке n символов, то индекс последнего символа равен n-1. У предпоследнего символа в текстовой строке индекс равен n-2, и так далее. Последней командой

в теле оператора цикла указана инструкция k++. Поэтому за каждый цикл значение переменной k увеличивается на единицу. Поскольку аргументом методу charAt() передается выражение n-k, и начальное значение переменной k равно 1, то при выполнении оператора цикла в текстовой строке code будут последовательно перебираться символы (с конца текстовой строки в ее начало, по одному символу за цикл). Далее, условие code.charAt(n-k)=='1' истинно, если считанный в текстовой строке символ равен '1'. В таком случае переменной a присваивается значение 1. Если условие code.charAt(n-k)=='1' ложно, переменной a присваивается значение 0.

НА ЗАМЕТКУ

Таким образом, аргументом методу decoder() передается текст, который интерпретируется как бинарный код целого неотрицательного числа. С сугубо математической точки зрения данный код должен состоять из нулей и единиц. Но на практике нет гарантии, что пользователь введет корректный код. Программный код метода decoder() организован так, что символ '1' в коде интерпретируется как единица, а все прочие символы интерпретируются как нуль. В этом смысле, например, код, реализованный текстом "1A201X" эквивалентен «текстовому» коду "100010".

Командой s+=q*a к текущему значению переменной s добавляется очередное слагаемое, после чего командой q*=2 значение переменной q увеличивается в два раза. Наконец, значение переменной k с помощью команды k++ увеличивается на единицу.

ДЕТАЛИ

Если задан бинарный код $\overline{a_n a_{n-1} \ldots a_2 a_1 a_0}$, в котором параметры a_k ($k = 0, 1, 2, \ldots, n$) могут принимать значения 0 и 1, то в десятичной системе соответствующее число вычисляется как сумма $\sum_{k=0}^{n} a_k 2^k$, которую можно формально записать в виде $\sum_{k=0}^{n} a_k q_k$ с параметрами $q_k = 2^k$. Значение данной суммы записывается в переменную s. За каждый цикл в переменную s добавляется очередное слагаемое вида $a_k q_k$. Значение параметра a_k для текущего цикла записывается в переменную a, а значение параметра q_k записывается в переменную q. Причем мы учли, что $q_{k+1} = 2qk$, то есть для следующего цикла значение переменной q должно быть в два раза больше. Следует также сделать поправку на границы изменения переменной k в программном коде метода decoder().

По окончании вычислений командой return s значение переменной s возвращается как результат метода decoder().

В главном методе программы объявляется объектная переменная input класса String. Текстовая переменная title определяет название диалоговых окон. Первое окно отображается с помощью метода showInputDialog() класса JOptionPane, а результат вызова метода записывается в переменную input.

Сначала в условном операторе проверяется условие input==null. Если переменная input не ссылается на текстовый объект, то командой System.exit(0) завершается выполнение программы. В противном случае в целочисленную переменную num записывается результат выражения decoder(input) — то есть фактически результат преобразования бинарного кода в десятичное число. Результат вычислений отображается с помощью статического метода showMessageDialog() класса JOptionPane.

ДЕТАЛИ

В главном методе main() вызывается метод decoder(). Метод decoder() описан как статический, поэтому его можно вызывать без создания объекта класса. Более того, поскольку метод decoder() описан в том же самом классе, что и метод main(), то при вызове в методе main() статического метода decoder() имя класса можно не указывать.

При запуске программы на выполнение появляется диалоговое окно, в поле ввода которого необходимо ввести бинарный код числа. На рис. 4.10 показано диалоговое окно, в котором в поле ввода указан бинарный код 1011010.

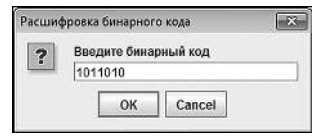


Рис. 4.10. В поле ввода указан бинарный код числа

После щелчка на кнопке **ОК** открывается новое диалоговое окно (показано на рис. 4.11), в котором сказано, что бинарному коду 1011010 соответствует число 90.

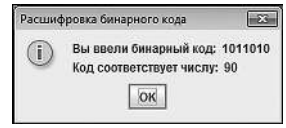


Рис. 4.11. Результат вычисления числа по бинарному коду

НА ЗАМЕТКУ

Для перевода бинарного кода 1011010 в десятичное число необходимо выполнить следующие вычисления: $1011010 = 0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^4 + 0 \cdot 2^5 + 1 \cdot 2^6 = 2 + 8 + 16 + 64 = 90$.

Если при отображении первого диалогового окна (см. рис. 4.10) пользователь щелкнет кнопку **Cancel** или закроет окно щелчком на системной пиктограмме с крестиком в правом верхнем углу диалогового окна, то программа завершает работу и второе окно со значением числа не отображается.

Оператор цикла do-while

Оператор цикла do-while похож на оператор while, но имеет несколько иной синтаксис. Описание оператора цикла do-while начинается с ключевого слова do, после которого в круглых скобках указываются команды, выполняемые за каждый цикл. После блока команд следует ключевое слово while, а в круглых скобках указывается условие, проверяемое каждый раз, когда выполнены команды в теле цикла. В конце всей этой конструкции ставится точка с запятой. Таким образом, шаблон описания оператора цикла do-while такой (жирным шрифтом выделены ключевые элементы шаблона):

```
do{
  // Команды
}while(условие);
```

Схема выполнения оператора цикла do-while проиллюстрирована на рис. 4.12.

Сначала выполняются команды в теле оператора цикла. Затем проверяется условие, и если оно истинно, снова выполняются команды. После выполнения команд проверяется условие. Если условие истинно, снова выполняются команды, проверяется условие, и так далее. Для завершения выполнения оператора цикла необходимо, чтобы при проверке условия оно оказалось ложным.

НА ЗАМЕТКУ

В операторе цикла while сначала проверяется условие, а затем (при истинном условии) выполняются команды в теле оператора цикла. Если в операторе while при первой проверке условия окажется,

что оно ложно, то команды в теле оператора выполняться не будут. В операторе цикла do-while сначала выполняются команды, и только после этого проверяется условие. Поэтому в операторе цикла do-while команды в теле оператора цикла выполняются, по крайней мере, один раз.

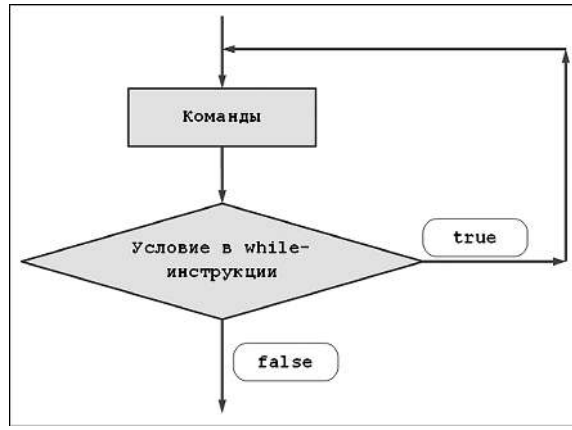


Рис. 4.12. Схема выполнения оператора цикла do-while

Пример использования оператора do-while приведен в листинге 4.4. В программе решается задача, противоположная рассмотренной ранее: для целого положительного числа, введенного пользователем в поле ввода, вычисляется и отображается бинарный код. Для выполнения преобразования (вычисления на основе числового значения текстовой строки с бинарным кодом числа) в классе UsingDoWhileDemo описан статический метод coder(). Аргументом методу передается число, а результатом метод возвращает текстовую строку с бинарным кодом этого числа. Метод coder() вызывается в главном методе main() программы, который также описан в классе UsingDoWhileDemo. То есть здесь наша стратегия такая же, как и в предыдущем примере.

Рассмотрим и проанализируем программный код примера.

Листинг 4.4. Программный код проекта UsingDoWhileApplicaion

```
import javax.swing.JOptionPane;
// Класс с главным методом программы:
class UsingDoWhileDemo{
  // Статический метод для определения бинарного
```

```
// кода (результат метода) числа (аргумент метода):
static String coder(int number){
  // Текстовая переменная для записи результата:
  String s="";
  // Число для вычисления бинарного кода:
  int n=number;
  do{
    // Добавление к текстовой строке s слева
    // очередного параметра в бинарном представлении:
    s=(n%2)+s;
    // Значение переменной n уменьшается в два раза:
    n/=2;
  }while(n!=0);
  return s;
}
// Главный метод программы:
public static void main(String[] args){
  // Текстовая переменная для записи значения
  // из поля ввода в диалоговом окне:
  String input;
  // Название для диалоговых окон:
  String title="Вычисление бинарного кода";
  // Отображение диалогового окна для ввода числа:
  input=JOptionPane.showInputDialog(null,
    "Введите целое число", // Надпись над полем ввода
    title,               // Название окна
    JOptionPane.QUESTION_MESSAGE // Тип пиктограммы
  );
  // Если пользователь не ввел число:
  if(input==null){
    // Завершение выполнения программы:
    System.exit(0);
```

```
    }
    // Определение числа на основе текста:
    int num=Integer.parseInt(input);
    // Определение бинарного кода:
    String code=coder(num);
    // Текст для отображения в диалоговом окне:
    String text="Вы ввели число:  "+num;
    text+="\nБинарный код числа:  "+code;
    // Отображение диалогового окна:
    JOptionPane.showMessageDialog(null,
      text,  // Текст сообщения
      title, // Название окна
      JOptionPane.INFORMATION_MESSAGE // Тип пиктограммы
    );
  }
}
```

В теле метода coder() объявляется текстовая переменная s с начальным значением в виде пустой текстовой строки (литерал ""). После «заполнения» данная переменная возвращается результатом метода coder(). Целочисленная переменная n в качестве начального значения получает значение аргумента метода. Вычисления выполняются в теле оператора цикла do-while. Там всего две команды. Сначала командой s=(n%2)+s слева к текущему текстовому значению, на которое ссылается переменная s, дописывается 0 или 1 — остаток от деления текущего значения переменной n на 2. Затем командой n/=2 значение переменной n уменьшается в два раза (выполняется целочисленное деление). Оператор цикла выполняется, пока истинно условие n!=0 (значение переменной n не равно нулю).

ДЕТАЛИ

Если вычислить остаток от деления целого неотрицательного числа на два, то получим значение последнего разряда в бинарном представлении числа. То есть если некоторое число имеет бинарное представление $\overline{a_n a_{n-1} \ldots a_2 a_1 a_0}$, то остаток от деления данного числа на два равен a_0. Далее, если поделить число с бинарным

представлением $\overline{a_n a_{n-1} \ldots a_2 a_1 a_0}$ на два, то получим число с бинарным кодом $\overline{a_n a_{n-1} \ldots a_2 a_1}$. Другими словами, целочисленное деление некоторого числа на два равнозначно отбрасыванию в бинарном представлении этого числа последнего разряда (или младшего бита). На этом базируется наша стратегия вычисления битового кода для числа: на каждом цикле вычисляя остаток от деления переменной n на два (выражение n%2), определяем значение младшего разряда, а затем с помощью команды n/=2 «сдвигаем» вправо битовое представление значения переменной n так, что предпоследний разряд становится последним. На следующем цикле он будет считан как последний разряд. Процесс продолжается до тех пор, пока текущее значение переменной n не станет равным нулю.

Результат метода coder() возвращается командой return s.

В главном методе программы в текстовую переменную input записывается текстовое представление для целого числа, которое пользователь вводит в поле ввода первого диалогового окна. Если пользователь выполнил ввод, в переменную num записывается значение командой Integer.parseInt(input), представляющее собой результат извлечения целого числа из текстового значения input. Вычисление бинарного кода выполняется инструкцией coder(num), а результат записывается в текстовую переменную code. Полученные значения отображаются в диалоговом окне.

На рис. 4.13 показано диалоговое окно с полем ввода, в которое введено целочисленное значение 90, для которого нужно вычислить бинарный код.

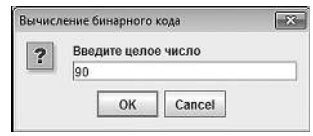


Рис. 4.13. В поле ввода указано число для вычисления его бинарного кода

После подтверждения ввода появляется диалоговое окно, как на рис. 4.14.

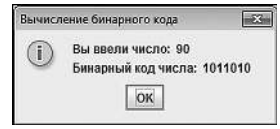


Рис. 4.14. В диалоговом окне указано исходное число и его бинарный код

Несложно заметить, что для числа 90 мы получили бинарный код 1011010, как и должно быть.

Оператор цикла for

У оператора цикла for не самый тривиальный синтаксис, но, несмотря на это, он достаточно удобен в использовании. Шаблон описания оператора цикла for представлен ниже (жирным шрифтом выделены ключевые элементы шаблона):

```
for(инициализация;условие;инкремент/декремент){
  // Команды
}
```

Начинается описание оператора с ключевого слова for, после которого в круглых скобках размещается три блока инструкций. Блоки разделяются точкой с запятой. После круглых скобок в фигурных скобках размещаются команды, формирующие тело оператора цикла.

> **НА ЗАМЕТКУ**
>
> Если тело оператора цикла состоит из одной команды, то фигурные скобки можно не использовать. Однако это не самая блестящая идея и ею лучше не злоупотреблять.
>
> Если блок инструкций в фигурных скобках состоит из нескольких команд, то команды между собой разделяются запятыми.

Схема выполнения оператора цикла for иллюстрируется с помощью рис. 4.15.

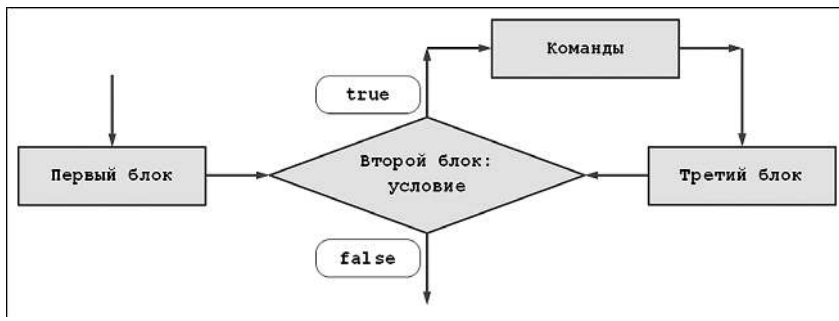


Рис. 4.15. Схема выполнения оператора цикла for

Сначала выполняются команды в первом блоке (в круглых скобках после инструкции for). Обычно этот блок называют блоком *инициализации*, поскольку там удобно размещать команды по присваиванию всевозможным переменным начальных значений. Эти команды выполняются один и только один раз. Затем проверяется условие, которое размещается во втором блоке в for-инструкции (блок *условия*). Если условие истинно, выполняются команды в теле оператора цикла, а затем команды в третьем блоке for-инструкции (называют блоком *инкремента* или *декремента*, поскольку обычно в нем размещают команды изменения счетчиков и индексных переменных). После этого проверяется условие во втором блоке, и при его истинности выполняются команды в теле оператора цикла и в третьем блоке, затем опять проверяется условие, и так далее. Выполнение оператора цикла завершается, если при проверке условия оно окажется ложным.

Пример использования оператора цикла for представлен в программе в листинге 4.5. Программа предназначена для вычисления суммы квадратов натуральных чисел.

НА ЗАМЕТКУ

Речь идет о сумме вида $1^2 + 2^2 + 3^2 + \ldots + n^2$. Имеет место соотношение $1^2 + 2^2 + 3^2 + n^2 = (n(n + 1)(2n + 1))/6$. Данная формула может использоваться для проверки результатов вычисления суммы квадратов натуральных чисел, полученных с помощью описываемой далее программы.

Программный код примера достаточно простой. Рассмотрим его.

Листинг 4.5. Программный код проекта UsingForApplicaion

```
class UsingForDemo{
  // Главный метод программы:
  public static void main(String[] args){
    // Целочисленные переменные:
    int s=0,k,n=10;
    // Вычисление суммы квадратов натуральных чисел:
    for(k=1;k<=n;k++){
      s+=k*k;
    }
```

```
    // Текст для отображения в окне вывода:
    String txt="Сумма квадратов чисел от 1 до "+n+": "+s;
    // Отображение результата вычислений:
    System.out.println(txt);
  }
}
```

Вычисления выполняются в методе main(). Там объявляются три целочисленные переменные: переменная s с начальным значением 0 (переменная для записи суммы квадратов натуральных чисел), индексная переменная k (переменная для подсчета количества слагаемых) и переменная n, значение которой определяет верхнюю границу для суммы. В операторе цикла for в первом блоке командой k=1 присваивается начальное значение переменной k. Затем проверяется условие k<=n во втором блоке. При истинности условия в теле цикла выполняется команда s+=k*k, которой текущее значение переменной s увеличивается на величину k*k (к сумме прибавляется очередное слагаемое), после чего командой k++ в третьем блоке в for-выражении значение индексной переменной k увеличивается на единицу. На следующем этапе снова проверяется условие, и так далее. Оператор цикла выполняется, пока истинно условие k<=n.

В результате выполнения программы в окне вывода появится следующее сообщение:

Результат выполнения программы (из листинга 4.5)

```
Сумма квадратов чисел от 1 до 10: 385
```

Легко проверить, что мы получили правильное значение для суммы.

Сравнение операторов цикла

Операторы цикла позволяют создавать достаточно гибкие программные коды. Причем одна и та же задача может быть реализована разными способами.

Как иллюстрацию рассмотрим некоторые способы реализации следующего блока программного кода из листинга 4.5 (комментарии удалены):

```
int s=0,k,n=10;
for(k=1;k<=n;k++){
  s+=k*k;
}
```

В первую очередь стоит отметить, что переменные можно объявлять непосредственно в теле оператора цикла.

Например, ниже показан фрагмент кода, в котором переменная k объявляется непосредственно в первом блоке в for-инструкции:

```
int s=0,n=10;
for(int k=1;k<=n;k++){
  s+=k*k;
}
```

Однако поскольку переменные доступны в пределах того блока, где они объявлены (а блок определяется парой фигурных скобок), то переменная k в данном случае будет доступна только в теле оператора цикла и не будет доступна за его пределами.

Как отмечалось ранее, блоки в for-выражении могут содержать несколько команд. Такая ситуация представлена ниже:

```
int s,k,n=10;
for(k=1,s=0;k<=n;s+=k*k,k++);
```

Здесь начальные значения переменным k и s присваиваются в первом блоке оператора цикла, а команда s+=k*k из тела оператора цикла вынесена в третий блок (в результате тело оператора цикла не содержит команд).

Код, в котором в операторе цикла for когда первый и третий блоки пустые, выглядит следующим образом:

```
int s=0,k=1,n=10;
for(;k<=n;){
  s+=k*k;
  k++;
}
```

Пустым может быть и второй блок (блок условия). В таком случае получаем формально бесконечный оператор цикла (то есть условие считается истинным):

```
int s=0,k=1,n=10;
for(;;){
  if(k>n) break;
  s+=k*k;
  k++;
}
```

Для выхода из оператора цикла мы использовали инструкцию break, которая выполняется в условном операторе при истинном условии k>n.

Для сравнения, тот же блок кода с использованием оператора цикла while мог бы быть реализован следующим образом:

```
int s=0,k=1,n=10;
while(k<=n){
  s+=k*k;
  k++;
}
```

Практически идентичен ему такой код, в котором использован оператор цикла do-while:

```
int s=0,k=1,n=10;
do{
  s+=k*k;
  k++;
}while(k<=n);
```

Правда, в случае с оператором do-while команды в теле цикла выполняются как минимум один раз (первый раз условие проверяется после того, как выполнены команды в теле цикла). При положительных значениях переменной n это не принципиально. Если же переменной n присвоить отрицательное или нулевое значение, то код на основе оператора do-while будет давать для переменной s иное значение, по сравнению с прочими рассмотренными выше кодами.

Оператор выбора

Почки заячьи перченые, головы щучьи с чесноком... Икра черная, красная... Да, заморская икра, баклажанная.

Из к/ф «Иван Васильевич меняет профессию»

Оператор выбора switch позволяет выполнять разные блоки команд в зависимости от значения некоторого выражения. Важное ограничение состоит в том, что выражение может быть либо целочисленным, либо символьным.

Описание оператора выбора начинается с ключевого слова switch, после которого в круглых скобках указывается проверяемое выражение. Тело оператора выбора заключается в фигурные скобки. В них один за другим следуют case-блоки:

- после ключевого слова case указывается контрольное значение, которое будет сравниваться со значением выражения;
- после контрольного значения ставится двоеточие;
- далее следуют команды данного case-блока, которые обычно (но не всегда) заканчиваются инструкцией break;
- последним может указываться блок, помеченный ключевым словом default.

Общий синтаксис оператора выбора определяется таким шаблоном (жирным шрифтом выделены ключевые элементы шаблона):

```
switch(выражение){
  case значение:
    // Команды
    break;
  case значение:
    // Команды
    break;
  // Другие case-блоки
  case значение:
    // Команды
```

```
    break;
  default:
    // Команды
}
```

Общая схема выполнения оператора выбора иллюстрируется с помощью рис. 4.16.

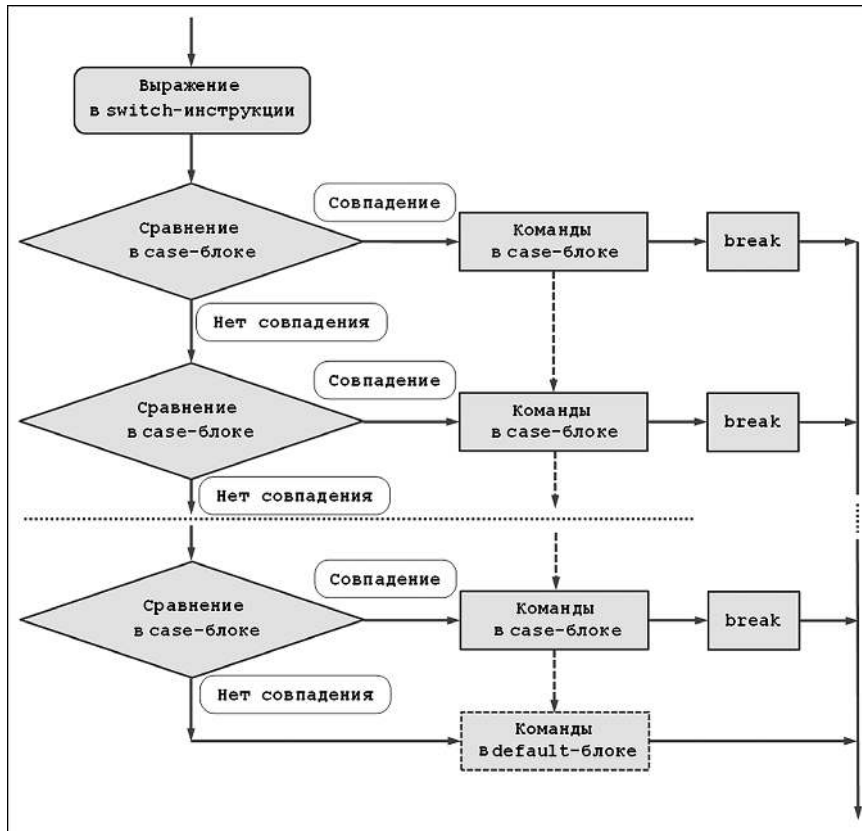


Рис. 4.16. Схема выполнения оператора выбора

При выполнении оператора выбора вычисляется значение выражения, указанного в круглых скобках после ключевого слова switch. Затем вычисленное значение сравнивается с контрольными значениями, указанными в case-блоках — процесс продолжается до первого совпадения. Как только совпадение значений найдено, начинают выполняться команды

в соответствующем case-блоке. Пикантность ситуации в том, что выполняются они не до окончания блока, а до конца оператора цикла. Проще говоря, команды выполняются не только в том case-блоке, где найдено совпадение значений, но и в следующих блоках. Чтобы выполнение команд прекратилось раньше, используют инструкцию break. В частности, если необходимо, чтобы выполнялись команды только в том блоке, где имеет место совпадение значения выражения и контрольного значения, блок заканчивается инструкцией break. Последним необязательным блоком может быть default-блок. Команды в default-блоке выполняются, если при проверке контрольных значений в case-блоках совпадений со значением выражения не было. Поскольку default-блок является последним, в конце этого блока инструкцию break не используют.

НА ЗАМЕТКУ

Использование инструкции break оператором выбора не ограничивается. Инструкцию break можно использовать для завершения работы оператора цикла.

Небольшой пример использования оператора выбора представлен в программе в листинге 4.6. В программе (кроме главного метода программы) в классе UsingSwitchDemo описано два статических метода, в каждом из которых используется оператор выбора. Методом getDay() по номеру дня недели возвращается текст с названием дня. Методом testDay() по номеру дня недели возвращается текст, определяющий будний это день или праздничный. В главном методе программы указанные методы вызываются с разными аргументами. Теперь рассмотрим программный код.

Листинг 4.6. Программный код проекта UsingSwitchApplicaion

```
class UsingSwitchDemo{
  // Метод для определения дня недели по номеру:
  static String getDay(int num){
    // Текстовая переменная для записи результата:
    String day;
    // Оператор выбора для определения дня недели:
    switch(num){
      case 1:
        day="понедельник";
```

```
        break;
      case 2:
        day="вторник";
        break;
      case 3:
        day="среда";
        break;
      case 4:
        day="четверг";
        break;
      case 5:
        day="пятница";
        break;
      case 6:
        day="суббота";
        break;
      case 7:
        day="воскресенье";
        break;
      default:
        day="неизвестно";
    }
    // Результат метода:
    return day;
  }
  // Метод для определения будних и выходных дней:
  static String testDay(int num){
    // Переменная для записи результата:
    String day;
    // Оператор выбора для определения
    // будних и рабочих дней:
    switch(num){
      case 1:
```

```
      case 2:
      case 3:
      case 4:
      case 5:
        day="будний день";
        break;
      case 6:
      case 7:
        day="выходной день";
        break;
      default:
        day="неизвестно";
    }
    // Результат метода:
    return day;
  }
  // Главный метод программы:
  public static void main(String[] args){
    // Оператор цикла:
    for(int k=0;k<=8;k++){
      // Вызов статических методов:
      System.out.println(k+": "+getDay(k)+"\t- "+testDay(k));
    }
  }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 4.6)

```
0: неизвестно    — неизвестно
1: понедельник   — будний день
2: вторник       — будний день
3: среда         — будний день
```

```
4: четверг      — будний день
5: пятница      — будний день
6: суббота      — выходной день
7: воскресенье  — выходной день
8: неизвестно   — неизвестно
```

В теле метода getDay() по значению целочисленного аргумента num с помощью оператора выбора определяется значение текстовой переменной day, которая в итоге возвращается результатом метода. При этом значениям аргумента в диапазоне от 1 до 7 соответствуют названия дней недели, а для прочих числовых значений методом возвращается текстовое значение "неизвестно".

Похожая ситуация имеет место в методе testDay(), но там вариантов меньше: для числовых значений аргумента от 1 до 5 возвращается значение "будний день", для значений аргумента 6 и 7 возвращается значение "выходной день", а для прочих значений аргумента возвращается значение "неизвестно". Чтобы не дублировать одинаковые команды отдельно для каждого case-блока, мы некоторые блоки оставили пустыми. Такие блоки все равно проверяются, и правило остается неизменным: при наличии совпадения проверяемого выражения и контрольного значения в case-блоке выполняются все команды от места совпадения до конца тела оператора выбора или первой инструкции break.

В главном методе программы запускается оператор цикла, в котором переменная k пробегает значения в диапазоне от 0 до 8 включительно. Для каждого значения переменной k вызываются методы getDay() и testDay() с соответствующим аргументом. Результаты вызова методов выводятся в окно вывода.

НА ЗАМЕТКУ

В тексте мы использовали управляющий символ \t, представляющий собой инструкцию вставки символа табуляции.

Резюме

Если мы допустим беспорядок в документации, потомки нам этого не простят.

Из к/ф «Гостья из будущего»

- Условный оператор if позволяет выполнять разные блоки команд в зависимости от истинности или ложности некоторого условия. Условие указывается в круглых скобках после ключевого слова if. Если условие истинно, выполняется блок команд в фигурных скобках после if-инструкции. Если условие ложно, выполняется блок команд (в фигурных скобках) после ключевого слова else. Также существует упрощенная форма условного оператора без else-блока.
- Оператор выбора switch позволяет выполнять разные блоки команд в зависимости от значения некоторого выражения (числового или символьного типа). Проверяемое выражение указывается в круглых скобках после ключевого слова switch. Далее следуют блоки команд. Каждый блок начинается ключевым словом case, после которого указывается контрольное значение. Значение выражения последовательно сравнивается с контрольными значениями в case-блоках до первого совпадения. Если совпадение найдено, выполняются команды, начиная с соответствующего case-блока. Команды выполняются до конца тела оператора выбора или пока не встретится инструкция break (поэтому обычно case-блоки заканчиваются инструкцией break). В операторе выбора может быть default-блок, команды которого выполняются, если при сравнении значения выражения с контрольными значениями совпадений не найдено.
- Оператор цикла while позволяет многократно выполнять блок команд. После ключевого слова while в круглых скобках указывается некоторое условие. Если условие истинно, выполняются команды в теле оператора цикла (размещаются в фигурных скобках после while-инструкции). После выполнения команд снова проверяется условие, и так далее: для продолжения выполнения оператора цикла необходимо, чтобы при проверке условия оно было истинным.
- Оператор цикла do-while напоминает оператор цикла while, однако имеет более сложный синтаксис. Описание оператора начинается с ключевого слова do, после которого в фигурных скобках размещается блок команд. После блока команд указывается ключевое слово while и, в круглых скобках, условие (выражение, возвращающее значение логического типа). При выполнении оператора цикла выполняются команды в фигурных скобках, после чего проверяется условие в while-инструкции. Если условие истинно, команды выполняются еще раз, после чего снова выполняется условие, и так далее. Выполнение оператора цикла прекращается, когда при проверке условия оно оказывается ложным.

- Описание оператора цикла for начинается с ключевого слова for, после которого в круглых скобках указывается три блока инструкций. Блоки между собой разделяются точками с запятой. Если в блоке несколько команд, они разделяются запятыми. Команды в первом блоке выполняются только один раз в начале выполнения оператора цикла. Затем проверяется условие во втором блоке. Если условие истинно, выполняются команды в теле оператора цикла (блок в фигурных скобках после for-инструкции), затем команды в третьем блоке в for-инструкции. После этого проверяется условие, и так далее. Выполнение оператора цикла прекращается, если при проверке условия оно окажется ложным.

Глава 5
МАССИВЫ

Как говорит наш дорогой шеф, в нашем деле главное — этот самый реализм.

Из к/ф «Бриллиантовая рука»

В этой главе мы обсудим *массивы*. Массив представляет собой набор однотипных данных, объединенных общим именем. Доступ к элементу массива выполняется через имя массива и индекс или индексы, идентифицирующие место, или «позицию», элемента в массиве. Существует такое понятие, как *размерность* массива, — это количество индексов, необходимых для однозначной идентификации элемента массива. Теоретически массив может быть практически любой размерности. Но на практике массивы размерности большей, чем два, используются редко. Наиболее простыми являются *одномерные* массивы.

Одномерные массивы

Посторонние разговоры прекратить. Операция началась. За мной!

Из к/ф «Старики-разбойники»

В одномерном массиве элемент идентифицируется с помощью одного индекса. Количество элементов в одномерном массиве называется *размером* массива. Далее мы рассмотрим процесс создания и использования одномерных массивов базовых (простых) типов.

Создание одномерного массива

Массив в Java создается практически по той же схеме, что и объекты. А именно объявляется *переменная массива*, которая не является массивом, а лишь содержит ссылку на массив. Сам массив создается с помощью оператора `new`. Таким образом, создание массива подразумевает:

- объявление переменной массива;
- собственно создание массива и запись ссылки на этот массив в переменную массива.

Переменная массива объявляется подобно обычной переменной, однако после идентификатора типа, определяющего тип элементов массива, указываются пустые квадратные скобки. То есть команда объявления переменной массива имеет такой вид:

```
тип[] имя_переменной;
```

Пустые квадратные скобки можно указывать не после идентификатора типа, а после имени переменной:

```
тип имя_переменной[];
```

Мы в основном будем использовать первый способ объявления переменной массива (когда пустые квадратные скобки указываются после идентификатора типа).

Для создания массива используют оператор new, после которого указывается тип элементов создаваемого массива. Размер массива (количество элементов в массиве) указывается в квадратных скобках после идентификатора типа элементов массива. Следовательно, при создании массива используем такой шаблон:

```
new тип[размер];
```

При выполнении команды с new-инструкцией создается массив. Результатом инструкцией возвращается ссылка на массив. Ссылку на массив записывают в переменную массива. Таким образом, вся процедура по созданию массива и записи ссылки на него в переменную массива реализуется командами следующего вида:

```
тип[] переменная;
переменная=new тип[размер];
```

Данные команды можно объединить в одну:

```
тип[] переменная=new тип[размер];
```

Например, ниже приведены команды, которыми создается массив nums из десяти элементов типа int, а также массив symbs из пяти элементов типа char:

```
// Целочисленный массив:
int[] nums=new int[10];
// Переменная массива:
char[] symbs;
// Массив символов:
symbs=new char[5];
```

При обращении к массиву используется переменная массива, которую мы будем отождествлять с массивом. Вместе с тем при выполнении некоторых операций с массивами (такими, как присваивание массивов или передача массива аргументом методу) следует все же помнить, что переменная массива лишь ссылается на массив. Ситуацию иллюстрирует рис. 5.1.

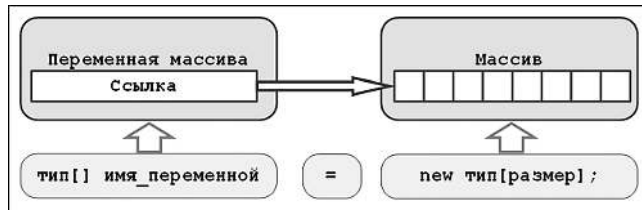


Рис. 5.1. Переменная массива ссылается на массив

Обращение к элементам массива выполняется так: после имени переменной массива в квадратных скобках указывается индекс элемента массива.

Индексация элементов массива начинается с нуля. Поэтому индекс последнего элемента в массиве на единицу меньше размера массива. Размер массива, в свою очередь, можно узнать с помощью свойства `length` массива. В частности, значением выражения вида `переменная_массива.length` является количество элементов в массиве (размер массива), на который ссылается `переменная_массива`.

Небольшой пример с использованием массива приведен в листинге 5.1. В представленной программе на основе неотрицательного целочисленного значения, которое вводит пользователь в поле ввода, создается целочисленный массив, который заполняется биномиальными коэффициентами. Затем значения элементов массива отображаются в диалоговом окне.

НА ЗАМЕТКУ

По определению биномиальный коэффициент $C_n^m = n!/(m!(n - m)!)$. При заданном нижнем индексе n верхний индекс $0 \le m \le n$. Через $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$ обозначен факториал числа — произведение натуральных чисел от единицы до данного числа включительно.

Существуют некоторые простые соотношения для биномиальных коэффициентов. Например, непосредственно из определения биномиальных коэффициентов следует, что $C_n^{n-m} = C_n^m$, $C_n^0 = 1$, $C_n^1 = n$ и $C_n^2 = (n(n - 1))/2$. Мы при заполнении массива будем использовать соотношение $C_n^m = C_n^{m-1} \cdot (n - m + 1)/m$ и то обстоятельство, что $C_n^0 = 1$.

Рассмотрим и проанализируем программный код примера.

Листинг 5.1. Программный код проекта UsingArrayApplication

```
import javax.swing.JOptionPane;
class UsingArrayDemo{
  public static void main(String[] args){
    // Текстовые переменные для записи значения в поле
    // ввода и названия для диалоговых окон:
    String input,title="Биномиальные коэффициенты";
    // Отображение окна с полем ввода:
    input=JOptionPane.showInputDialog(null,
      // Надпись над полем ввода:
      "Укажите значение нижнего индекса",
      // Название окна:
      title,
      // Тип пиктограммы:
      JOptionPane.QUESTION_MESSAGE
    );
    // Если пользователь отменил ввод:
    if(input==null){
      // Завершение выполнения программы:
      System.exit(0);
    }
```

```
// Переменная для записи значения нижнего индекса:
int n;
// Определение числового значения на основе текста:
n=Integer.parseInt(input);
// Проверка корректности числового значения:
if(n<0){ // Если введено некорректное значение
  // Отображение диалогового окна:
  JOptionPane.showMessageDialog(null,
    // Текст сообщения:
    "Указан неверный параметр!",
    // Название окна:
    title,
    // Тип пиктограммы:
    JOptionPane.ERROR_MESSAGE
  );
  // Завершение выполнения программы:
  System.exit(0);
}
// Создание массива:
int[] binoms=new int[n+1];
// Значение первого элемента массива:
binoms[0]=1;
// Текст для отображения в диалоговом окне:
String txt="Содержимое массива:\n| "+binoms[0]+" |";
// Вычисление значений элементов массива:
for(int m=1;m<binoms.length;m++){
  // Значение элемента:
  binoms[m]=binoms[m-1]*(n-m+1)/m;
  // В текст дописывается значение элемента:
  txt+=" "+binoms[m]+" |";
}
// Отображение диалогового окна:
```

```
    JOptionPane.showMessageDialog(null,
      // Текст сообщения:
      txt,
      // Название окна:
      title,
      // Тип пиктограммы:
      JOptionPane.INFORMATION_MESSAGE
    );
  }
}
```

При выполнении программы отображается диалоговое окно с полем для ввода значения нижнего индекса вычисляемых биномиальных коэффициентов (речь об индексе n в биномиальном коэффициенте C_n^m). Введенное пользователем значение (в виде текста) записывается в переменную input. Значение переменной проверяется: если оно равно null (пустая ссылка — такое значение у переменной будет, если пользователь закрыл окно с полем ввода щелчком на системной пиктограмме с крестиком или щелкнул кнопку **Cancel**), то командой System.exit(0) выполнение программы завершается. В противном случае командой n=Integer.parseInt(input) в целочисленную переменную n записывается результат «извлечения» из текстовой переменной input целочисленного значения. Полученное значение проверяется на корректность. В частности, оно должно быть неотрицательным. Поэтому в условном операторе проверяется условие n<0, и если условие истинно, то сначала отображается окно с сообщением о некорректном значении параметра, после чего прекращается выполнение программы. Если завершать программу причин нет, то командой int[] binoms=new int[n+1] создается массив binoms с целочисленными элементами. Размер массива на единицу больше значения переменной n.

НА ЗАМЕТКУ

Если нижний индекс n для биномиальных коэффициентов C_n^m задан, то верхний индекс m принимает значения $m = 0, 1, \ldots, n$ — всего $n + 1$ значение. Таким образом, в общей сложности существует $n + 1$ биномиальный коэффициент с нижним индексом n. Поэтому размер массива, предназначенного для записи биномиальных коэффициентов, на единицу больше значения нижнего индекса.

Значение первому элементу массива присваивается командой binoms[0]=1 (здесь мы учли, что $C_n^0 = 1$). Значения прочим элементам присваиваются в операторе цикла. Там объявляется индексная переменная m, которая определяет индекс элемента массива, которому присваивается значение. Начальное значение переменной m равно 1, а оператор цикла выполняется, пока истинно условие m<binoms.length. Значение выражения binoms.length — это количество элементов в массиве binoms. Поскольку в условии m<binoms.length использовано строгое неравенство, то последнее значение переменной m, для которого еще выполняются команды в теле цикла, равно binoms.length-1 — значение, на единицу меньшее количества элементов в массиве и равное индексу последнего элемента массива.

За каждый цикл при заданном значении переменной m командой binoms[m]=binoms[m-1]*(n-m+1)/m на основе значения предыдущего элемента binoms[m-1] вычисляется значение очередного элемента binoms[m].

Следующей командой txt+=" "+binoms[m]+" |" вычисленное значение (вместе с пробелами и разделительной вертикальной чертой) «дописывается» к текстовой переменной txt. Ее начальное значение содержит вводный текст, инструкцию перехода к новой строке \n, значение первого элемента массива и «декоративные» пробелы и вертикальные черточки. В итоге по завершении оператора цикла массив binoms заполнен биномиальными коэффициентами, а текстовая переменная txt содержит (на самом деле ссылается на текстовый объект) значения этих элементов. Остается только отобразить в диалоговом окне значение переменной txt, что, собственно, и делается.

При запуске программы на выполнение появляется диалоговое окно, как показано на рис. 5.2.

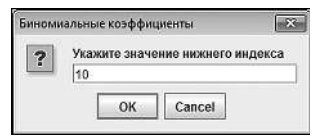


Рис. 5.2. В поле ввода указывается значение нижнего индекса для биномиальных коэффициентов

Если в поле ввода введено корректное (неотрицательное целое число) значение, то на следующем этапе появится окно со значениями биномиальных коэффициентов, как показано на рис. 5.3.

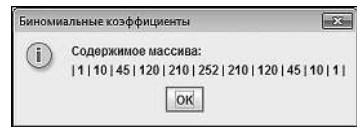


Рис. 5.3. Результат вычисления биномиальных коэффициентов

Если в поле ввода пользователь указал отрицательное целое число, то появится окно, как на рис. 5.4.

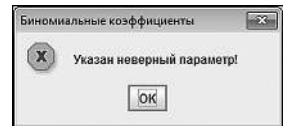


Рис. 5.4. Данное окно появляется при некорректно указанном значении для нижнего индекса биномиальных коэффициентов

Также напомним, что если в первом диалоговом окне (см. рис. 5.2) щелкнуть кнопку **Cancel** (или системную пиктограмму закрытия окна), то программа просто прекратит выполнение и второе диалоговое окно отображаться не будет.

Инициализация одномерного массива

В рассмотренном выше примере после создания массива для присваивания значений его элементам использовался оператор цикла, в котором перебирались элементы массива — все, за исключением первого (с нулевым индексом). Такой подход возможен, если значения элементов массива подчиняются некоторой «логике». Но это имеет место далеко не всегда.

Если не использовать оператор цикла, то значение каждому элементу пришлось бы присваивать в явном виде, что не только неудобно, но в большинстве случаев просто невозможно. Поэтому нередко используют иной механизм, который называется *инициализацией массива* и состоит в том, что при создании массива для его элементов в явном виде указываются значения. Значения, которыми инициализируется массив, оформляются в виде заключенного в фигурные скобки списка (значения в списке разделяются запятыми), который присваивается в качестве значения переменной массива. Например, ниже приведена команда объявления и инициализации целочисленного массива из трех элементов:

```
int[] nums={-1,2,1};
```

Первому элементу массива присваивается значение -1, второй элемент массива получает значение 2, а третьему элементу присваивается значение 1. Альтернативный способ объявления и инициализации массива подразумевает использование оператора new:

```
int[] nums=new int[]{-1,2,1};
```

Здесь мы получаем такой же результат, как и в предыдущем случае, но инструкция немного длиннее.

Как пример объявления и инициализации массивов рассмотрим пример, в котором сначала отображается диалоговое окно с раскрывающимся списком. В раскрывающемся списке выбирается название животного (*волк*, *лиса*, *медведь* или *енот*), после чего открывается новое диалоговое окно с соответствующим изображением. В программе мы реализуем названия животных и названия файлов с их изображениями в виде текстовых массивов (элементами массива являются текстовые значения). При этом массивы при объявлении инициализируются списками из текстовых значений. Далее обратимся к программному коду в листинге 5.2.

Листинг 5.2. Программный код проекта UsingListApplication

```
import javax.swing.*;
class UsingListDemo{
  public static void main(String[] args){
    // Текстовый массив с названиями животных:
    String[] txt={"Волк","Лиса","Медведь","Енот"};
    // Текстовый массив из названий файлов
    // с изображениями животных:
    String[] files={"wolf.jpg","fox.jpg","bear.jpg","raccoon.jpg"};
    // Текстовая переменная, определяющая путь
    // к файлам с изображениями животных:
    String folder="d:/books/pictures/";
    // Текст для отображения в диалоговом окне:
    String msg="Кого показать?";
    // Текст с названием диалогового окна:
```

```
String title="В мире животных";
// Объект для пиктограммы, отображаемой
// в диалоговом окне:
ImageIcon img=new ImageIcon(folder+"giraffe.png");
// Текстовая переменная для записи выбранного
// пользователем животного:
String animal;
// Отображение диалогового окна
// с раскрывающимся списком для выбора животного:
animal=(String)JOptionPane.showInputDialog(null,
        msg,   // Текст над раскрывающимся списком
        title, // Название окна
        JOptionPane.PLAIN_MESSAGE, // Тип окна
        img, // Пиктограмма, отображаемая в окне
        txt,   // Элементы раскрывающегося списка
        txt[0] // Выбранный по умолчанию элемент
);
// Если пользователь отменил ввод:
if(animal==null){
  // Завершение выполнения программы:
  System.exit(0);
}
// Определение пиктограммы для отображения
// в диалоговом окне:
for(int k=0;k<txt.length;k++){
  // Если текст в переменной animal совпадает
  // с текстовым значением элемента в массиве:
  if(animal.equals(txt[k])){
    // Создание объекта пиктограммы:
    img=new ImageIcon(folder+files[k]);
    // Завершение оператора цикла:
    break;
```

```
      }
    }
    // Отображение диалогового окна с пиктограммой:
    JOptionPane.showMessageDialog(null,
      img,   // Отображаемое изображение
      animal, // Название окна
      JOptionPane.PLAIN_MESSAGE // Тип окна
    );
  }
}
```

Сначала рассмотрим результат выполнения программы. При выполнении программы отображается диалоговое окно (с названием **В мире животных**), содержащее пользовательскую пиктограмму (с изображением жирафа) и раскрывающемся списком из четырех позиций. По умолчанию в раскрывающемся списке выбран пункт **Волк**. На рис. 5.5 показано означенное диалоговое окно с раскрытым списком выбора.

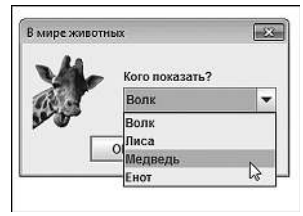


Рис. 5.5. Диалоговое окно содержит пользовательскую пиктограмму и раскрывающийся список

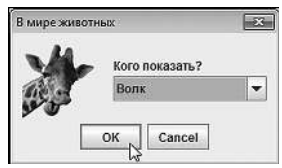
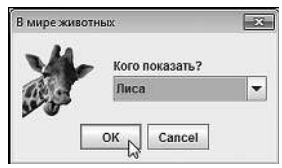
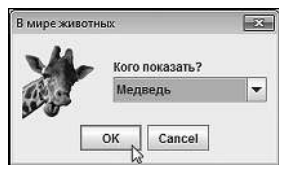
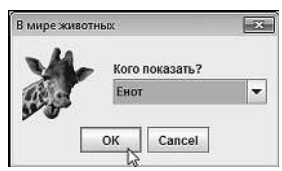
После того как выбор в раскрывающемся списке сделан, следует щелкнуть на кнопке **ОК**. Окно с раскрывающимся списком закроется, но откроется диалоговое окно с изображением выбранного животного. Возможные варианты перечислены в табл. 5.1.

Если в окне с раскрывающемся списком вместо кнопки **ОК** щелкнуть кнопку **Cancel**, то выполнение программы будет завершено.

Далее проанализируем программный код примера. В программе (главном методе) командой `String[] txt={"Волк","Лиса","Медведь","Енот"}` создается

и инициализируется текстовый массив txt из четырех элементов. Каждый из элементов значением получает название животного из списка значений "Волк", "Лиса", "Медведь" и "Енот". Еще один текстовый массив files создается командой String[] files={"wolf.jpg","fox.jpg","bear.jpg","raccoon.jpg"}. Значениями элементов данного массива являются названия файлов с изображениями зверей. При этом между элементами массивов txt и files существует соответствие: названию из массива txt соответствует файл с названием массива files на такой же позиции (с таким же индексом).

Табл. 5.1. Результат выбора с помощью раскрывающегося списка

Позиция в списке	Результат	Пояснение
		В раскрывающемся списке выбрана позиция **Волк**. После щелчка на кнопке **OK** отображается новое диалоговое окно (с названием **Волк**) с изображением волка
		В раскрывающемся списке выбрана позиция **Лиса**. После щелчка на кнопке **OK** отображается новое диалоговое окно (с названием **Лиса**) с изображением лисы
		В раскрывающемся списке выбрана позиция **Медведь**. После щелчка на кнопке **OK** отображается новое диалоговое окно (с названием **Медведь**) с изображением медведя
		В раскрывающемся списке выбрана позиция **Енот**. После щелчка на кнопке **OK** отображается новое диалоговое окно (с названием **Енот**) с изображением енота

Помимо массивов, объявляется текстовая переменная folder со значением "d:/books/pictures/", определяющим папку, в которой хранятся файлы с изображениями. В переменную msg записывается текст "Кого показать?", отображаемый над раскрывающимся списком в диалоговом окне. Переменная title значением содержит текст "В мире животных", определяющий

название окна. Текстовая переменная animal предназначена для записи результата выбора пользователя в окне с раскрывающимся списком.

Командой ImageIcon img=new ImageIcon(folder+"giraffe.png") создается объект пиктограммы на основе файла giraffe.png (файл с изображением жирафа). Пиктограмма используется при отображении диалогового окна с раскрывающимся списком. Такое окно отображается с помощью статического метода showInputDialog() класса JOptionPane. Ранее мы использовали данный метод для отображения окна с полем ввода. Как видим, данный метод позволяет отобразить окно с раскрывающимся списком — весь вопрос в том, какие аргументы передаются. Описание аргументов, переданных в метод showInputDialog() при отображении с его помощью окна с раскрывающимся списком, приведено в табл. 5.2.

Табл. 5.2. Аргументы метода showInputDialog()

Номер аргумента	Значение	Описание
1	null	Ссылка на родительское окно — в данном случае пустая, поскольку такого окна нет
2	msg	Этим аргументом определяется текст над раскрывающимся списком в диалоговом окне
3	title	Аргументом задается название для диалогового окна
4	JOptionPane.PLAIN_MESSAGE	Аргумент определяет тип сообщения. В данном случае конкретное значение аргумента роли не играет и в любом случае отображается пользовательская пиктограмма (определяется следующим аргументом)
5	img	Аргументом определяется пиктограмма, которая отображается в левой части основной области диалогового окна
6	txt	Массив, элементы которого определяют пункты раскрывающегося списка
7	txt[0]	Элемент массива, который соответствует пункту списка, выбранному по умолчанию

Результат вызова метода showInputDialog() присваивается значением переменной animal. Но перед инструкцией вызова метода есть инструкция (String) приведения типа: результат метода showInputDialog() к типу String. Дело в том, что если выбор делается в окне с раскрывающимся списком, то результатом возвращается ссылка на выбранный пункт, причем ссылка эта класса Object. Класс Object — класс, который находится в вершине иерархии классов Java. Иерархия реализуется на основе *наследования*

(наследование обсуждается в следующей главе). В данном случае подробности нас волнуют мало, а практические последствия такие: результатом метода showInputDialog() является некоторая ссылка на объект, а класс этого объекта необходимо указать в явном виде, через инструкцию приведения типа. Приведение типа выполняется просто: перед соответствующим выражением в круглых скобках указывается идентификатор типа, к которому приводится выражение.

В условном операторе проверяется значение переменной animal. Если оно рано null (пустая ссылка — если пользователь отменил ввод), то командой System.exit(0) выполнение программы завершается. Если пользователь выбрал пункт в раскрывающемся списке и щелкнул кнопку **ОК**, то запускается оператор цикла, в котором с помощью индексной переменной k перебираются все возможные значения индексов элементов из массива txt (при этом размер массива определяется инструкцией txt.length). В теле оператора цикла посредством условного if-оператора проверяется условие animal.equals(txt[k]). Условие фактически состоит в том, что текст в переменной animal совпадает с текстовым значением элемента txt[k]. Для сравнения текстовых значений мы использовали метод equals(), вызываемый из одного текстового объекта, а аргументом методу передается другой текстовый объект. Если совпадение найдено, то командой img=new ImageIco n(folder+files[k]) создается объект для пиктограммы и ссылка на объект присваивается переменной img. После этого инструкцией break выполнение оператора цикла завершается.

ДЕТАЛИ

Результатом метода showInputDialog() возвращается ссылка на тот элемент в списке, который выбрал пользователь. В данном случае это текст. Нас интересует индекс соответствующего (выбранного) элемента в массиве txt. Индекс нам нужен, чтобы по этому индексу определить название файла с изображением из массива files. Собственно поэтому запускается оператор цикла, в котором ищется совпадение считанного текста и значений текстовых элементов в массиве txt.

После завершения выполнения оператора цикла вызовом статического метода showMessageDialog() из класса JOptionPane отображается диалоговое окно с изображением. Первым аргументом методу передается пустая ссылка null на родительское окно (окна нет), вторым аргументом передается переменная img (определяет отображаемое в окне изображение), третьим аргументом передается переменная animal (определяет название

окно), четвертый аргумент JOptionPane.PLAIN_MESSAGE означает, что окно не содержит пиктограммы в левой части основной области окна.

Оператор цикла for по коллекции

Существует специальная форма оператора цикла for, которая позволяет перебирать не индексы элементов массива, а собственно элементы, входящие в массив. Обычно эту форму оператора цикла for называют *циклом по коллекции*.

Синтаксис объявления оператора цикла по коллекции достаточно простой: после ключевого слова for в круглых скобках объявляется переменная того же типа, что и элементы массива, а через двоеточие указывается сам массив. В фигурных скобках указываются команды тела оператора цикла — как и в обычном операторе цикла for. То есть синтаксис оператора цикла по коллекции выглядит следующим образом (жирным шрифтом выделены ключевые элементы шаблона):

```
for(тип переменная: массив){
  // Команды
}
```

Принцип выполнения оператора цикла по коллекции весьма прост: объявленная в операторе переменная последовательно получает значение элементов массива. Например, в приведенном ниже выражении подразумевается, что символьная (тип char) переменная p последовательно принимает значения элементов из массива symbols (состоящего из символов):

```
for(char p: symbols){
  // Команды
}
```

Таким образом, в операторе цикла по коллекции перебираются не индексы элементов массива, а сами элементы массива. Такой способ обработки массива в некоторых случаях бывает удобен. Рассмотрим по этому поводу небольшой пример, представленный в листинге 5.3. Там создается целочисленный массив, и затем он заполняется случайными числами (в диапазоне возможных значений от 1 до 10). Содержимое массива отображается в диалоговом окне. При отображении содержимого

массива мы используем два разных оператора цикла, выполняя перебор как индексов элементов, так и непосредственно элементов массива. Теперь рассмотрим интересующий нас программный код.

Листинг 5.3. Программный код проекта UsingForArrayApplication

```
// Импорт классов:
import javax.swing.JOptionPane;
import java.util.Random;
// Класс с главным методом программы:
class UsingForArrayDemo{
  // Главный метод программы:
  public static void main(String[] args){
    // Размер массива:
    int size=10;
    // Объект для генерирования случайных чисел:
    Random rnd=new Random();
    // Создание массива:
    int[] nums=new int[size];
    // Текст для отображения в диалоговом окне:
    String txt="Массив случайных чисел:\n| ";
    // Заполнение массива случайными числами:
    for(int k=0;k<nums.length;k++){
      // Присваивание значением элементу
      // случайного числа:
      nums[k]=rnd.nextInt(10)+1;
      // Добавление к тексту значения элемента:
      txt+=nums[k]+" | ";
    }
    // Дополнение текстового значения:
    txt+="\nПроверка:\n| ";
    // Отображение элементов с помощью
    // оператора цикла по коллекции:
    for(int s: nums){
      // Добавление в текст значения элемента:
```

```
      txt+=s+" | ";
    }
    // Отображение диалогового окна:
    JOptionPane.showMessageDialog(null,
      txt, // Текст сообщения в окне
      "Случайные числа", // Название окна
      JOptionPane.PLAIN_MESSAGE // Тип окна
    );
  }
}
```

Окно, которое появляется в результате выполнения программы, может выглядеть так, как показано на рис. 5.6.

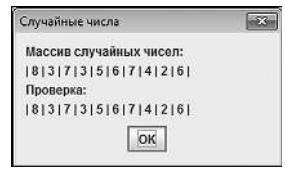


Рис. 5.6. При выполнении программы отображается диалоговое окно со списком случайных чисел

НА ЗАМЕТКУ

Конкретные числовые значения в диалоговом окне изменяются от запуска к запуску, поскольку здесь речь идет о случайных числах. Однако в любом случае обе строки с числами совпадают.

В программе, кроме класса JOptionPane, используется еще и класс Random. Поэтому программа начинается инструкциями import javax.swing.JOptionPane и import java.util.Random, которыми импортируются соответственно классы JOptionPane и Random. В классе JOptionPane нас интересует метод showMessageDialog(), которым отображается диалоговое окно с сообщением. Класс Random понадобился нам для создания объекта, посредством которого будут генерироваться случайные числа.

В теле главного метода в целочисленную переменную size записывается числовое значение, определяющее размер числового массива. Сам массив создается командой int[] nums=new int[size].

Для генерирования случайных чисел командой Random rnd=new Random() создается объект rnd класса Random. Объект rnd используется в операторе цикла, в котором индексная переменная k пробегает значения от 0 до nums.length-1. Таким образом, индексная переменная k перебирает индексы элементов массива. В теле оператора цикла при заданном значении переменной k командой nums[k]=rnd.nextInt(10)+1 присваивается значение элементу массива с соответствующим индексом. Здесь из объекта rnd вызывается метод nextInt(). Результатом метод возвращает случайное целое число. Нижняя граница диапазона значений для генерируемых случайных чисел равна 0. Верхняя граница диапазона значений для генерируемых случайных чисел на единицу меньше аргумента, переданного методу nextInt() при вызове. Таким образом, результатом выражения rnd.nextInt(10) является целое случайное число в диапазоне значений от 0 до 9 включительно. Если к выражению rnd.nextInt(10) прибавить единицу, получим случайное целое число в диапазоне возможных значений от 1 до 10.

После того как значение элементу массива nums[k] присвоено, командой txt+=nums[k]+" | " к текущему значению текстовой переменной txt добавляется значение элемента массива. Затем значения элементов массива считываются повторно (и заносятся в переменную txt), но на этот раз используется оператор цикла по коллекции. Описание оператора по коллекции начинается инструкцией for(int s: nums), а в фигурных скобках указана команда txt+=s+" | ". В данном случае переменная s пробегает значения элементов из массива nums (в порядке увеличения индекса элементов). За каждый цикл ткущее значение переменной s дописывается в текст из переменной txt.

После того как текст для сообщения (значение переменной txt) сформирован, с помощью статического метода showMessageDialog() отображается сообщение.

Присваивание массивов

Если переменные массива имеют одинаковый тип, то одной переменной значением может быть присвоена другая переменная.

> **НА ЗАМЕТКУ**
>
> Переменная массива, как известно, ссылается на массив. Тип переменной массива связан с типом элементов массива, на которые

ссылается (или может ссылаться) переменная массива. Переменные массива относятся к одному типу, если предназначены для работы с массивами, состоящими из элементов определенного типа. Размерности массивов также должны совпадать. А вот размер массива значения не имеет. Например, переменная массива `nums`, объявленная инструкцией `int[] nums`, может ссылаться на одномерный массив любого размера из целочисленных элементов.

Чтобы понять последствия такой операции, рассмотрим небольшой пример, представленный в листинге 5.4.

Листинг 5.4. Программный код проекта

```
class AssigningArraysDemo{
  // Статический метод для отображения
  // содержимого массива (аргумент метода):
  static void show(int[] nums){
    // Оператор цикла по коллекции:
    for(int s: nums){
      // Отображение значения элемента массива:
      System.out.print("| "+s+" ");
    }
    System.out.println("|");
  }
  // Главный метод программы:
  public static void main(String[] args){
    // Первый массив:
    int[] odd={1,3,5,7,9};
    // Второй массив:
    int[] even={2,4,8,10,12,14,16};
    System.out.println("Массив odd:");
    // Отображение содержимого первого массива:
    show(odd);
    System.out.println("Массив even:");
    // Отображение содержимого второго массива:
    show(even);
```

```
    System.out.println("Выполняется присваивание.");
    // Присваивание массивов:
    even=odd;
    // Изменение значения элемента массива:
    even[0]=-1;
    System.out.println("Массив odd:");
    // Отображение содержимого первого массива:
    show(odd);
    System.out.println("Массив even:");
    // Отображение содержимого второго массива:
    show(even);
  }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 5.4)

```
Массив odd:
| 1 | 3 | 5 | 7 | 9 |
Массив even:
| 2 | 4 | 8 | 10 | 12 | 14 | 16 |
Выполняется присваивание.
Массив odd:
| -1 | 3 | 5 | 7 | 9 |
Массив even:
| -1 | 3 | 5 | 7 | 9 |
```

В программе (в главном методе) объявляются и инициализируются два целочисленных массива `odd` и `even`. Для начала содержимое обоих массивов отображается в консольном окне. Для отображения содержимого массива использован статический метод `show()`, не возвращающий результат. Аргументом методу передается массив — точнее, переменная массива (команды `show(odd)` и `show(even)`). Аргумент метода, являющийся переменной массива, описывается точно так же, как объявляется

переменная массива: после идентификатора типа элементов массива указываются пустые квадратные скобки.

После отображения содержимого массивов odd и even командой even=odd выполняется присваивание массивов (точнее, присваивание переменных массива), а командой even[0]=-1 изменяется значение первого элемента массива.

Затем снова отображается содержимое массивов odd и even. Что же при этом происходит? Во-первых, следует разобраться с результатом выполнения команды even=odd. Переменные массива, как неоднократно отмечалось ранее, ссылаются на массив. Значение переменной массива удобно представлять как некоторый адрес массива. Поэтому если одной переменной массива значением присваивается другая переменная массива, как в команде even=odd, то первая переменная (в нашем случае even) значением получает адрес массива, содержащийся во второй переменной (здесь это переменная odd). Таким образом, обе переменные массива после присваивания содержат значением адрес одного и того же массива. Следовательно, обе переменные будут ссылаться на один и тот же массив. После выполнения команды even=odd и переменная even, и переменная odd ссылаются на один и тот же массив — массив, на который первоначально ссылалась переменная odd. Когда командой even[0]=-1 формально изменяется значение начального элемента массива even, то на самом деле изменяется элемент в том массиве, на который ссылается и переменная odd. Поэтому когда командами show(odd) и show(even) отображается содержимое массивов odd и even, отображается содержимого одного и того же массива.

Двумерные массивы

Эх, погубят тебя слишком широкие возможности.

Из к/ф «Айболит-66»

В двумерном массиве идентификация элемента выполняется с помощью двух индексов. Его удобно представлять в виде массива, элементами которого являются массивы, или таблицы, состоящей из определенного количества строки и столбцов. Первый индекс элемента массива определяет строку в такой таблице, а второй индекс определяет столбец, на пересечении которых находится элемент.

Создание двумерного массива

Принцип реализации двумерного массива концептуально такой же, как и одномерного массива: объявляется переменная двумерного массива, значением которой присваивается ссылка на двумерный массив. Двумерный массив создается с помощью оператора new. Более конкретно, при объявлении переменной двумерного массива указывается идентификатор типа элементов, две пары пустых квадратных скобок и, собственно, имя переменной массива. При создании двумерного массива указывается оператор new, затем идентификатор типа элементов массива и две пары квадратных скобок, в каждой из которых указывается размер массива по соответствующему индексу. Шаблон создания двумерного массива такой:

```
тип[][] переменная=new тип[размер][размер];
```

Например, ниже приведена команда, которой создается двумерный целочисленный массив из 3 строк и 5 столбцов:

```
int[][] nums=new int[3][5];
```

Данную команду можно «разбить» на две:

```
int[][] nums;
nums=new int[3][5];
```

Командой int[][] nums объявляется переменная nums двумерного массива. При выполнении команды nums=new int[3][5] создается двумерный массив из целых (тип int) из трех строк и пяти столбцов (всего пятнадцать элементов). Ссылка на массив записывается в переменную nums.

При обращении к элементу двумерного массива указывается имя переменной, ссылающейся на массив, и индексы элемента: каждый индекс помещается в отдельные квадратные скобки. Индексация (по каждому из индексов) начинается с нуля. Так, скажем, инструкция nums[0][0] означает обращение к первому элементу в первой строке, а инструкция nums[1][2] является обращением к элементу во второй строке и третьем столбце.

Так же как и для одномерного массива, для двумерного размер определяется с помощью свойства length. Но в двумерном массиве два размера — для каждого из индексов. Если обратиться к свойству length через переменную массива, то результатом получим количество строк

в массиве — то есть размер по первому индексу. Например, если переменная nums объявлена так, как указано выше, то результатом выражения nums.length является значение 3. Если нас интересует количество столбцов в массиве, то после имени массива следует указать (в квадратных скобках) индекс строки в массиве, после чего (через точку) указывается свойство length. Так, значением выражения nums[0].length является количество элементов в первой строке массива nums (то есть значение 5). Поскольку массив содержит в каждой строке одинаковое количество элементов, то в принципе от индекса в команде nums[0].length мало что зависит — например, значение выражения nums[1].length тоже равно 5.

ДЕТАЛИ

Двумерный массив удобно представлять как одномерный (назовем его «внешний») массив, элементами которого являются одномерные (назовем их «внутренними») массивы. Если переменная массив ссылается на двумерный массив, то значением выражения вида массив[индекс] является элемент «внешнего» одномерного массива, то есть «внутренний» массив — или, другими словами, ссылка на строку двумерного массива с соответствующим индексом. Фактически выражение вида массив[индекс] представляет собой одномерный массив, формирующий строку двумерного массива. Поэтому инструкция вида массив[индекс].length значением возвращает количество элементов в таком массиве — то есть количество элементов в строке массива. В рамках этой концепции становится понятен и смысл команды массив.length — это количество элементов во «внешнем» массиве, или количество «внутренних» массивов (количество строк в двумерном массиве).

Небольшой пример, в котором создается, заполняется и отображается двумерный целочисленный массив, представлен в листинге 5.5. В данной программе в консольное окно выводится *таблица умножения*. В первой строке и первом столбце таблицы размещается ряд натуральных чисел от 1 до 9, а на пересечении строк и столбцов находится результат произведения соответствующих чисел в первой строке и первом столбце. Рассмотрим несложный программный код, в котором реализуется задача о формировании таблицы умножения.

Листинг 5.5. Программный код проекта MultiplicationTableApplication

```
class MultiplicationTableDemo{
  public static void main(String[] args){
```

```
    // Размеры массива:
    int rows=9,cols=9;
    // Создание двумерного массива:
    int[][] table=new int[rows][cols];
    // Текстовое значение для отображения в окне вывода:
    String txt="\t|\t";
    for(int i=1;i<=cols;i++){
      txt+=i+"\t";
    }
    txt+="\n";
    for(int i=1;i<=10+8*cols;i++){
      txt+="-";
    }
    // Заполнение массива:
    for(int i=0;i<table.length;i++){
      // Дополнение текстового значения:
      txt+="\n"+(i+1)+"\t|\t";
      for(int j=0;j<table[i].length;j++){
        // Вычисление значения элемента массива:
        table[i][j]=(i+1)*(j+1);
        // Дополнение текстового значения:
        txt+=table[i][j]+"\t";
      }
    }
    // Отображение сообщения в окне вывода:
    System.out.println("Таблица умножения:");
    System.out.println(txt);
  }
}
```

Ниже приведен результат выполнения программы (для удобства количество пробелов между числами уменьшено, а для удобства восприятия первая строка и первый столбец выделены жирным шрифтом):

Результат выполнения программы (из листинга 5.5)

```
Таблица умножения:
  | 1  2  3  4  5  6  7  8  9
-----------------------------------------
1 | 1  2  3  4  5  6  7  8  9
2 | 2  4  6  8  10 12 14 16 18
3 | 3  6  9  12 15 18 21 24 27
4 | 4  8  12 16 20 24 28 32 36
5 | 5  10 15 20 25 30 35 40 45
6 | 6  12 18 24 30 36 42 48 54
7 | 7  14 21 28 35 42 49 56 63
8 | 8  16 24 32 40 48 56 64 72
9 | 9  18 27 36 45 54 63 72 81
```

В программе целочисленными переменными rows и cols определяются размеры двумерного массива, а сам массив создается командой int[] [] table=new int[rows][cols]. В массив заносятся значения попарных произведений чисел (первое число принимает значения в диапазоне от 1 до значения переменной rows, а второе число принимает значения от 1 до значения переменной cols). Также в программе объявляется текстовая переменная txt (с начальным значением "\t|\t"). Запускается оператор цикла, в котором к текстовому значению последовательно дописываются через управляющий символ табуляции \t числовые значения от 1 до значения переменной cols. Так формируется начальная строка с числовыми значениями. После этого она дополняется инструкцией перехода к новой строке \n, а затем с помощью еще одного оператора цикла формируется импровизированная линия из черточек.

ДЕТАЛИ

При отображении текста в консоли в каждой строке есть определенные равноудаленные «метки» — позиции табуляции. Обычно они следуют с интервалом в 8 символов. Поэтому позиции табуляции в строке соответствуют позициям с номерами 1, 9, 17, 25 и так далее. Если при отображении в консоли (окне вывода) текста встречается инструкция выполнения табуляции \t, то курсор автоматически переносится вправо к ближайшей позиции табуляции, и оттуда продолжается вывод данных. В рассматриваемом примере мы

используем символ табуляции \t для выравнивания отображаемых значений в столбцах.
При формировании импровизированной горизонтальной «линии» из черточек количество таких черточек определяется, исходя из количества позиций, которые выделяются на отображение первой строки в таблице. Если количество чисел в первой строке определяется значением переменной cols, то с учетом отступа на первый «декоративный столбец», последнее число будет отображаться в позиции с порядковым номером 9+8*cols. Еще один символ добавляем исходя из эстетических соображений. Отсюда общее количество «черточек» определяется значением выражения 10+8*cols.

Во вложенных операторах цикла индексная переменная i во внешнем операторе пробегает значения от 0 до table.length-1 включительно (индекс строки двумерного массива), а индексная переменная j во внутреннем операторе цикла пробегает значения от 0 до table[i].length-1 включительно (индекс столбца). Значение элементу массива присваивается командой table[i][j]=(i+1)*(j+1). Также при выполнении операторов цикла выполняются дополнительные манипуляции с текстовой переменной txt. После завершения всех вычислений командой System.out.println(txt) сформированное текстовое значение из переменной txt отображается в консольном окне.

Инициализация двумерного массива

Как и одномерный массив, двумерный массив можно инициализировать при создании. В этом случае объявляется переменная двумерного массива и значением этой переменной присваивается список значений для элементов массива. Список представляет собой конструкцию, состоящую из фигурных скобок, внутри которых через запятую указываются списки значений. Каждый внутренний список заключается в фигурные скобки, списки между собой разделяются запятыми, как и значения внутри списков. Ниже приведен пример объявления и инициализации двумерного символьного массива из 2 строк и 3 столбцов:

```
char[][] symbs={{'A','B','C'},{'D','E','E'}};
```

В данном случае значения элементов массива symbs в первой строке равны 'A', 'B' и 'C', а во второй строке значения элементов массива равны 'D', 'E' и 'E'.

Небольшой пример, в котором использована инициализация двумерных массивов, представлен в листинге 5.6.

Листинг 5.6. Программный код проекта Show2DArrayApplication

```
class Show2DArrayDemo{
  // Статический метод для отображения
  // двумерного массива:
  static void show(int[][] nums){
    for(int i=0;i<nums.length;i++){
      for(int j=0;j<nums[i].length;j++){
        // Отображение значения элемента массива:
        System.out.print(nums[i][j]+" ");
      }
      // Переход к новой строке:
      System.out.println("");
    }
  }
  // Главный метод программы:
  public static void main(String[] args){
    // Массив из двух строк и трех столбцов:
    int[][] alpha={{1,2,3},{4,5,6}};
    // Массив со строками разной длины:
    int[][] bravo={{1,2,3},{4,5},{6,7,8,9}};
    // Отображение содержимого первого массива:
    System.out.println("Массив alpha:");
    show(alpha);
    // Отображение содержимого второго массива:
    System.out.println("Массив bravo:");
    show(bravo);
  }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 5.6)

```
Массив alpha:
1 2 3
```

```
4 5 6
Массив bravo:
1 2 3
4 5
6 7 8 9
```

Программный код простой, но в нем есть несколько моментов, на которые стоит обратить внимание. Во-первых, в программе описан статический метод show(), предназначенный для отображения в консольном окне (окне вывода) содержимого двумерного целочисленного массива. Аргументом методу передается переменная массива. Аргумент описан как int[][] nums — то есть, аргумент описывается так же, как объявляется переменная двумерного массива. Команды в теле метода достаточно простые: запускаются вложенные операторы цикла, с помощью которых выполняется построчный вывод значений элементов массива в консольное окно (окно вывода).

В главном методе программы командами int[][] alpha={{1,2,3},{4,5,6}} и int[][] bravo={{1,2,3},{4,5},{6,7,8,9}} объявляются и инициализируются два двумерных целочисленных массива. Командой int[][] alpha={{1,2,3},{4,5,6}} создается двумерный массив alpha из двух строк и трех столбцов. Значения элементов в первой строке массива равны 1, 2 и 3, а значения элементов во второй строке равны 4, 5 и 6. С помощью команды show(alpha) содержимое массива выводится в окно вывода. Командой int[][] bravo={{1,2,3},{4,5},{6,7,8,9}} создается двумерный массив bravo, состоящий из трех строк. Но количество элементов в каждой строке различно. Первая строка стоит из трех элементов со значениями 1, 2 и 3. Вторая строка состоит из двух элементов со значениями 4 и 5. Третья строка состоит из четырех элементов со значениями 6, 7, 8 и 9. Содержимое массива отображается с помощью команды show(bravo). Результат выполнения команды подтверждает, что массив bravo содержит в строках разное количество элементов. Как такое возможно? Объяснение простое. Мы знаем, что двумерный массив является одномерным массивом («внешний» массив), элементами которого являются одномерные массивы («внутренние» массивы). Но технически это все реализуется так: элементами «внешнего» являются переменные одномерного массива, которые ссылаются на «внутренние» одномерные массивы. Поскольку переменная одномерного массива может ссылаться на массив любого размера, то становится понятным, что количество элементов в строках двумерного массива в принципе может быть разным для каждой строки. Далее мы эту ситуацию рассмотрим более подробно.

Массив со строками разной длины

Выше мы создали двумерный массив со строками разной длины. При этом мы инициализировали массив при создании. Но процесс создания такого массива можно реализовать и несколько иначе. Как пример в листинге 5.7 приведена программа, в которой создается двумерный символьный массив со строками разной длины. Массив заполняется символами в алфавитном порядке. Количество элементов в разных строках определяется значениями элементов в целочисленном массиве. Рассмотрим программный код примера.

Листинг 5.7. Программный код проекта Using2DCharArrayApplication

```
import javax.swing.*;
class Using2DCharArrayDemo{
  public static void main(String[] args){
    // Массив содержит значения, определяющие
    // длину строк символьного массива:
    int[] size={3,12,4,7};
    // Создается двумерный символьный массив,
    // в котором определено количество строк,
    // но не задано количество столбцов:
    char[][] symbs=new char[size.length][];
    // Символьная переменная с начальным символом
    // для заполнения массива:
    char s='A';
    // Текстовая переменная для формирования текста,
    // отображаемого в диалоговом окне:
    String txt="";
    // Создание строк массива и присваивание
    // значений элементам массива:
    for(int i=0;i<symbs.length;i++){
      // Создание строки массива:
      symbs[i]=new char[size[i]];
      // Заполнение строки значениями:
      for(int j=0;j<symbs[i].length;j++){
```

```
        // Элементу массива присваивается значение:
        symbs[i][j]=s;
        // Следующий символ:
        s++;
        // К текстовому значению дописывается
        // значение элемента массива:
        txt+="| "+symbs[i][j]+" ";
      }
      // Последняя черта и переход к новой строке:
      txt+="|\n";
    }
    // Отображение сообщения:
    JOptionPane.showMessageDialog(null,
      // Текст сообщения:
      txt,
      // Название окна:
      "Массив со строками разной длины",
      // Тип окна:
      JOptionPane.PLAIN_MESSAGE,
      // Объект пиктограммы:
      new ImageIcon("d:/books/pictures/giraffe.png")
    );
  }
}
```

При выполнении программы отображается диалоговое окно, представленное на рис. 5.7.

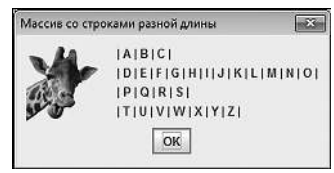


Рис. 5.7. Диалоговое окно содержит значения массива со строками разной длины

Далее проанализируем программный код, выполнение которого приводит к таким результатам. Итак, в программе объявляется и инициализируется одномерный целочисленный массив size. Значения элементов массива определяют количество элементов в строках двумерного символьного массива. Таким образом, количество элементов в массиве size определяет количество строк в двумерном символьном массиве. Сам двумерный символьный массив создается командой char[][] symbs=new char[size.length][]. Особенность данной команды в том, что в инструкции создания массива new char[size.length][] вторые квадратные скобки пустые. В первых квадратных скобках указано выражение size.length. Это означает, что создается массив, количество элементов в котором равно size.length. Элементы массива являются переменными одномерного массива с элементами типа char.

ДЕТАЛИ

Команду создания двумерного массива вида new тип[размер][] можно рассматривать как команду создания одномерного массива, количество элементов в котором определяется параметром размер, а тип элементов определяется как тип[] — то есть речь идет о переменных массива, которые могут ссылаться на одномерные массивы с элементами, имеющими указанный тип.

Символьная переменная s получает в качестве начального значение 'A'. Такое значение будет присвоено первому элементу в первой строке двумерного массива symbs. Каждый следующий элемент значением получает очередную букву в кодовой таблице (следующая буква в алфавите). Текстовая переменная txt инициализируется с пустой строкой в качестве начального значения. В это текстовое значение последовательно дописываются новые «фрагменты». Все основные вычисления производятся в рамках вложенных операторов цикла. Во внешнем операторе индексная переменная i перебирает строки двумерного массива. В теле внешнего оператора командой symbs[i]=new char[size[i]] создается одномерный символьный массив, размер которого определяется значением элемента size[i]. Ссылка на созданный массив записывается в элемент symbs[i]. После того, как строка двумерного массива создана, запускается еще один (внутренний) оператор цикла, в котором индексная переменная j перебирает значения элементов в строке с индексом i. В теле внутреннего оператора цикла командой symbs[i][j]=s присваивается значение элементу с индексами i и j двумерного массива symbs. Командой s++ значение переменной s увеличивается на единицу (значением переменной становится

следующий символ после текущего). Наконец, командой txt+="| "+symbs[i][j]+" " к текущему значению текстовой переменной txt дописывается «декоративная» вертикальная черта, пробел, значение элемента symbs[i][j] и еще один пробел. После того, как строка массива symbs «пройдена», после завершения внутреннего оператора, командой txt+="|\n" к тексту добавляется последняя «завершающая» вертикальная черта и инструкция перехода к новой строке.

После того как все вычисления выполнены, отображается сообщение с информацией о содержимом двумерного массива symbs.

> **НА ЗАМЕТКУ**
>
> Команды отображения диалогового окна мы уже использовали многократно, так что хочется верить, что в комментариях они не нуждаются.

Массивы и методы

Ну, а это довесок к кошмару.

Из к/ф «Старики-разбойники»

Далее обсудим еще два аспекта работы с массивами: передачу массива аргументом методу и возвращение массива результатом метода. В некоторых примерах мы уже встречались с ситуацией, когда массив передается аргументом методу. Делается это просто: соответствующий аргумент описывается как переменная массива. Значением переменной массива является ссылка на массив. Когда массив передается аргументом методу, метод получает ссылку на массив. В основном это вся «философия», связанная с передачей массива аргументом методу.

Примерно такая же схема используется при возвращении методом массива в качестве результата. В этом случае обычно массив, возвращаемый результатом, создается в теле метода при его выполнении. Результатом метод возвращает ссылку на созданный массив.

Как иллюстрацию рассмотрим математическую задачу о вычислении произведения матрицы на вектор, результатом которого является другой вектор. Векторы реализуются в виде одномерных числовых массивов, а матрица реализуется в виде двумерного числового массива.

ДЕТАЛИ

Матрица представляет собой таблицу из чисел. Каждый элемент матрицы определяется парой индексов — то есть все как в двумерном массиве. Массив представляет собой набор элементов, каждый из которых определяется индексом. Векторы мы будем отождествлять с одномерным массивом. Предположим, что задана некоторая матрица A с элементами a_{ij}, индексы которых $1 \le i \le m$ и $1 \le j \le n$. Пускай вектор B состоит из элементов b_j (индекс $j = 1, 2, \ldots, n$). Тогда результатом произведения матрицы A на вектор B является вектор $C = AB$, элементы c_i (индекс $i = 1, 2, \ldots, m$) которого вычисляется в виде суммы $c_i = \sum_{j=1}^{n} a_{ij} b_j$. Стоит заметить, что количество столбцов в матрице A должно совпадать с количеством элементов в векторе B.

Рассмотрим программный код в листинге 5.8.

Листинг 5.8. Программный код проекта ArraysAndMethodsApplication

```
class ArraysAndMethodsDemo{
  // Метод для отображения содержимого
  // одномерного целочисленного массива:
  static void show1D(int[] nums){
    // Оператор цикла по коллекции:
    for(int s: nums){
      // Форматированный вывод числового значения:
      System.out.printf("%4d",s);
    }
    // Переход к новой строке:
    System.out.println("");
  }
  // Метод для отображения содержимого двумерного
  // целочисленного массива:
  static void show2D(int[][] nums){
    // Внешний оператор цикла по коллекции:
    for(int[] p: nums){
      // Внутренний оператор цикла по коллекции:
      for(int s: p){
```

```
        // Форматированный вывод числового значения:
        System.out.printf("%4d",s);
      }
      // Переход к новой строке:
      System.out.println("");
    }
  }
  // Метод для вычисления произведения матрицы и вектора.
  // Аргументами методу передаются двумерный
  // и одномерный массивы. Результатом возвращается
  // одномерный массив:
  static int[] prod(int[][] A,int[] B){
    // Создание одномерного массива:
    int[] C=new int[A.length];
    // Вычисление значений элементов массива — результата
    // произведения матрицы и вектора:
    for(int i=0;i<C.length;i++){
      // Начальное нулевое значение элемента:
      C[i]=0;
      for(int j=0;j<B.length;j++){
        // Добавление очередного слагаемого:
        C[i]+=A[i][j]*B[j];
      }
    }
    // Результат метода:
    return C;
  }
  // Главный метод программы:
  public static void main(String[] args){
    // Двумерный массив (матрица):
    int[][] A={{1,0,3,-1},{2,-1,-2,3},{-1,1,0,2}};
    // Одномерный массив (вектор):
```

```
    int[] B={1,-1,3,2};
    // Результат произведения матрицы на вектор:
    int[] C=prod(A,B);
    // Отображение содержимого матрицы:
    System.out.println("Матрица A:");
    show2D(A);
    // Отображение содержимого вектора:
    System.out.println("Вектор B:");
    show1D(B);
    // Отображение результат произведения
    // матрицы на вектор:
    System.out.println("Вектор C=AB:");
    show1D(C);
  }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 5.8)

```
Матрица A:
  1  0  3 -1
  2 -1 -2  3
 -1  1  0  2
Вектор B:
  1 -1  3  2
Вектор C=AB:
  8  3  2
```

В программе, кроме главного метода программы, описаны еще три метода. Метод `show1D()` предназначен для отображения содержимого одномерного целочисленного массива, который передается аргументом методу. Для отображения значений переданного аргументом массива `nums` используется оператор цикла по коллекции, в котором переменная `s` последовательно принимает значения массива `nums`. Отображение

значения выполняется командой System.out.printf("%4d",s). В данной команде вызывается метод printf(). Второй аргумент метода s — это значение, которое отображается в окне вывода. Первый аргумент "%4d" метода является строкой форматирования. Символ % является стандартным началом инструкции форматирования. Литера d означает, что отображается целое число. Цифра 4 означает, что под число выделяется не меньше четырех позиций.

Статический метод show2D() используется для отображения содержимого двумерного целочисленного массива. В теле оператора цикла использованы вложенные циклы по коллекции. Во внешнем операторе цикла перебор значений в массиве nums (аргумент метода) реализуется через переменную p, объявленную с типом int[]. Этот «тип» соответствует переменной одномерного массива. Переменная p последовательно принимает значения из массива nums, но при этом данный массив интерпретируется как одномерный массив, состоящий из одномерных массивов (строки двумерного массива). Таким образом, значения переменной p — это одномерные массивы (ссылки на эти массивы), формирующие строки в массиве nums.

Во внутреннем операторе цикла по коллекции переменная s типа int перебирает элементы из «коллекции» p, которая, напомним, является очередной строкой в массиве nums. Вывод такой: переменная p «перебирает» строки в массиве nums, а переменная s «перебирает» элементы в строке.

Метод prod() нужен для вычисления произведения матрицы и вектора. Аргументами методу передается двумерный массив A и одномерный массив B. Результатом метод возвращает ссылку на одномерный целочисленный массив. Поэтому идентификатор типа результата метода обозначен как int[]. В теле метода командой int[] C=new int[A.length] создается одномерный массив C, размер которого определяется количеством строк в двумерном массиве A. Запускается оператор цикла, в котором индексная переменная i пробегает значения индексов элементов массива C. За каждый цикл командой C[i]=0 элементу с соответствующим индексом присваивается начальное нулевое значение. После этого запускается внутренний оператор цикла, в котором индексная переменная j пробегает значения от 0 до B.length-1 (диапазон изменения индексов элементов в массиве B и одновременно это диапазон изменения второго индекса в двумерном массиве A). За каждый цикл командой C[i]+=A[i][j]*B[j] к текущему значению элемента C[i] прибавляется очередная добавка (произведение A[i][j]*B[j]). После завершения вычислений командой return C ссылка на созданный и заполненный массив возвращается результатом метода.

В главном методе программы командами int[][] A={{1,0,3,-1},{2,-1,-2,3},{-1,1,0,2}} и int[] B={1,-1,3,2} создаются исходные массивы: двумерный массив A из трех строк и четырех столбцов и одномерный массив B из четырех элементов. Командой int[] C=prod(A,B) вычисляется произведение матрицы, представленной двумерным массивом A, и вектора, представленного одномерным массивом B, а результат произведения записывается в одномерный массив C. С помощью методов show2D() и show1D() содержимое исходных массивов и массива, вычисленного как произведение, отображается в окне вывода.

Резюме

У правды нет предела.

Из к/ф «Старики-разбойники»

- Массив представляет собой набор однотипных элементов, объединенных общим именем. Элемент в массиве идентифицируется с помощью индекса или индексов. Количество индексов, необходимых для однозначного определения массива, определяет размерность массива. Обычно используются одномерные и двумерные массивы.
- Массив создается в два этапа: объявляется переменная массива, а значением ей присваивается ссылка на массив. Ссылку на массив получаем при собственно создании массива с помощью оператора new.
- При объявлении переменной для одномерного массива указывается идентификатор типа элементов массива и пара пустых квадратных скобок. В случае переменной двумерного массива указывают две пары квадратных скобок.
- При создании массива используют оператор new, после которого указывается тип элементов массива и в квадратных скобках — размер по каждому из индексов.
- При обращении к элементу массива указывают имя массива и в квадратных скобках — индекс массива. Если индексов больше одного, то для каждого индекса используется своя пара квадратных скобок. Индексация элементов массива (по каждому из индексов) начинается с нуля. Для определения размера массива используют свойство length.
- Двумерный массив можно рассматривать как одномерный массив, элементами которого являются одномерные массивы.

- При создании массивы можно инициализировать. В таком случае переменной массива присваивается список со значениями, которые присваиваются элементам массива. Значения заключаются в фигурные скобки. Если массив двумерный, то присваиваемый переменной массива список состоит из списков, построчно инициализирующих элементы массива.
- При присваивании массивов одна переменная массива получает значением ссылку из другой переменной массива, так что в результате обе переменные ссылаются на один и тот же массив.
- Оператор цикла по коллекции позволяет перебирать не индексы элементов массива, а значения элементов. Описание оператора цикла начинается с ключевого слова for, а в круглых скобках объявляется переменная типа, совпадающего с типом элементов массива. Через двоеточие указывается имя массива. Объявленная переменная последовательно принимает значения элементов массива, а при каждом значении выполняются команды в теле оператора цикла.
- При передаче массива аргументом методу на самом деле передается переменная массива. Если результатом метода возвращается массив, то такой массив создается при вызове метода, а результатом метода возвращается ссылка на массив.

Глава 6
НАСЛЕДОВАНИЕ

Вот так начнешь изучать фамильные портреты, и уверуешь в переселение душ. Он, оказывается, тоже Баскервиль!

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Далее мы обсудим очень мощный механизм, который называется *наследованием*. Идея, положенная в основу наследования, проста: при создании (описании) класса за основу берется другой, ранее созданный (описанный) класс. Такой подход позволяет экономить время, силы, способствует уменьшению объема программного кода, а также повышает совместимость программных кодов.

Реализация наследования

Ну, зачем такие сложности?!

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Итак, при наследовании новый класс создается на основе уже существующего класса. Класс, на основе которого создается новый класс, называется *суперклассом*. Новый класс, который создается на основе суперкласса, называется *подклассом*.

Главное преимущество создания класса «не на пустом месте» связано с тем, что подкласс получает «в наследство» от суперкласса все его незакрытые поля и методы. Проще говоря, все поля и методы суперкласса, не являющиеся закрытыми, автоматически добавляются в подкласс (хотя там они явно и не описываются).

НА ЗАМЕТКУ

Закрытые поля и методы описываются с ключевым словом private. Если такие члены есть в суперклассе, то в подклассе, как отмечалось

выше, они не наследуются (в том смысле, что недоступны). Более подробно особенности использования закрытых членов класса при наследовании обсуждаются в этой главе несколько позже.

Создание подкласса

С технической точки зрения создать подкласс на основе суперкласса очень просто: в описании подкласса после его имени через ключевое слово extends следует указать имя суперкласса. То есть используется следующий шаблон (жирным шрифтом выделены ключевые элементы шаблона):

```
class Подкласс extends Суперкласс{
  // Поля и методы подкласса
}
```

Например, если класс Bravo создается наследованием класса Alpha, то описание класса Bravo выполняется кодом следующего вида:

```
class Bravo extends Alpha{
  // Дополнительные (кроме членов класса Alpha)
  // поля и методы класса Bravo
}
```

В результат класс Bravo имеет такие же поля и методы, как класс Alpha, а также дополнительно те поля и методы, что описаны непосредственно в классе Bravo.

Если рассмотреть более конкретную ситуацию, то она могла бы выглядеть так:

```
// Описание суперкласса:
class Alpha{
  // Целочисленное поле:
  int number;
  // Метод:
  void show(){
    System.out.println("Числовое поле: "+number);
  }
}
```

```
// Описание подкласса:
class Bravo extends Alpha{
  // Символьное поле:
  char symbol;
}
```

В данном случае в классе Alpha описано целочисленное поле number и метод show(), которым в консольное окно выводится сообщение со значением поля number. Соответственно, у объекта, созданного на основе класса Alpha, будет в наличии поле number и метод show().

Непосредственно в классе Bravo описано только символьное поле symbol. Но поскольку класс Bravo создается на основе класса Alpha (класс Alpha является суперклассом для класса Bravo), то у объектов класса Bravo будут, кроме поля symbol, еще и поле number, а также метод show().

НА ЗАМЕТКУ

Связь наследования устанавливается для классов. Объекты, созданные на основе суперкласса и подкласса совершенно независимы. Более того, понятие суперкласса является относительным: некоторый класс может быть суперклассом для другого класса и в то же время сам являться подклассом другого класса. Такой принцип наследования (когда подкласс сам является суперклассом для другого класса) называется *многократным наследованием*.

Суперкласс может быть как пользовательским (описан непосредственно в программе), так и библиотечным. Например, мы для отображения сообщений в диалоговых окнах и считывания значений с помощью окна с полем ввода достаточно часто использовали статические методы класса JOptionPane. При этом нам нередко приходилось указывать несколько довольно громоздких аргументов. Если указанные методы вызываются достаточно часто, а часть аргументов, которые передаются методам при разных вызовах, одни и те же, то удобно на основе класса JOptionPane путем наследования создать собственный класс, в котором определить новые методы (в которых тем не менее используются утилиты класса JOptionPane). Как иллюстрацию использования механизма наследования рассмотрим небольшой пример, в котором в пользовательском классе MyPane наследуется библиотечный класс JOptionPane. Интересующий нас программный код представлен в листинге 6.1.

Листинг 6.1. Программный код проекта ExtendingJOptionPaneApplication

```
import javax.swing.*;
// Класс создается наследование класса JOptionPane:
class MyPane extends JOptionPane{
  // Статический метод с двумя аргументами
  // для отображения диалогового окна:
  static void showMessage(String txt,String title){
    // Вызов статического метода showMessageDialog() из
    // класса JOptionPane:
    showMessageDialog(null,txt,title,PLAIN_MESSAGE, new ImageIcon("d:/books/pictures/
giraffe.png"));
  }
  // Статический метод с одним аргументом
  // для отображения диалогового окна:
  static void showMessage(String txt){
    // Вызов версии метода с двумя аргументами:
    showMessage(txt,"Сообщение");
  }
  // Статический метод для отображения окна с полем ввода
  // для считывания целого числа:
  static int getInteger(String txt){
    // Текстовая переменная:
    String res;
    // Отображение окна с полем ввода с помощью
    // метода showInputDialog() из класса JOptionPane:
    res=showInputDialog(null,txt,"Число (по умолчанию  10)",QUESTION_MESSAGE);
    // Проверяется значение текстовой переменной:
    if(res==null){
      // Если пользователь отменил ввод числа
      // (значение, возвращаемое по умолчанию):
      return 10;
    }
```

```
    else{
      // Преобразование текстового представления числа
      // в число:
      return Integer.parseInt(res);
    }
  }
}
// Класс с главным методом программы:
class ExtendingJOptionPaneDemo{
  public static void main(String[] args){
    // Отображение диалогового окна с сообщением:
    MyPane.showMessage("Всем привет!");
    // Переменная для записи числового значения:
    int number;
    // Отображение диалогового окна с полем ввода
    // и считывание целочисленного значения:
    number=MyPane.getInteger("Введите целое число");
    // Текст для отображения в диалоговом окне:
    String txt="Числа от 1 до "+number+":\n";
    // Формирование текстовой строки
    // с последовательностью натуральных чисел:
    for(int k=1;k<=number;k++){
      txt+=k+" ";
    }
    // Отображение диалогового окна с сообщением:
    MyPane.showMessage(txt,"Целые числа");
  }
}
```

В программе описывается класс MyPane, являющийся подклассом класса JOptionPane. В результате в классе MyPane наследуются все открытые члены класса JOptionPane, включая (но не ограничиваясь) статические методы showMessageDialog() и showInputDialog(), а также статические константы,

определяющие тип пиктограмм, используемых в диалоговых окнах. Указанные методы и константы становятся членами класса MyPane. Поэтому при обращении к ним в теле класса MyPane нет необходимости указывать имя класса JOptionPane.

Для удобства в теле класса MyPane описаны две версии статического метода showMessage(): с двумя текстовыми аргументами и с одним текстовым аргументом. Если метод вызывается с двумя текстовыми аргументами, то первый аргумент определяет текстовое сообщение, отображаемое в диалоговом окне, а второй аргумент задает название диалогового окна. Если метод вызывается с одним аргументом, то он определяет текст сообщения. Окно при этом имеет название **Сообщение**.

Версия метода showMessage() реализована так, что при вызове этого метода на самом деле вызывается метод showMessageDialog(). Первым аргументом данному методу передается пустая ссылка null, второй и третий аргументы определяются, соответственно, аргументами метода showMessage(), четвертый аргумент является статической константой PLAIN_MESSAGE (хотя в данном случае этот аргумент фактически ни на что не влияет, поскольку задан пятый аргумент), а пятым аргументом указано выражение new ImageIcon("d:/books/pictures/giraffe.png")). Это команда создания объекта пиктограммы (объект класса ImageIcon) на основе изображения в файле giraffe.png, находящемся в каталоге d:\books\pictures. Но в отличие от рассмотренных ранее случаев, здесь результат инструкции создания объекта не записывается в объектную переменную, а сразу передается аргументом методу. Так можно поступать, и данный прием вполне «законный». Чтобы понять, почему так, достаточно вспомнить что результатом команды создания объекта является ссылка на объект. Эта ссылка, собственно, и указана аргументом метода. По большому счету то же самое происходит, если мы аргументом методу передаем объектную переменную, значением которой является ссылка на объект.

НА ЗАМЕТКУ

Для корректного выполнения программы предварительно создается файл giraffe.png (с прозрачной подложкой) с графическим изображением размерами в 75 пикселов в ширину и высоту. Файл помещается в каталог d:\books\pictures.

Версия метода showMessage() с одним текстовым аргументом определена таким образом, что вызывается версия этого же метода с двумя аргументами, и вторым аргументом указан текст "Сообщение".

Еще в классе MyPane мы описываем статический метод getInteger() для считывания целочисленных значений. При вызове метода отображается диалоговое окно, в которое пользователь должен ввести целое число. Веденное пользователем число возвращается результатом метода (именно число, а не его текстовое представление!). У метода один текстовый аргумент, определяющий надпись над полем ввода.

В теле метода объявляется текстовая переменная res, значение которой определяется при вызове метода showInputDialog() (унаследован в классе MyPane из класса JOptionPane). Аргументами методу showInputDialog() при вызове передаются: пустая ссылка null, собственно аргумент метода getInteger(), текст "Число (по умолчанию 10)" (название окна), и статическая константа QUESTION_MESSAGE, определяющая тип пиктограммы в окне. Затем, после того как переменная res получает значение, оно проверяется с помощью условного оператора. Если значение переменной равно null (пользователь отменил ввод, щелкнув кнопку **Cancel** или системную пиктограмму закрытия окна), командой return 10 значение 10 возвращается результатом метода getInteger(). В противном случае командой return Integer.parseInt(res) результатом метода возвращается число, получающееся преобразованием в целочисленный тип текстового представления числа из переменной res.

НА ЗАМЕТКУ

Если пользователь ввел некорректное значение, при попытке преобразовать текстовое значение числа в число может возникнуть ошибка. Как мы узнаем несколько позже, такие ситуации можно отслеживать и обрабатывать с помощью механизма *обработки исключительных ситуаций*.

В главном методе программы командой MyPane.showMessage("Всем привет!") отображается диалоговое окно с приветствием. Командой number=MyPane.getInteger("Введите целое число") отображается окно с полем ввода, а введенное пользователем целое число (результат метода getInteger()) записывается в переменную number. Затем с помощью оператора цикла формируется текст, содержащий последовательность натуральных чисел от 1 до значения переменной number. Результат записывается в текстовую переменную txt. Наконец, командой MyPane.showMessage(txt,"Целые числа") отображается окно с соответствующим сообщением.

В начале выполнения программы появляется диалоговое окно (с названием **Сообщение**) с сообщением, как показано на рис. 6.1.

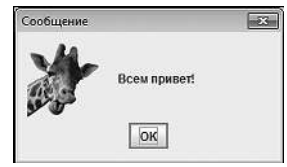


Рис. 6.1. Окно с приветствием отображается в начале выполнения программы

После того как пользователь закрывает первое окно, появляется следующее окно, но уже с полем ввода, в которое следует ввести целое число. На рис. 6.2 показано окно с полем, в которое введено значение 12.

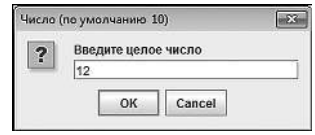


Рис. 6.2. Окно с полем ввода для считывания целочисленного значения

После щелчка на кнопке **ОК** появляется диалоговое окно (называется **Целые числа**), содержащее в основной части окна последовательность натуральных чисел (от 1 до того числа, что было введено на предыдущем этапе) — на рис. 6.3 показано окно, в котором отображается последовательность натуральных чисел от 1 до 12.

Рис. 6.3. Окно с последовательностью натуральных чисел

Если в окне с полем ввода (см. рис. 6.2) щелкнуть кнопку **Cancel**, или закрыть окно, щелкнув системную пиктограмму с крестиком в правом верхнем углу окна, то эффект будет таким, как если бы пользователь ввел число 10: в появившемся диалоговом окне **Целые числа** будет содержаться последовательность натуральных чисел от 1 до 10.

НА ЗАМЕТКУ

В некоторых случаях бывает удобно наследовать в главном классе (класс с главным методом) какой-то другой класс — например,

класс JOptionPane. В таком случае в главном методе программы можно было бы использовать статические методы класса JOptionPane без ссылки на этот класс.

Конструктор подкласса

Существуют специальные правила описания конструктора подкласса. Дело в том, что при создании объекта подкласса сначала вызывается конструктор суперкласса. Суть проблемы легко представить, если предположить, что конструктору суперкласса необходимо передать аргументы. Тогда при создании объекта подкласса необходимо как-то эти аргументы конструктору суперкласса передать. Такой механизм существует, и состоит он в том, что при описании конструктора подкласса первой командой в теле конструктора подкласса вызывается конструктор суперкласса. Делается это достаточно просто: указывается ключевое слово super, после которого в круглых скобках указываются аргументы, которые передаются конструктору суперкласса. Если аргументов нет, круглые скобки (пустые) все равно указываются.

Рассмотрим небольшой пример в листинге 6.2, в котором используется наследование и, кроме прочего, определяется конструктор производного класса.

Листинг 6.2. Программный код проекта SubclassConstructorApplication

```
// Суперкласс:
class Alpha{
  // Текстовое поле:
  String name;
  // Целочисленное поле:
  int code;
  // Конструктор с тремя аргументами:
  Alpha(String name,int code){
    this.name=name;
    this.code=code;
    System.out.println("Класс Alpha:");
    System.out.println("Поле name — "+this.name);
    System.out.println("Поле code — "+this.code);
```

```
  }
  // Конструктор с одним текстовым аргументом:
  Alpha(String name){
    // Вызов конструктора с двумя аргументами:
    this(name,0);
  }
  // Конструктор с одним целочисленным аргументом:
  Alpha(int code){
    // Вызов конструктора с двумя аргументами:
    this("Белый",code);
  }
  // Конструктор без аргументов:
  Alpha(){
    this("Желтый",1);
  }
}
// Подкласс:
class Bravo extends Alpha{
  // Символьное поле:
  char symbol;
  // Закрытый метод для отображения значения
  // символьного поля:
  private void show(){
    System.out.println("Класс Bravo:");
    // Отображение значения символьного поля:
    System.out.println("Поле symbol — "+this.symbol);
    // Отображение "горизонтальной линии":
    for(int k=1;k<=18;k++){
      System.out.print("-");
    }
    System.out.println("");
  }
```

```
// Конструктор с тремя аргументами:
Bravo(String name,int code,char symbol){
  // Вызов конструктора суперкласса
  // с двумя аргументами:
  super(name,code);
  // Присваивание символьному полю значения:
  this.symbol=symbol;
  // Вызов закрытого метода:
  show();
}
// Конструктор с одним символьным аргументом:
Bravo(char symbol){
  // Вызов конструктора суперкласса без аргументов:
  super();
  // Присваивание символьному полю значения:
  this.symbol=symbol;
  // Вызов закрытого метода:
  show();
}
// Конструктор с одним текстовым аргументом:
Bravo(String name){
  // Вызов конструктора суперкласса
  // с одним текстовым аргументом:
  super(name);
  // Присваивание символьному полю значения:
  this.symbol='A';
  // Вызов закрытого метода:
  show();
}
// Конструктор с одним целочисленным аргументом:
Bravo(int code){
  // Вызов конструктора суперкласса
```

```
    // с одним целочисленным аргументом:
    super(code);
    // Присваивание символьному полю значения:
    this.symbol='B';
    // Вызов закрытого метода:
    show();
  }
  // Конструктор без аргументов:
  Bravo(){
    // Сначала неявно вызывается конструктор
    // суперкласса без аргументов.
    // Присваивание значения символьному полю:
    this.symbol='C';
    // Вызов закрытого метода:
    show();
  }
  // Конструктор с двумя аргументами:
  Bravo(String name,int code){
    // Вызов конструктора подкласса с тремя аргументами:
    this(name,code,'D');
  }
}
// Класс с главным методом программы:
class SubclassConstructorDemo{
  public static void main(String[] args){
    // Объектная переменная подкласса:
    Bravo obj;
    // Розные способы создания объекта подкласса:
    obj=new Bravo();
    obj=new Bravo("Красный");
    obj=new Bravo(100);
    obj=new Bravo("Зеленый",200);
```

```
    obj=new Bravo('Y');
    obj=new Bravo("Синий",300,'Z');
  }
}
```

Результат выполнения программы будет таким, как показано ниже:

Результат выполнения программы (из листинга 6.2)

```
Класс Alpha:
Поле name — Желтый
Поле code — 1
Класс Bravo:
Поле symbol — C
------------------
Класс Alpha:
Поле name — Красный
Поле code — 0
Класс Bravo:
Поле symbol — A
------------------
Класс Alpha:
Поле name — Белый
Поле code — 100
Класс Bravo:
Поле symbol — B
------------------
Класс Alpha:
Поле name — Зеленый
Поле code — 200
Класс Bravo:
Поле symbol — D
------------------
Класс Alpha:
```

```
Поле name — Желтый
Поле code — 1
Класс Bravo:
Поле symbol — Y
------------------
Класс Alpha:
Поле name — Синий
Поле code — 300
Класс Bravo:
Поле symbol — Z
------------------
```

В этой программе мы встречаемся с несколькими новыми «моментами», которые имеет смысл обсудить. В первую очередь проанализируем общую структуру программы. Она несложная. В частности, описывается класс Alpha с двумя полями: текстовым полем name и целочисленным полем code. У класса Alpha есть четыре конструктора: без аргументов, с одним текстовым аргументом, с одним целочисленным аргументом и, наконец, с двумя аргументами (текстовым и целочисленным). Класс Bravo создается на основе класса Alpha путем наследования. Поскольку в классе Alpha поля name и code описаны как открытые, то они наследуются в классе Bravo. Помимо этого в классе Bravo описано открытое символьное поле symbol. Закрытый метод show() класса Bravo предназначен для отображения значения данного поля. Метод show() вызывается в конструкторах класса Bravo, которых несколько: без аргументов, с одним символьным аргументом, с одним текстовым аргументом, с одним целочисленным аргументом, с двумя аргументами (текстовым и целочисленным), с тремя аргументами (текстовым, целочисленным и символьным). Все шесть конструкторов класса Bravo задействованы в главном методе программы при создании объектов производного класса.

Проанализируем особенности описания класса Alpha. Здесь встречается ключевое слово this. Данное ключевое слово может использоваться как ссылка на объект, из которого вызывается метод (если используется в конструкторе, то является ссылкой на создаваемый объект), а также при вызове в теле одной версии конструктора другой версии конструктора. Так, в описании конструктора класса Alpha с двумя аргументами, текстовый аргумент конструктора называется name, а целочисленный

аргумент конструктора называется code. Проблема в том, что названия аргументов конструктора совпадают с названиями полей класса. Если в такой ситуации использовать в теле класса name или code, то это будет обращение к аргументам конструктора. Причина в том, что аргументы методов, в том числе и конструкторов, имеют «статус» локальных переменных. Если название локальной переменной совпадает с названием поля, то по умолчанию соответствующий идентификатор в теле метода (или конструктора) обозначает локальную переменную. Чтобы получить доступ к полю, необходимо выполнять полную ссылку на поле, указанием объекта (из которого, как подразумевается, будет вызываться метод). Ключевое слово this, как отмечалось выше, и обозначает объект, из которого вызывается метод. Поэтому в теле конструктора класса Alpha инструкции this.name и this.code обозначают поля объекта (который создается), а инструкции name и code обозначают аргументы конструктора.

Кроме версии конструктора с двумя аргументами, в классе Alpha есть еще три версии конструктора. Код каждой из этих версий определяется через вызов версии конструктора с двумя аргументами, но аргументы передаются по-разному. Например, в версии конструктора с одним текстовым аргументом name в теле конструктора есть всего одна команда this(name,0). Она означает, что должен выполняться код конструктора с двумя аргументами, причем первым аргументом передается переменная name, а вторым аргументом передается значение 0. Далее, в версии конструктора с одним целочисленным аргументом code выполняется команда this("Белый",code), которой вызывается конструктор с двумя аргументами, причем первым аргументом передается значение "Белый", а вторым аргументом передается значение code. Наконец, в теле конструктора без аргументов выполняется команда this("Желтый",1), означающая вызов конструктора с двумя аргументами со значениями "Желтый" и 1.

Таким образом, здесь мы используем инструкцию this для вызова одной из версий конструктора. В круглых скобках после ключевого слова this указываются аргументы, которые передаются вызываемой версии конструктора. Важное условие вызова в конструкторе другой версии конструктора состоит в том, что соответствующая команда должна быть первой в теле конструктора.

На основе класса Alpha создается подкласс Bravo. Как отмечалось, в классе Bravo описано символьное поле symbol и закрытый метод show(), вызов которого приводит к отображению значения символьного поля. Но поскольку метод закрытый, то он может быть вызван только в теле класса Bravo. Мы используем метод для вызова в конструкторе (во всех шести версиях).

НА ЗАМЕТКУ

В теле метода show() обращение к полю symbol выполняется в формате this.symbol. Однако необходимости в этом нет — можно было вместо инструкции this.symbol использовать идентификатор symbol.

Основной код класса Bravo — это программный код разных версий конструктора. Например, есть версия конструктора с тремя аргументами: текстовым name, целочисленным code и символьным symbol. Первой в теле данной версии конструктора является команда super(name,code), которой вызывается версия конструктора суперкласса с двумя аргументами. В результате поля name и code получают значения, а в окне вывода появляется соответствующее сообщение. Затем командой this.symbol=symbol присваивается значение символьному полю, а вызовом метода show() значение поля отображается в окне вывода.

Похожим образом организован программный код и в прочих версиях конструктора класса Bravo. Поэтому остановимся только на наиболее интересных моментах. Так, инструкция super() в теле конструктора означает вызов конструктора суперкласса без аргументов (напомним, что инструкция вызова конструктора суперкласса указывается первой в теле конструктора). Но по умолчанию, если команда вызова конструктора суперкласса не указана, автоматически вызывается версия конструктора суперкласса без аргументов. Поэтому теоретически инструкцию super() можно не указывать. Для сравнения: в версии конструктора с одним символьным аргументом инструкция super() присутствует, а в версии конструктора класса Bravo без аргументов такой инструкции нет. Тем не менее в обоих случаях при вызове конструктора подкласса сначала вызывается версия конструктора суперкласса без аргументов.

Далее, в теле конструктора класса Bravo с двумя аргументами (текстовым и числовым) выполняется команда this(name,code,'D'). Данной командой вызывается версия конструктора класса Bravo с тремя аргументами. То есть здесь мы используем уже знакомый нам прием.

ДЕТАЛИ

Инструкция this с круглыми скобками означает вызов версии конструктора того же самого класса, в конструкторе которого размещена соответствующая команда. Инструкция super с круглыми скобками означает вызов версии конструктора другого класса, но являющегося

при этом суперклассом для класса, в конструкторе которого находится команда с super-инструкцией.
Что касается непосредственно конструктора класса Bravo, в котором выполняется команда this(name,code,'D'), то стоит заметить, что явно указанной super-инструкции там нет, но при этом и конструктор суперкласса без аргументов не вызывается. Ситуация более «хитрая», по сравнению с тем, когда просто явно не указана команда вызова конструктора суперкласса. Дело в том, что соответствующая инструкция по вызову конструктора суперкласса «спрятана» в команде this(name,code,'D'): вызов конструктора суперкласса происходит при вызове версии конструктора класса Bravo с тремя аргументами.

В главном методе программы командой Bravo obj объявляется объектная переменная obj класса Bravo. Затем разными командами (в них по-разному вызывается конструктор) создаются объекты, а ссылка записывается в переменную obj.

НА ЗАМЕТКУ

На самом деле можно было ограничиться созданием анонимных объектов и не присваивать значением объектной переменной ссылку на очередной создаваемый объект.

Наследование и закрытые члены

При наследовании из суперкласса в подкласс автоматически «перескакивают» все поля и методы, которые не являются закрытыми. Возникает резонный вопрос: а что же происходит с закрытыми членами класса? Вопрос актуальный, поскольку нередко встречается, например, такая ситуация: в суперклассе имеется открытый метод, который обращается к закрытым полям. Поскольку метод открытый, то он наследуется в подклассе и его можно вызвать из объекта подкласса. При этом методу для работы нужно обращаться к закрытым полям, которые в принципе не наследуются. Чтобы разобраться в такой ситуации, рассмотрим небольшой пример, представленный в листинге 6.3.

Листинг 6.3. Программный код проекта UsingPrivatesApplication

```
// Суперкласс:
class Alpha{
  // Закрытое поле:
```

```
  private int code;
  // Открытый метод для присваивания значения
  // закрытому полю:
  public void set(int code){
    this.code=code;
  }
  // Открытый метод для отображения значения
  // закрытого поля:
  public void show(){
    System.out.println("Поле code: "+code);
  }
  // Конструктор класса:
  Alpha(int code){
    set(code);
  }
}
// Подкласс:
class Bravo extends Alpha{
  // Конструктор подкласса:
  Bravo(int code){
    // Вызов конструктора суперкласса:
    super(code);
  }
}
// Класс с главным методом программы:
class UsingPrivatesDemo{
  public static void main(String[] args){
    // Создание объекта подкласса:
    Bravo obj=new Bravo(100);
    // Проверка значения поля:
    obj.show();
    // Присваивание значения полю:
```

```
    obj.set(200);
    // Отображение значения поля:
    obj.show();
  }
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 6.3)

```
Поле code: 100
Поле code: 200
```

Программа очень простая. В суперклассе Alpha описано закрытое целочисленное поле code. Для присваивания значения полю предназначен открытый метод set(). Отобразить значение поля в консольном окне можно с помощью открытого метода show(). Также в классе описан конструктор с одним аргументом. Аргумент определяет значение поля code создаваемого объекта.

На основе класса Alpha путем наследования создается подкласс Bravo. В подклассе Bravo описан лишь конструктор, код которого состоит из команды вызова конструктора суперкласса. Поскольку методы set() и show() в суперклассе Alpha описаны как открытые, то в подклассе Bravo они наследуются. Поэтому у объекта obj подкласса Bravo, создаваемого в главном методе программы, имеется и метод set(), и метод show(). Но пикантность ситуации в том, что и методы set() и show(), и конструктор класса Bravo, обращаются к закрытому полю code, описанному в классе Alpha и ненаследуемому в классе Bravo. Вместе с тем команда Bravo obj=new Bravo(100) создания объекта подкласса проходит в «штатном режиме», а проверка с помощью команды obj.show() показывает, что «не унаследованное» поле code получает значение 100. Более того, если выполнить команду obj.set(200) и снова выполнить проверку с помощью команды obj.show(), легко заметить, что значение поля code изменилось и стало равно 200. Как такое возможно? Все на самом деле просто. Достаточно конкретизировать, что мы понимаем под понятием «не наследуется». Дело в том, что «технически» поле code в объекте подкласса Bravo существует, просто в теле класса Bravo прямого доступа к этому полю нет. Проще говоря, если мы в программном коде в классе Bravo

попытаемся использовать идентификатор code, то это приведет к ошибке на этапе компиляции программы. Но если мы получаем доступ к полю через унаследованные открытые методы, то они «знают» где искать поле. Данное замечание относится не только к закрытым полям, но и к закрытым методам в суперклассе. Поэтому наличие в суперклассе закрытых полей и методов обычно при наследовании к проблемам не приводит.

Наследование, пакеты и уровни доступа

Мы уже знаем, что если член класса описан без спецификатора уровня доступа или со спецификатором уровня доступа public, то он является *открытым*. Также мы знаем, что если член класса описан со спецификатором уровня доступа private, то такой член класса является *закрытым*. Кроме этого, существуют еще *защищенные* члены класса. Защищенные члены класса описываются со спецификатором доступа protected. Различие между всеми перечисленными ситуациями фактически сводится к тому, где, в каком месте программного кода соответствующий член доступен, а где — нет.

Язык программирования Java полностью объектно-ориентированный. Когда речь идет о получении доступа к члену класса, то код, из которого мы хотим получить доступ, может быть в том же классе (в котором описан член класса), а может быть в другом классе. Случай, когда мы получаем доступ к члену класса в пределах этого же класса, тривиальный: в классе все его члены доступны. Поэтому имеет смысл остановиться на ситуации, когда мы хотим в одном классе получить доступ к члену другого класса. При этом важны два обстоятельства:

- связаны ли классы механизмом наследования;
- находятся ли классы в одном *пакете* или в разных пакетах (пакеты обсуждаются далее).

Доступность членов класса, описанных с различными спецификаторами уровня доступа, иллюстрирует табл. 6.1 (символ + означает доступность члена класса, и символ — означает недоступность члена класса).

Содержимое таблицы можно кратко резюмировать следующим набором правил.

- Открытые члены класса, описанные со спецификатором public, доступны везде (во всех классах и пакетах).

- Защищенные члены класса, описанные с ключевым словом protected, доступны в классах того же пакета, а также во всех подклассах, вне зависимости от того, в каком они пакете.
- Открытые члены класса, описанные без спецификатора доступа, доступны в классах в пределах того же пакета.
- Закрытые члены класса, описанные со спецификатором private, доступны только в пределах класса, в котором они описаны.

Табл. 6.1. Уровни доступа членов класса

Место, из которого получаем доступ	Спецификатор уровня доступа, с которым описан член класса			
	public	protected	не указан	private
В том же классе	+	+	+	+
В подклассе в том же пакете	+	+	+	-
В обычном класса в том же пакете	+	+	+	-
В подклассе другого пакета	+	+	-	-
В обычном классе в другом пакете	+	-	-	-

НА ЗАМЕТКУ

Классы могут описываться со спецификатором доступа public. Если класс описан без спецификатора доступа public, то он доступен в пределах пакета. Если класс описан со спецификатором доступа public, то он доступен и вне пакета. Такой класс называют public-классом или открытым классом. В файле может быть только один открытый класс, а название файла должно совпадать с названием открытого класса.

Нам осталось приоткрыть завесу над принципом распределения классов по пакетам. Идея проста: классы группируются по пакетам, которые играют роль своеобразных «контейнеров», или объединений классов (и других пакетов, которые обычно называют подпакетами). В пределах контейнера название класса должно быть уникальным, но если классы находятся в разных пакетах, то их названия могут совпадать.

Чтобы создать пакет, необходимо в самом начале программного кода (с описанием классов, входящих в пакет) поместить ключевое слово package, после которого указывается имя создаваемого пакета:

```
package имя_пакета;
```

Если создается подпакет в пакете, то имя пакета и подпакета разделяется точкой:

```
package имя_пакета.имя_подпакета;
```

Так, при создании пакета с именем mydata используем такую инструкцию:

```
package mydata;
```

При создании подпакета mybox в пакете mydata, понадобится такая инструкция:

```
package mydata.mybox;
```

При этом «технически» размещение файлов пакетов по каталогам строго регламентировано и соотнесено с логической структурой пакетов и подпакетов.

> **НА ЗАМЕТКУ**
>
> Файл может содержать только одну package-инструкцию, и она должна быть первой инструкцией в файле.

Для того чтобы в программе можно было использовать класс или классы из какого-то пакета, класс или весь пакет *импортируют*. В таком случае используют инструкцию import, после которой указывается имя пакета и, через точку, имя импортируемого класса или звездочка * (если импортируются все public-классы из пакета). Для импортирования класса с названием MyClass из пакета mydata используют следующую инструкцию:

```
import mydata.MyClass;
```

Если мы хотим импортировать все public-классы из пакета mydata, вместо имени класса следует поставить звездочку *, как показано ниже:

```
import mydata.*;
```

При этом классы из подпакетов импортируемого пакета не импортируются. Чтобы импортировать классы из подпакета mybox пакета mydata используем такую инструкцию:

```
import mydata.mybox.*;
```

Альтернативой к использованию import-инструкции является использование полных ссылок на классы и методы, с явным указанием пакетов и подпакетов, в которых они размещены. Например, мы могли бы без использования инструкции import javax.swing.* вызывать статический метод showMessageDialog() из класса JOptionPane в формате javax.swing.JOptionPane.showMessageDialoig(), но это, очевидно, не очень удобно.

НА ЗАМЕТКУ

Файл может содержать несколько import-инструкций. Такие инструкции размещаются в начале программного кода, а если в программном коде есть package-инструкция, то сразу после нее.

Хочется также подчеркнуть, импортируются только public-классы. Несмотря на то, что в файле может описываться только один public-класс, данное обстоятельство не отменяет возможности для пакета содержать несколько public-классов. Просто каждый такой класс описывается в отдельном файле. Каждый файл, в свою очередь, содержит package-инструкцию с названием одного и того же пакета.

Многие стандартные классы, наиболее часто используемые в работе, размещаются в пакете java.lang. Но поскольку утилиты из этого пакета доступны по умолчанию, нет необходимости явно импортировать данный пакет.

При работе со статическими членами классов из импортируемых пакетов существует возможность несколько упростить процедуру обращения к таким членам. Для этого используют *статический импорт*. При статическом импорте после инструкции import указывается ключевое слово static, а далее — полная ссылка на статический член класса. Пример такой инструкции приведен ниже:

```
import static javax.swing.JOptionPane.showMessageDialog;
```

Как следствие метод showMessageDialog() можно вызывать без указания класса JOptionPane. Если в инструкции статического импорта заменить название статического члена класса на звездочку, то в программе можно будет использовать все статические члены данного класса без ссылки на класс. Ниже приведен пример инструкции статического импорта всех статических членов класса JOptionPane:

```
import static javax.swing.JOptionPane.*;
```

Еще один вопрос прикладного характера, который мы кратко рассмотрим, связан с особенностями использования пакетов при работе со средой NetBeans.

По умолчанию, если мы при создании проекта не создаем пакет, все созданные в программе классы помещаются в так называемый *пакет по умолчанию*. Это не очень хороший стиль программирования, поэтому далее отметим основные моменты, которые следует иметь в виду при создании в среде NetBeans нового проекта, в котором создается пользовательский пакет.

Итак, напомним, что для создания нового проекта в окне среды разработки NetBeans выбирается команда **New Project** из меню **File**. Откроется окно **New Project**, в котором в разделе **Categories** выбирается пункт **Java**, а в разделе **Projects** выбираем пункт **Java Application**. Откроется окно **New Java Application**, представленное на рис. 6.4.

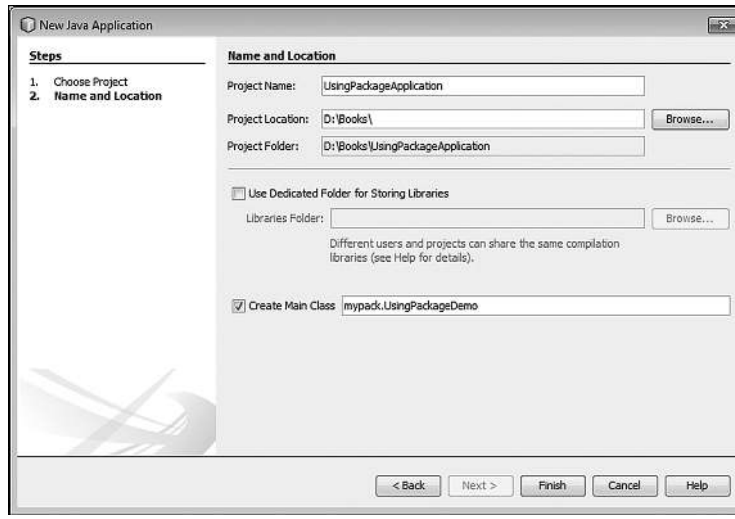


Рис. 6.4. Окно **New Java Application** с выполненными в нем настройками

В поле **Project Name** указываем названием проекта UsingPackageApplication. В поле **Create Main Class** (с установленным флажком) указываем текст mypack.UsingPackageDemo. Последнее выражение означает, что главный класс в проекте должен называться UsingPackageDemo, и размещается он в пакете mypack. Поэтому программный код должен начинаться инструкцией package mypack. Мы используем для проекта программный код, представленный в листинге 6.4.

Листинг 6.4. Программный код проекта UsingPackageApplication

```
// Пакет:
package mypack;
```

```
// Статический импорт:
import static javax.swing.JOptionPane.*;
// Главный класс:
class UsingPackageDemo{
  // Главный метод:
  public static void main(String[] args){
    // Отображение диалогового окна с сообщением:
    showMessageDialog(null,
      // Текст сообщения:
      "Статический импорт — это удобно!",
      // Название окна:
      "Пакет mypack",
      // Тип окна:
      WARNING_MESSAGE
    );
  }
}
```

На рис. 6.5 показано окно среды разработки NetBeans с открытым в нем проектом UsingPackageApplication.

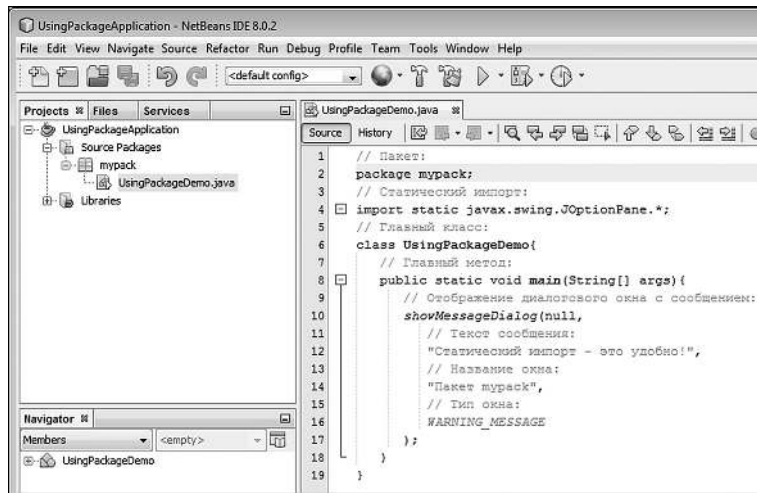


Рис. 6.5. В окне среды разработки открыт проект UsingPackageApplication

НА ЗАМЕТКУ

Во внутреннем окне **Projects** в левой верхней части окна среды разработки видно, что файл UsingPackageDemo.java с программным кодом находится внутри раскрывающегося элемента **mypack**, соответствующего пакету, в котором размещается класс UsingPackageDemo.

При выполнении программы появляется диалоговое окно, представленное на рис. 6.6.

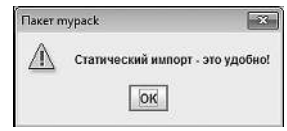


Рис. 6.6. При выполнении программы с использованием статического импорта появляется диалоговое окно

Что касается собственно программного кода (см. листинг 6.4), то он достаточно простой. Помимо инструкции package mypack в начале программного кода, внимания достойны лишь несколько особенностей этого кода. Так, инструкцией import static javax.swing.JOptionPane.* статически импортируются все статические члены класса JOptionPane. Поэтому при вызове статического метода showMessageDialog() и обращении к статической константе WARNING_MESSAGE имя класса JOptionPane не указывается, что сокращает объем программного кода и делает команды не такими громоздкими.

Переопределение методов

Ты туда не ходи. Ты сюда ходи.

Из к/ф «Джентльмены удачи»

Наследуемые из суперкласса в подклассе методы можно *переопределять*. Речь идет о следующем. Допустим, имеется некоторый суперкласс, а в нем метод, который наследуется в подклассе. Однако нам в силу определенных причин нужно сделать так, чтобы при вызове из объекта подкласса методы выполнялся несколько иначе, по сравнению с тем, как метод выполняется при вызове из объекта суперкласса. В таком случае, чтобы переопределить унаследованный метод, в подклассе данный метод описывается заново в явном виде. Это и есть краткая суть механизма переопределения методов при наследовании. Далее рассмотрим

небольшие примеры и некоторые «ситуации», которые иллюстрируют особенности переопределения методов.

ДЕТАЛИ

В языке Java есть ключевое слово final, которое позволяет решать ряд важных задач. Так, если с ключевым словом final описана переменная, то такая переменная является константой: после инициализации значение этой переменной изменить нельзя. Если с ключевым словом final описан метод в суперклассе, то в подклассе такой метод не переопределяется. Наконец, если описать с ключевым словом final класс, то такой класс не может быть суперклассом — другими словами, на основе такого класса нельзя создать (путем наследования) другой класс.

Общие принципы переопределения методов

Программа в листинге 6.5 содержит описание суперкласса Alpha, у которого есть текстовое поле name. Также в классе описан метод show(), которым отображается значение данного поля. На основе суперкласса Alpha создается подкласс Bravo. В классе Bravo описывается целочисленное поле code, а метод show() переопределяется так, что теперь при вызове метода отображается не только значение поля name, но и значение поля code. Далее рассмотрим программный код примера.

Листинг 6.5. Программный код проекта OverrideMethodApplication

```
// Суперкласс:
class Alpha{
  // Текстовое поле:
  String name;
  // Метод для отображения значения текстового поля:
  void show(){
    System.out.println("Объект класса Alpha:");
    System.out.println("Поле name — "+name);
  }
  // Конструктор класса:
  Alpha(String name){
    this.name=name;
```

```
  }
}
// Подкласс:
class Bravo extends Alpha{
  // Целочисленное поле:
  int code;
  // Переопределение метода. Новой версией метода
  // отображаются значения двух полей:
  void show(){
    System.out.println("Объект класса Bravo:");
    System.out.println("Поле name — "+name);
    System.out.println("Поле code — "+code);
  }
  // Конструктор класса:
  Bravo(String name,int code){
    // Вызов конструктора суперкласса:
    super(name);
    this.code=code;
  }
}
// Главный класс:
class OverrideMethodDemo{
  // Главный метод:
  public static void main(String[] args){
    // Создание объекта суперкласса:
    Alpha objA=new Alpha("objA");
    // Создание объекта подкласса:
    Bravo objB=new Bravo("objB",123);
    // Вызов метода из объекта суперкласса:
    objA.show();
    // Вызов метода из объекта подкласса:
    objB.show();
  }
}
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 6.1)

```
Объект класса Alpha:
Поле name — objA
Объект класса Bravo:
Поле name — objB
Поле code — 123
```

Код программы более чем простой, поэтому комментариев не требует. Мы лишь отметим, что если бы в подклассе метод show() не был переопределен, то при вызове этого метода из объекта подкласса отображалось бы лишь значение поля name. Но мы, несмотря на то что метод show() наследуется в классе Bravo, фактически переписали заново данный метод. Как следствие, если метод show() вызывается из объекта суперкласса, то выполняется тот код, которым определяется метод show() в суперклассе. Если метод show() вызывается из объекта подкласса, то выполняется тот код, с которым метод описан (переопределен) в подклассе.

НА ЗАМЕТКУ

В подклассе в строке перед описанием новой версии переопределяемого метода можно поместить инструкцию @Override, которую называют *аннотацией* (как и все инструкции, начинающиеся с символа @). Аннотация @Override перед переопределяемым методом повышает надежность кода. При наличии данной аннотации компилятор проверяет, действительно ли переопределен метод и есть ли в суперклассе метод с такой сигнатурой. В данном случае предупреждается ошибка, связанная с тем, что пользователь вместо переопределения унаследованного метода случайно в подклассе описал новый метод.

Вызов разных версий метода

В некоторых случаях, при переопределении методов, бывает удобно или даже необходимо вызвать исходную (описанную в суперклассе) версию переопределенного метода. В листинге 6.6 представлена небольшая вариация на тему предыдущего примера (практически все комментарии для сокращения объема кода удалены, жирным шрифтом в подклассе выделена команда вызова исходной версии переопределяемого метода).

Листинг 6.6. Программный код проекта MoreOverridingApplication

```
class Alpha{
  String name;
  void show(){
    System.out.println("Объект "+name);
  }
  Alpha(String name){
    this.name=name;
  }
}
class Bravo extends Alpha{
  int code;
  void show(){
    // Вызов версии метода из суперкласса:
    super.show();
    System.out.println("Числовое поле "+code);
  }
  Bravo(String name,int code){
    // Вызов конструктора суперкласса:
    super(name);
    this.code=code;
  }
}
class MoreOverridingDemo{
  public static void main(String[] args){
    Alpha objA=new Alpha("objA");
    Bravo objB=new Bravo("objB",123);
    objA.show();
    objB.show();
  }
}
```

Результат выполнения программы следующий:

Результат выполнения программы (из листинга 6.6)

```
Объект objA
Объект objB
Числовое поле 123
```

Общая ситуация такая же, как в предыдущем примере (см. листинг 6.5): описан суперкласс Alpha с текстовым полем name и классом show(), отображающим значение поля. В подклассе Bravo добавляется целочисленное поле code, а метод show() переопределяется так, что теперь методом в окно вывода выводятся значения обоих полей. Интерес представляет код, которым переопределяется метод show(). В теле метода из класса Bravo первой командой указана инструкция super.show(). Инструкция super.show() означает вызов версии метода show() из суперкласса. Таким образом, переопределение метода show() в подклассе Bravo выполнено так, что сначала вызывается исходная версия метода, описанная в суперклассе, а затем выполняются дополнительные команды (в данном случае команда System.out.println("Числовое поле "+code)).

Ситуация может быть еще более запутанной, если в подклассе объявляются поля с такими же названиями, как у полей в суперклассе. Такие поля «перекрываются» в том смысле, что объект подкласса имеет непосредственный доступ к полю, описанному в подклассе. Для доступа в объекте подкласса к полю, унаследованному из суперкласса и «перекрытому» полем в подклассе с таким же названием, как и в случае с разными версиями переопределяемого метода, используют ключевое слово super. Небольшой пример подобной ситуации приведен в листинге 6.7.

Листинг 6.7. Программный код проекта HidingFieldApplication

```
// Суперкласс:
class Alpha{
  // Текстовое поле:
  String name;
}
// Подкласс:
class Bravo extends Alpha{
  // Текстовое поле:
```

```
  String name;
  // Метод для отображения значений полей:
  void show(){
    // Значение унаследованного из суперкласса поля:
    System.out.println("Из класса Alpha: "+super.name);
    // Значение описанного в подклассе поля:
    System.out.println("Из класса Bravo: "+name);
  }
  // Конструктор:
  Bravo(String a,String b){
    // Вызов конструктора (по умолчанию) суперкласса:
    super();
    // Значение унаследованного из суперкласса поля:
    super.name=a;
    // Значение описанного в подклассе поля:
    name=b;
  }
}
// Главный класс:
class HidingFieldDemo{
  public static void main(String[] args){
    // Создание объекта подкласса:
    Bravo obj=new Bravo("alpha","bravo");
    // Проверка значений полей:
    obj.show();
  }
}
```

Ниже приведен результат выполнения программы.

Результат выполнения программы (из листинга 6.7)

```
Из класса Alpha: alpha
Из класса Bravo: bravo
```

В данном примере суперкласс Alpha описывается всего с одним текстовым полем name. Но в подклассе Bravo описано текстовое поле с таким же названием. Получается, что одно поле name наследуется из суперкласса Alpha, а другое поле name описывается непосредственно в подклассе. В таком случае в подклассе Bravo по умолчанию инструкция name означает поле, описанное в подклассе. Если нам нужно получить доступ к полю name, унаследованному из суперкласса, используем инструкцию super.name. Такого типа инструкции использованы в описании конструктора и метода show() в подклассе Bravo.

Виртуальность методов и конструкторов

Прежде, чем приступить к обсуждению следующей темы, связанной с наследованием, рассмотрим небольшой, но иллюстративный пример, представленный в листинге 6.8. Мы описываем суперкласс Alpha, в котором имеется текстовое поле name. В классе также описан конструктор и три метода. Методы hello() и objectCreated() очень простые — при их вызове в окне вывода отображается сообщение с названием класса Alpha. Метод hello() вызывается в теле метода show(). После этого выводится сообщение со значением поля name.

Метод objectCreated() вызывается в теле конструктора класса Alpha. Помимо этого, в конструкторе присваивается значение полю name.

Подкласс Bravo создается наследованием класса Alpha. В классе Bravo описан конструктор, в теле которого вызывается конструктор суперкласса. Еще в подклассе Bravo переопределяются методы hello() и objectCreated(). Переопределение сводится к тому, что в сообщениях, которые выводятся методами, вместо имени класса Alpha использовано имя класса Bravo.

В главном классе программы создаются объекты суперкласса и подкласса. Затем из каждого из объектов вызывается метод show(). Теперь рассмотрим программный код.

Листинг 6.8. Программный код проекта VirtualMethodsApplication

```
// Суперкласс:
class Alpha{
  // Текстовое поле:
  String name;
  // Метод для отображения сообщения:
```

```
  void objectCreated(){
    System.out.println("Создан объект класса Alpha");
  }
  // Метод для отображения сообщения:
  void hello(){
    System.out.println("Объект класса Alpha");
  }
  // Метод для отображения значения поля:
  void show(){
    // Вызов метода для отображения сообщения:
    hello();
    // Отображение значения поля:
    System.out.println("Поле name: "+name);
  }
  // Конструктор:
  Alpha(String txt){
    // Вызов метода для отображения сообщения:
    objectCreated();
    // Присваивание значения полю:
    name=txt;
  }
}
// Подкласс:
class Bravo extends Alpha{
  // Переопределение метода:
  void objectCreated(){
    System.out.println("Создан объект класса Bravo");
  }
  // Переопределение метода:
  void hello(){
    System.out.println("Объект класса Bravo");
  }
```

```
  // Конструктор:
  Bravo(String txt){
    // Вызов конструктора суперкласса:
    super(txt);
  }
}
// Главный класс:
class VirtualMethodsDemo{
  // Главный метод:
  public static void main(String[] args){
    // Создание объекта суперкласса:
    Alpha objA=new Alpha("alpha");
    // Отображение значения поля:
    objA.show();
    // Создание объекта подкласса:
    Bravo objB=new Bravo("bravo");
    // Отображение значения поля:
    objB.show();
  }
}
```

Ниже показано, как выглядит результат выполнения программы.

Результат выполнения программы (из листинга 6.8)

```
Создан объект класса Alpha
Объект класса Alpha
Поле name: alpha
Создан объект класса Bravo
Объект класса Bravo
Поле name: bravo
```

Проанализируем результат выполнения программы. При создании в главном методе командой `Alpha objA=new Alpha("alpha")` объекта `objA`

суперкласса в конструкторе вызывается метод objectCreated(). Методом отображается сообщение Создан объект класса Alpha. Далее, при вызове метода show() из объекта objA на самом деле вызывается метод hello(), которым выводится сообщение Объект класса Alpha. После этого появляется сообщение Поле name: alpha со значением "alpha" поля name объекта objA. Здесь все просто.

В принципе нечто похожее происходит и с объектом objB подкласса Bravo. Правда, без дополнительных пояснений результат может быть неожиданным. Итак, выясним, что происходит при выполнении команды Bravo objB=new Bravo("bravo"), которой создается объект objB подкласса Bravo. Здесь вызывается конструктор подкласса Bravo, а в конструкторе подкласса вызывается конструктор суперкласса Alpha. При вызове конструктора суперкласса вызывается метод objectCreated(), после чего полю name присваивается значение "bravo". Несложно сообразить, что в результате вызова метода objectCreated() появляется сообщение Создан объект класса Bravo. Таким образом, в данной ситуации вызывается переопределенная в подклассе Bravo версия метода objectCreated(). Нечто похожее происходит при вызове метода show() из объекта objB. При выполнении метода show() вызывается метод hello(), причем переопределенная версия. Поэтому при вызове метода hello() появляется сообщение Объект класса Bravo. Ситуация показательная: метод show() наследуется без переопределения из суперкласса Alpha в подклассе Bravo, и в этом методе вызывается переопределенный метод hello(). Можно все сформулировать несколько иначе. Если метод show() вызывается из объекта суперкласса, то версия метода hello() используется из суперкласса. Если метод show() вызывается из объекта подкласса, то используется версия метода hello() из подкласса. Такое свойство метода show() называется *виртуальностью*. В Java все методы виртуальные. Это же относится и к конструкторам.

Перегрузка и переопределение методов

При *перегрузке* методов создается несколько версий одного и того же метода. Все версии имеют одинаковые названия, но отличаются сигнатурами. При *переопределении* методов в подклассе описывается новая версия метода из суперкласса, причем новая версия имеет не только точно такое же название, как исходный метод, но и такую же сигнатуру. Оба механизма (перегрузка и переопределение) могут использоваться одновременно. Очень простой иллюстративный пример приведен в листинге 6.9.

Листинг 6.9. Программный код проекта OverloadingAndOverridingApplication

```
// Суперкласс:
class Alpha{
  // Версия метода без аргумента:
  void show(){
    System.out.println("Класс Alpha");
  }
  // Версия метода с текстовым аргументом:
  void show(String txt){
    System.out.println(txt);
  }
}
// Подкласс:
class Bravo extends Alpha{
  // Переопределение версии метода без аргументов:
  void show(){
    System.out.println("Класс Bravo");
  }
  // Версия метода с целочисленным аргументом:
  void show(int num){
    System.out.println("Число "+num);
  }
}
// Главный класс:
class OverloadingAndOverridingDemo{
  // Главный метод:
  public static void main(String[] args){
    // Создание объекта суперкласса:
    Alpha objA=new Alpha();
    // Вызов разных версий метода:
    objA.show();
    objA.show("Объект objA");
```

```
    // Создание объекта подкласса:
    Bravo obj=new Bravo();
    // Вызов разных версий метода:
    obj.show();
    obj.show("Объект objB");
    obj.show(123);
  }
}
```

Ниже показано, как выглядит результат выполнения программы.

Результат выполнения программы (из листинга 6.1)

```
Класс Alpha
Объект objA
Класс Bravo
Объект objB
Число 123
```

В суперклассе Alpha описаны две версии метода show(): без аргументов и с текстовым аргументом. В подклассе Bravo переопределяется версия метода show() без аргументов, наследуется версия метода show() с текстовым аргументом, и описывается еще одна версия данного метода с целочисленным аргументом. Пример простой, поэтому думается, что особенности кода и результатов его выполнения особых комментариев не требуют.

Метод toString()

Все классы в языке Java через систему наследования являются (непрямыми) «наследниками» класса Object, который находится в вершине иерархии классов. В классе Object объявлен public-метод toString(), который автоматически вызывается при попытке преобразования объектов в текстовый формат. Для большинства стандартных классов метод toString() переопределен в соответствии со спецификой класса. Если в классе пользователя переопределить метод toString(), то объекты данного класса можно будет использовать в выражениях, подразумевающих наличие в соответствующем месте в выражении текстового операнда.

НА ЗАМЕТКУ

Объект пользовательского класса можно передавать в качестве текстового операнда и без переопределения метода toString(). Но в таком случае текстовое значение, к которому приводится объект, будет малоинформативным.

Таким образом, в классе можно описать метод с названием toString(). Метод описывается с ключевым словом public, у метода нет аргументов, а результатом метод возвращает текстовое значение класса String. Если это сделать, то при использовании объекта класса в выражениях, подразумевающих в соответствующем месте наличие текстового значения, вместо объекта будет «подставляться» текст, возвращаемый результатом методом toString(). Небольшой пример переопределения в пользовательском классе метода toString() приведен в листинге 6.10.

Листинг 6.10. Программный код проекта UsingToStringApplication

```
// Пользовательский класс:
class MyClass{
  // Текстовое поле:
  String name;
  // Целочисленное поле:
  int code;
  // Конструктор:
  MyClass(String txt,int num){
    name=txt;
    code=num;
  }
  // Переопределение метода toString():
  public String toString(){
    // Локальная текстовая переменная:
    String txt="Объект класса MyClass\n";
    txt+="Поле name: "+name+"\n";
    txt+="Поле code: "+code+"\n";
    // Импровизированная "линия":
    for(int k=1;k<=21;k++){
```

```
      txt+="-";
    }
    // Результат метода:
    return txt;
  }
}
// Главный класс:
class UsingToStringDemo{
  // Главный метод:
  public static void main(String[] args){
    // Создание объекта:
    MyClass obj=new MyClass("объект obj",123);
    // Объект передан аргументом методу:
    System.out.println(obj);
  }
}
```

Результат выполнения программы будет таким:

Результат выполнения программы (из листинга 6.10)

```
Объект класса MyClass
Поле name: объект obj
Поле code: 123
----------
```

В программе описан класс MyClass. В классе есть два поля: текстовое поле name и целочисленное поле code. Конструктор описан с двумя аргументами, которые определяют значения полей name и code. Но наибольший интерес представляет описание метода toString(). В теле метода объявляется локальная текстовая переменная txt. В текстовую переменную заносится текст, включая название класса MyClass, значения полей name и code, содержит несколько инструкций перехода к новой строке \n, и заканчивается импровизированной «линией» из черточек.

В главном методе программы командой MyClass obj=new MyClass("объект obj",123) создается объект obj класса MyClass. Затем выполняется команда System.out.

println(obj). Ее особенность в том, что аргументом методу println(), которому обычно передается аргументом текст, в данном случае аргументом передается объект obj класса MyClass. Поэтому автоматически для объекта obj вызывается метод toString(), а результат метода используется вместо ссылки на объект. Отсюда и результат выполнения программы.

Объект подкласса и переменная суперкласса

Меня не проведешь. Приемы сыщиков я вижу на пять футов вглубь.

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Есть важная, в некотором смысле фундаментальная, особенность, связанная с объектными переменными суперкласса и объектами подкласса. Истина проста: объектная переменная суперкласса может ссылаться на объект подкласса. Правда, через объектную переменную суперкласса можно получить доступ не ко всем членам объекта подкласса, а только к тем, которые объявлены в суперклассе. Для большей конкретики рассмотрим простой пример, в котором доступ к объекту подкласса выполняется, кроме прочего, через объектную переменную суперкласса. Программный код примера представлен в листинге 6.11.

Листинг 6.11. Программный код проекта SuperAndSubApplication

```
// Суперкласс:
class Alpha{
  // Текстовое поле:
  String name;
  // Метод для отображения значения поля:
  void show(){
    // Локальная текстовая переменная:
    String txt="Объект класса Alpha\n";
    txt+="Поле name: "+name+"\n";
    for(int k=1;k<=20;k++){
      txt+="-";
    }
    // Отображение сообщения:
```

```
    System.out.println(txt);
  }
}
// Подкласс:
class Bravo extends Alpha{
  // Целочисленное поле:
  int code;
  // Переопределение метода:
  void show(){
    // Локальная текстовая переменная:
    String txt="Объект класса Bravo\n";
    txt+="Поле name: "+name+"\n";
    txt+="Поле code: "+code+"\n";
    for(int k=1;k<=20;k++){
      txt+="-";
    }
    // Отображение сообщения:
    System.out.println(txt);
  }
}
// Главный класс:
class SuperAndSubDemo{
  // Главный метод:
  public static void main(String[] args){
    // Создание объекта суперкласса:
    Alpha objA=new Alpha();
    // Присваивание значения полю объекта суперкласса:
    objA.name="alpha";
    // Вызов метода из объекта суперкласса:
    objA.show();
    // Создание объекта подкласса:
    Bravo objB=new Bravo();
```

```
    // Присваивание значений полям объекта подкласса:
    objB.name="bravo";
    objB.code=123;
    // Вызов метода из объекта подкласса:
    objB.show();
    // Переменной суперкласса значением присваивается
    // ссылка на объект подкласса:
    objA=objB;
    // Значение поля объекта подкласса изменяется
    // через переменную суперкласса:
    objA.name="charlie";
    // Вызов метода из объекта подкласса
    // через переменную суперкласса:
    objA.show();
    // Вызов метода из объекта подкласса
    // через переменную подкласса:
    objB.show();
  }
}
```

В результате выполнения программы получим следующее:

Результат выполнения программы (из листинга 6.11)

```
Объект класса Alpha
Поле name: alpha
--------------------
Объект класса Bravo
Поле name: bravo
Поле code: 123
--------------------
Объект класса Bravo
Поле name: charlie
Поле code: 123
```

```
--------------------
Объект класса Bravo
Поле name: charlie
Поле code: 123
--------------------
```

В суперклассе Alpha описано текстовое поле name и метод show(), которым отображается сообщение с именем класса Alpha, значением поля name и горизонтальной «линией» из черточек.

В подклассе Bravo объявляется целочисленное поле code и переопределяется метод show(). Версией метода show() из класса Bravo отображается название класса, значения полей name и code и декоративная «линия».

> **НА ЗАМЕТКУ**
>
> Метод show() и в суперклассе, и в подклассе описан по такой схеме: в теле объявляется локальная текстовая переменная, формируется ее значение, после чего значение текстовой переменной отображается в окне вывода.

В главном методе программы командой Alpha objA=new Alpha() создается объект суперкласса, и ссылка на него записывается в переменную objA. Командой objA.name="alpha" полю name объекта суперкласса присваивается значение, после чего командой objA.show() информация об объекте выводится в консольное окно (окно вывода). Здесь все происходит штатно.

Командой Bravo objB=new Bravo() создается объект подкласса, ссылка на который записывается в переменную objB. Командами objB.name="bravo" и objB.code=123 полям объекта присваиваются значения, после чего командой objB.show() в окне вывода отображаются сведения об объекте подкласса. Здесь тоже все просто и ожидаемо.

В результате выполнения команды objA=objB ссылка на объект подкласса (тот, на который ссылается переменная подкласса objB), записывается в переменную суперкласса objA. В итоге и переменная objA, и переменная objB ссылаются на один и тот же объект (объект подкласса Bravo). Но между переменными objA и objB отличия более чем формальные: через переменную objB есть доступ ко всем членам объекта подкласса (поля name и code, а также метод show()), а через переменную objA есть доступ только к тем

членам объекта подкласса, которые объявлены в суперклассе (поле name и метод show()). Поэтому, например, корректной является команда objA.name="charlie", которой изменяется значение поля name объекта подкласса (в то же время, через переменную objA нельзя обратиться к полю code). Также через переменную objA, которая ссылается на объект подкласса, можно вызвать метод show() (команда objA.show()). Важно то, что в данном случае вызывается переопределенная версия метода show(). Для сравнения в программе использована команда objB.show(), которой метод show() вызывается из того же объекта, что в предыдущем случае, но теперь через переменную objB.

НА ЗАМЕТКУ

Таким образом, версия метода (переопределяемого в подклассе), которая вызывается через переменную суперкласса, определяется объектом, на который ссылается переменная. Если переменная ссылается на объект суперкласса, то вызывается версия метода, описанная в суперклассе. Если переменная ссылается на объект подкласса, то вызывается версия метода, описанная в подклассе.

Резюме

Коллектив большой, народ квалифицированный, работа проделана большая. У меня лично сомнений нет — это дело так не пойдёт.

Из к/ф «Карнавальная ночь»

- Новый класс можно создавать на основе уже существующего класса путем наследования. Класс, на основе которого создается новый класс, называется суперклассом. Класс, создаваемый на основе суперкласса, называется подклассом.
- При описании подкласса после имени класса указывается ключевое слово extends, а вслед за ним — имя суперкласса. Поля и методы суперкласса (если они не закрытые) наследуются в подклассе.
- При описании конструктора подкласса в теле конструктора первой указывается команда вызова конструктора суперкласса. Команда выглядит так: после ключевого слова super в круглых скобках перечисляются аргументы, которые передаются конструктору суперкласса. Если в теле конструктора подкласса явно команду вызова конструктора суперкласса не указать, то по умолчанию вызывается

версия конструктора суперкласса без аргументов (если такая версия конструктора в суперклассе имеется).

- Методы, наследуемые в подклассе, могут переопределяться. В таком случае в подклассе метод описывается в явном виде. Переопределение и перегрузка методов могут использоваться одновременно.
- Ключевое слово super, помимо использования в инструкции вызова конструктора суперкласса, может использоваться для вызова тех версий методов, что описаны в суперклассе и переопределены в подклассе. В таком случае ключевое слово super указывается перед именем метода и отделяется от него точкой. Аналогично можно обращаться к полям, объявленным в суперклассе и «замещенным» полями в подклассе с таким же именем.
- Ключевое слово this является ссылкой на объект, из которого вызывается метод (в конструкторе обозначает создаваемый объект). Также ключевое слово this используется при вызове в конструкторе иной версии конструктора того же класса. В этом случае в круглых скобках после ключевого слова this указываются аргументы, которые передаются конструктору. Инструкция с ключевым словом this, которой вызывается другая версия конструктора класса, должна быть первой командой в теле конструктора.
- Если в классе переопределить метод toString(), то объект такого класса в случае необходимости будет автоматически преобразовываться к тестовому формату, причем соответствующее текстовое значение является результатом метода toString().
- Переменная суперкласса может ссылаться на объект подкласса. Через такую переменную доступ есть только к тем членам объекта подкласса, которые объявлены в суперклассе.
- Методы и конструкторы являются виртуальными в том смысле, что версии переопределяемых в подклассах методов определяются не по типу объектной переменной, через которую вызываются, а по типу объекта, на который ссылается переменная.

Глава 7
АБСТРАКТНЫЕ КЛАССЫ И ИНТЕРФЕЙСЫ

Необходимо приложить игру фантазии, чтобы в соответствии со сметой провести наше мероприятие, так сказать, на высоком уровне.

Из к/ф «Карнавальная ночь»

Далее мы кратко рассмотрим *абстрактные классы* и более подробно — *интерфейсы*. Эти «конструкции» идеологически близки между собой и играют важную роль при решении многих прикладных задач, включая создание приложений с графическим интерфейсом.

Абстрактные классы и методы

— Пойдем простым логическим ходом.
— Пойдем вместе.

Из к/ф «Ирония судьбы, или С легким паром!»

При описании метода в классе указывается сигнатура метода (тип результат, название метода и список аргументов), а также тело метода с программным кодом, выполняемым при вызове метода. Но существует возможность ограничиться объявлением метода — указать его сигнатуру, но не описывать тело метода. Такие методы называются *абстрактными*. Каждый абстрактный метод описывается с ключевым словом abstract.

Класс, в котором есть хотя бы один абстрактный метод, также называется *абстрактным*. Абстрактный класс описывается с ключевым словом abstract. Более того, если класс просто описать с ключевым словом abstract, то он будет абстрактным, даже если в нем нет абстрактных методов.

Поскольку абстрактный класс содержит, по определению, абстрактные методы, то на основе абстрактного класса нельзя создать объект.

Объяснение простое и состоит в том, что такой объект, будь он создан, содержал бы методы, программный код которых не определен. Как бы там ни было, но факт остается фактом — на основе абстрактного класса объекты не создают. Возникает естественный вопрос: а для чего тогда нужны абстрактные классы? Обычно их используют при наследовании. Другими словами, на основе абстрактного класса создаются, путем наследования, подклассы, в которых определяются абстрактные методы. В таком случае абстрактный класс играет роль некоторого «каркаса» или «шаблона», на основе которого создаются подклассы.

НА ЗАМЕТКУ

Если в подклассе, который создается на основе абстрактного суперкласса, некоторые абстрактные методы оставить не описанными, то такой подкласс также будет абстрактным.

Также следует учесть одно немаловажное обстоятельство: хотя на основе абстрактного класса нельзя создать объект, но зато можно объявить объектную переменную абстрактного класса. Такая переменная может ссылаться на объект любого подкласса, созданного на основе данного абстрактного класса. Похожая ситуация реализуется в программе, код которой представлен в листинге 7.1.

Мы создаем несколько классов, которые в определенном смысле описывают разные геометрические фигуры (равносторонний треугольник, квадрат и круг). Мы предполагаем наличие у геометрических фигур таких характеристик, как цвет («раскрашенная» фигура), линейный размер (для треугольника и квадрата — длина стороны, для круга — радиус) и площадь.

ДЕТАЛИ

Если равносторонний треугольник имеет стороны длины R, то площадь треугольника дается выражением $\sqrt{3}/4R^2$. Для квадрата со стороной R площадь вычисляется как R^2. Площадь круга радиуса R вычисляется выражением πR^2, где иррациональная постоянная $\pi \approx 3{,}141592$. Все три случая можно свести к формуле kR^2, где параметр $k = \sqrt{3}/4$ для треугольника, $k = 1$ для квадрата и $k = \pi$ для круга.

В программе описывается абстрактный класс с названием ColoredFigure, на основе которого путем наследования создаются классы Triangle (класс для

описания треугольника), Square (класс для описания квадрата) и Circle (класс для описания круга). В главном методе программы создаются объекты производных классов и показано, как получить доступ к объектам через объектные переменные подклассов и через объектную переменную абстрактного суперкласса. Теперь рассмотрим программный код примера.

Листинг 7.1. Программный код проекта UsingAbstractClassApplication

```
// Абстрактный суперкласс:
abstract class ColoredFigure{
  // Текстовое поле (цвет):
  String color;
  // Целочисленное поле (размер):
  int size;
  // Конструктор:
  ColoredFigure(String clr,int s){
    // Присваивание значений полям:
    color=clr;
    size=s;
  }
  // Метод для отображения информации об объекте:
  void show(){
    // Цвет и название фигуры:
    System.out.println("Фигура: "+color+" "+getName());
    // Характерный размер:
    System.out.println("Характерный размер ("+getSizeName()+"): "+size);
    // Площадь:
    System.out.printf("Площадь: %.3f\n",getArea());
    // Импровизированная "линия" из "звездочек":
    String line="";
    for(int k=1;k<=30;k++){
      line+="*";
    }
    System.out.println(line);
  }
```

```
  // Метод результатом возвращает название для
  // параметра, определяющего характерный размер фигуры:
  String getSizeName(){
    return "сторона";
  }
  // Абстрактные методы:
  abstract String getName(); // Название фигуры
  abstract double getArea(); // Площадь фигуры
}
// Подкласс (фигура — треугольник):
class Triangle extends ColoredFigure{
  // Конструктор:
  Triangle(String clr,int s){
    // Вызов конструктора суперкласса:
    super(clr,s);
  }
  // Описание абстрактного метода из суперкласса,
  // возвращающего результатом название фигуры:
  String getName(){
    return "треугольник";
  }
  // Описание абстрактного метода из суперкласса,
  // возвращающего результатом площадь фигуры:
  double getArea(){
    double k=Math.sqrt(3)/4;
    return size*size*k;
  }
}
// Подкласс (фигура — квадрат):
class Square extends ColoredFigure{
   // Конструктор:
   Square(String clr,int s){
     // Вызов конструктора суперкласса:
```

```
    super(clr,s);
  }
  // Описание абстрактного метода из суперкласса,
  // возвращающего результатом название фигуры:
  String getName(){
    return "квадрат";
  }
  // Описание абстрактного метода из суперкласса,
  // возвращающего результатом площадь фигуры:
  double getArea(){
    double k=1;
    return size*size*k;
  }
}
// Подкласс (фигура — круг):
class Circle extends ColoredFigure{
  // Конструктор:
  Circle(String clr,int s){
  // Вызов конструктора суперкласса:
    super(clr,s);
  }
  // Описание абстрактного метода из суперкласса,
  // возвращающего результатом название фигуры:
  String getName(){
    return "круг";
  }
  // Описание абстрактного метода из суперкласса,
  // возвращающего результатом площадь фигуры:
  double getArea(){
    double k=Math.PI;
    return size*size*k;
  }
  // Переопределение метода, возвращающего результатом
```

```
  // название для параметра, определяющего характерный
  // размер фигуры:
  String getSizeName(){
    return "радиус";
  }
}
// Главный класс:
class UsingAbstractClassDemo{
  // Главный метод:
  public static void main(String[] args){
    // Создание объектов:
    Triangle T=new Triangle("красный",2); // Треугольник
    Square S=new Square("черный",3);     // Квадрат
    Circle C=new Circle("желтый",1);     // Круг
    // Отображение информации об объектах через
    // объектные переменные подклассов:
    System.out.println("Использование объектных переменных подкласса");
    T.show();
    S.show();
    C.show();
    // Объектная переменная абстрактного суперкласса:
    ColoredFigure F;
    // Отображение информации об объектах через
    // объектную переменную абстрактного суперкласса:
    System.out.println("Использование объектной переменной абстрактного суперкласса");
    F=T;     // Треугольник
    F.show();
    F=S;     // Квадрат
    F.show();
    F=C;     // Круг
    F.show();
  }
}
```

Результат выполнения программы такой, как показано ниже.

Результат выполнения программы (из листинга 7.1)

```
Использование объектных переменных подкласса
Фигура: красный треугольник
Характерный размер (сторона): 2
Площадь: 1,732
******************************
Фигура: черный квадрат
Характерный размер (сторона): 3
Площадь: 9,000
******************************
Фигура: желтый круг
Характерный размер (радиус): 1
Площадь: 3,142
******************************
Использование объектной переменной абстрактного суперкласса
Фигура: красный треугольник
Характерный размер (сторона): 2
Площадь: 1,732
******************************
Фигура: черный квадрат
Характерный размер (сторона): 3
Площадь: 9,000
******************************
Фигура: желтый круг
Характерный размер (радиус): 1
Площадь: 3,142
******************************
```

В абстрактном классе ColoredFigure описаны два поля: в текстовое поле color, как предполагается, будет записываться название для цвета, а в целочисленное поле size будет записываться значение для линейного размера фигуры (для простоты считаем, что характеристика размера выражается

целым числом). Поля заполняются при вызове конструктора, аргументами которому передаются два аргумента (значения полей).

В классе ColoredFigure описан метод show(), которым в окне вывода отображается общая информация об объекте. В частности, в теле метода show() отображаются значения полей color и size, а также вызываются методы getName(), getSizeName() и getArea(). Методы getName() и getArea() в классе ColoredFigure объявлены как абстрактные. Предполагается, что методом getName() возвращается текстовое значение с названием геометрической фигуры, а методом getArea() вычисляется площадь геометрической фигуры. Эти методы описываются в подклассах. Метод getSizeName() результатом возвращает текстовое значение, отображающее название параметра, определяющего характерный размер фигуры. Этот метод явно описан в классе ColoredFigure, но переопределяется в одном из подклассов (более конкретно, в классе Circle).

ДЕТАЛИ

В теле метода show() значение для площади фигуры отображается командой System.out.printf("Площадь: %.3f\n",getArea()). В данном случае мы используем метод printf(). Аргументов у метода два: текст "Площадь: %.3f\n" с инструкцией форматирования %.3f и собственно значение для площади фигуры, вычисляемое инструкцией getArea(). Значение, возвращаемое методом getArea() вставляется при отображении строки "Площадь: %.3f\n" в том месте, где размещена инструкция %.3f. В самой инструкции символ % является индикатором инструкции форматирования, литера f означает, что отображается действительное число, а выражение .3 (точка и число 3) означает, что в дробной части числа отображается три цифры.

На основе абстрактного класса ColoredFigure создаются подклассы Triangle, Square и Circle, в каждом из которых описываются методы getName() и getArea(), а в классе Circle еще и переопределяется метод getSizeName().

НА ЗАМЕТКУ

При описании метода getArea() в разных классах использована статическая константа PI из класса Math со значением постоянной $\pi \approx 3{,}141592$, а также статический метод sqrt() класса Math, используемый для вычисления квадратного корня.

В главном методе программы создается три объекта производных классов. Из каждого объекта вызывается метод show(). Причем делается это

дважды — сначала через объектную переменную соответствующего подкласса, а затем через объектную переменную абстрактного суперкласса ColoredFigure.

НА ЗАМЕТКУ

Таким образом, объектная переменная суперкласса, несмотря на то что он абстрактный, позволяет получать доступ к объектам подклассов (созданных на основе данного абстрактного суперкласса).

Интерфейсы

— Скажите, доктор Ватсон, вы понимаете всю важность моего открытия?
— Да, как эксперимент это интересно. Но какое практическое применение?
— Господи, именно практическое!

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

В известном смысле *интерфейс* напоминает абстрактный класс, но только в совершенно «абстрактном» и немного «урезанном» виде: интерфейс содержит объявление методов и/или статических констант. Интерес в первую очередь представляют, конечно, методы. Еще раз подчеркнем, что в интерфейсе они только объявляются.

НА ЗАМЕТКУ

В Java 8 интерфейс может содержать не только объявление методов, но еще и описание методов — так называемые версии методов по умолчанию. Но мы этот вопрос (создание реализаций методов по умолчанию) рассмотрим немного позже.

Интерфейсы напоминают абстрактные классы, и используются похожим образом. Интерфейсы используются путем *реализации* их в классах. Процесс реализации интерфейса в классе напоминает наследование абстрактного суперкласса в подклассе. В случае с интерфейсами класс, который реализует интерфейс, должен содержать описание всех методов, объявленных в интерфейсе. Важное обстоятельство связано с тем, что один и тот же класс может реализовать одновременно несколько интерфейсов.

НА ЗАМЕТКУ

В Java запрещено множественное наследование: у подкласса может быть один и только один суперкласс. Это серьезное ограничение (например, в языке C++ класс может наследовать одновременно несколько классов). С другой стороны, множественное наследование представляет собой гибкий и эффективный механизм. Как некоторый «компромисс» в данном вопросе можно рассматривать концепцию интерфейсов.

Реализация интерфейса

Интерфейс описывается практически так же, как и класс. Только вместо ключевого слова class используется ключевое слово interface. Также следует сделать поправку на то, что методы в интерфейсе (если речь не идет о методах с кодом по умолчанию) объявляются, но не описываются. Шаблон описания интерфейса выглядит так:

```
interface имя_интерфейса{
  // Объявление полей и методов
}
```

Кроме объявления методов, в интерфейсе можно объявлять статические константные поля. Такие поля объявляются как обычные поля со значениями, но автоматически интерпретируются, как если бы они были описаны с ключевыми словами static и final. Методы, объявленные в интерфейсе, являются public-методами (хотя при объявлении методов в интерфейсе ключевое слово public не используется). Далее приведен небольшой пример объявления интерфейса:

```
interface MyInterface{
  int NUMBER=100;
  int getNumber(double x);
  char getSymbol(int n);
}
```

Интерфейс называется MyInterface. В интерфейсе объявлена статическая целочисленная константа NUMBER со значением 100 и еще объявлены два метода:

- метод getNumber() возвращает значение типа int, а аргументом передается значение типа double;
- метод getSymbol() возвращает значение типа char, а аргументом методу передается значение типа int.

В интерфейсе мы только объявляем методы, но не описываем их.

Интерфейс реализуется в классе. В описании класса, реализующего интерфейс, после имени класса через ключевое слово implements указывается имя реализуемого интерфейса. Например, если исходить из того, интерфейс MyInterface описан так, как показано выше, то описание класса MyClass, реализующий этот интерфейс, могло бы выглядеть следующим образом:

```
class MyClass implements MyInterface{
  public int getNumber(double x){
    return (int)x;
  }
  public char getSymbol(int n){
    return (char)('A'+n);
  }
}
```

В классе явно описаны методы getNumber() и getSymbol(). Методы описываются с ключевым словом public (обязательное условие). У каждого объекта класса MyClass будут такие методы.

НА ЗАМЕТКУ

Методом getNumber() возвращается целая часть от double-аргумента метода. Методом getSymbol() возвращается символ, код которого получается прибавлением к коду символа 'A' значения целочисленного аргумента метода.

Также у класса MyClass есть статическое константное поле NUMBER (со значением 100), «полученное» из интерфейса MyInterface. В листинге 7.2 представлена программа, в которой иллюстрируется использование интерфейсов.

Листинг 7.2. Программный код проекта UsingInterfaceApplication

```
// Интерфейс:
interface MyInterface{
```

```
  // Статическая константа:
  int NUMBER=100;
  // Объявление методов:
  int getNumber(double x);
  char getSymbol(int n);
}
// Класс реализует интерфейс:
class MyClass implements MyInterface{
  // Описание методов:
  public int getNumber(double x){
    return (int)x;
  }
  public char getSymbol(int n){
    return (char)('A'+n);
  }
}
// Главный класс:
class UsingInterfaceDemo{
  public static void main(String[] args){
    MyClass obj=new MyClass();
    System.out.println("Статическая константа: "+MyClass.NUMBER);
    System.out.println("Целое число: "+obj.getNumber(12.5));
    System.out.println("Символ: "+obj.getSymbol(3));
  }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 7.2)

```
Статическая константа: 100
Целое число: 12
Символ: D
```

Пример достаточно простой и результаты особых комментариев не требуют.

Класс может реализовывать сразу несколько интерфейсов. Если класс реализует несколько интерфейсов, то в описании класса после ключевого слова implements через запятую перечисляются интерфейсы, реализуемые в классе. В классе должны быть описаны все методы из всех реализуемых интерфейсов. Рассмотрим небольшой пример, представленный в листинге 7.3. В представленной там программе описывается два интерфейса First и Second, которые реализуются в классе MyClass.

Листинг 7.3. Программный код проекта UsingInterfacesApplication

```
// Первый интерфейс:
interface First{
  void hello();
}
// Второй интерфейс:
interface Second{
  void hi();
}
// Класс реализует два интерфейса:
class MyClass implements First, Second{
  // Описание метода из первого интерфейса:
  public void hello(){
    System.out.println("Метод из интерфейса First");
  }
  // Описание метода из второго интерфейса:
  public void hi(){
    System.out.println("Метод из интерфейса Second");
  }
}
// Главный класс:
class UsingInterfacesDemo{
  public static void main(String[] args){
    // Создание объекта:
```

```
    MyClass obj=new MyClass();
    // Вызов методов из объекта:
    obj.hello();
    obj.hi();
  }
}
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 7.3)

```
Метод из интерфейса First
Метод из интерфейса Second
```

В интерфейсе First объявлен метод hello(), а в интерфейсе Second объявлен метод hi(). В классе MyClass описывается как метод hello(), так и метод hi(). В главном методе программы создается объект класса MyClass, и из него вызываются методы hello() и hi().

Если в классе реализуется больше двух интерфейсов, то подход остается тем же: реализуемые интерфейсы перечисляются через запятую после ключевого слова implements в описании класса, а в самом классе описываются все методы из всех интерфейсов.

Интерфейсные переменные

Важная особенность, связанная с использованием интерфейсов, состоит в том, что мы можем объявить *интерфейсную переменную* — переменную, «тип» которой определяется названием интерфейса. Такая переменная может ссылаться на объект класса, реализующего данный интерфейс. Небольшой пример в листинге 7.4 дает представление о том, как используются интерфейсные переменные.

Листинг 7.4. Программный код проекта UsingInterfaceVarsApplication

```
// Интерфейс:
interface Base{
  // Объявление метода:
  void show();
```

```
}
// Класс реализует интерфейс Base:
class Alpha implements Base{
  // Текстовое поле:
  String word;
  // Конструктор:
  Alpha(String txt){
    word=txt;
  }
  // Описание метода из интерфейса:
  public void show(){
    System.out.println("Объект класса Alpha");
    System.out.println("Текстовое поле: "+word);
  }
}
// Класс реализует интерфейс Base:
class Bravo implements Base{
  // Целочисленное поле:
  int number;
  // Конструктор:
  Bravo(int n){
    number=n;
  }
  // Описание метода из интерфейса:
  public void show(){
    System.out.println("Объект класса Bravo");
    System.out.println("Целочисленное поле: "+number);
  }
}
// Главный класс:
class UsingInterfaceVarsDemo{
  public static void main(String[] args){
```

```
    // Интерфейсная переменная:
    Base ref;
    // Объект класса Alpha:
    ref=new Alpha("текст");
    ref.show();
    // Объект класса Bravo:
    ref=new Bravo(123);
    ref.show();
  }
}
```

Ниже показано, как выглядит результат выполнения программы:

Результат выполнения программы (из листинга 7.4)

```
Объект класса Alpha
Текстовое поле: текст
Объект класса Bravo
Целочисленное поле: 123
```

В данном примере мы описываем интерфейс Base, в котором объявлен всего один метод show(). У метода нет аргументов, и метод не возвращает результат. Классы Alpha и Bravo реализуют интерфейс Base. В классе Alpha описано текстовое поле word, конструктор и метод show(). В классе Bravo описано целочисленное поле number, конструктор и метод show(). В каждом из классов метод show() описан так, что отображает имя класса и значение поля.

В главном методе программы командой Base ref объявляется интерфейсная переменная ref. Эта переменная может ссылаться на объект любого класса, который реализует интерфейс Base. Таким образом, поскольку и класс Alpha, и класс Bravo, реализуют интерфейс Base, то интерфейсная переменная ref может ссылаться на объекты классов Alpha и Bravo. При этом действует ограничение такое же, как и при получении доступа к объектам подкласса через переменную суперкласса: доступ имеется лишь к тем методам объекта, которые объявлены в реализуемом интерфейсе.

При выполнении команды ref=new Alpha("текст") создается объект класса Alpha, а ссылка на объект записывается в интерфейсную переменную ref.

При выполнении команды ref.show() из созданного объекта вызывается метод show(). Затем командой ref=new Bravo(123) создается объект класса Bravo, и ссылка на объект записывается в переменную ref. Теперь при выполнении команды ref.show() метод show() вызывается из объекта класса Bravo.

Методы с кодом по умолчанию

Новшеством в языке Java (в версии Java 8) является возможность не только объявлять методы в интерфейсе, но и задавать код таких методов. Проще говоря, метод в интерфейсе можно не только объявить, но и *описать*. Если метод описан в интерфейсе, то в классе, реализующем соответствующий интерфейс, метод можно не описывать. В таком случае для метода используется код из интерфейса. Фактически в интерфейсе для метода задается код, используемый по умолчанию в случае, если в классе код метода явно не определен.

Методы с кодом по умолчанию описываются в интерфейсе с ключевым словом default. Небольшой пример, в котором использован интерфейс с методом, у которого есть код по умолчанию, приведен в листинге 7.5.

Листинг 7.5. Программный код проекта UsingDefaultMethodsApplication

```
// Интерфейс с методом, имеющим код по умолчанию:
interface Base{
  // Метод с кодом по умолчанию:
  default void show(String txt){
    System.out.println("Интерфейс Base: "+txt);
  }
  // Объявление метода:
  void hello();
}
// Класс реализует интерфейс Base:
class Alpha implements Base{
  // Описание обычного метода:
  public void hello(){
    System.out.println("Объект класса Alpha");
  }
```

```
  // Описание метода с кодом по умолчанию:
  public void show(String txt){
    System.out.println("Класс Alpha: "+txt);
  }
}
// Класс реализует интерфейс Base:
class Bravo implements Base{
  // Описание обычного метода:
  public void hello(){
    System.out.println("Объект класса Bravo");
  }
}
// Главный класс:
class UsingDefaultMethodsDemo{
  public static void main(String[] args){
    // Интерфейсная переменная:
    Base ref;
    // Объект класса Alpha:
    Alpha objA=new Alpha();
    // Вызов методов через объектную переменную:
    objA.hello();
    objA.show("объектная переменная objA");
    // Интерфейсной переменной присваивается ссылка
    // на объект класса Alpha:
    ref=objA;
    // Вызов метода через интерфейсную переменную:
    ref.show("интерфейсная переменная ref");
    // Объект класса Bravo:
    Bravo objB=new Bravo();
    // Вызов методов через объектную переменную:
    objB.hello();
    objB.show("объектная переменная objB");
```

```
    // Интерфейсной переменной присваивается ссылка
    // на объект класса Bravo:
    ref=objB;
    // Вызов метода через интерфейсную переменную:
    ref.show("интерфейсная переменная ref");
  }
}
```

Результат выполнения программы такой, как показано ниже.

Результат выполнения программы (из листинга 7.5)

```
Объект класса Alpha
Класс Alpha: объектная переменная objA
Класс Alpha: интерфейсная переменная ref
Объект класса Bravo
Интерфейс Base: объектная переменная objB
Интерфейс Base: интерфейсная переменная ref
```

В данном случае в интерфейсе Base объявлен метод hello() без аргументов и не возвращающий результат. Еще один метод show() в интерфейсе Base описан с кодом по умолчанию — описание метода начинается с ключевого слова default, и явно указано тело метода с кодом: при вызове метода в окне вывода отображается текст "Интерфейс Base: " и значение текстового аргумента, переданного аргументом методу.

В классе Alpha реализуется интерфейс Base. В классе описываются методы hello() и show() (последний фактически переопределяется). В классе Bravo описывается только метод hello().

В главном методе программы создаются два объекта: один класса Alpha, другой класса Bravo. Доступ к методам получаем через объектные переменные и через интерфейсную переменную. В любом случае при вызове метода show() из объекта класса Alpha используется версия метода, описанная в данном классе. При вызове метода show() из объекта класса Bravo вызывается версия метода show(), описанная в интерфейсе Base. Причина в том, что в классе Bravo метод show() явно не описан, поэтому по умолчанию используется тот код, что определен для метода show() в интерфейсе Base.

Возможность задавать в интерфейсах для методов код по умолчанию, с одной стороны, удобна и расширяет спектр возможностей по созданию гибкого программного кода. С другой стороны, здесь возможны нетривиальные ситуации.

Представим себе следующую ситуацию. Допустим, имеется два интерфейса (назовем их First и Second), в каждом из которых объявлен метод с одной и той же сигнатурой — для определенности пускай метод называется hello(), не имеет аргументов и не возвращает результат. Далее, пускай класс MyClass реализует оба эти интерфейса. Возможны такие ситуации:

- методы только объявлены в обоих интерфейсах;
- метод объявлен в одном интерфейсе и описан (с кодом по умолчанию) в другом интерфейсе;
- метод описан (с кодом по умолчанию) в каждом из интерфейсов.

В первом случае особых проблем не возникает: в классе MyClass достаточно описать один раз метод hello(). Во втором случае метод можно не описывать: для метода будет использован код по умолчанию из того интерфейса, в котором он описан. В третьем случае метод придется описать в классе, поскольку в противном случае возникает ошибка на этапе компиляции: если метод описан в обоих интерфейсах, то непонятно какую версию кода по умолчанию использовать в классе.

ДЕТАЛИ

В некоторых случаях приходится в явном виде вызывать версию метода, описанную в интерфейсе. Это можно сделать. Ссылка на версию метода из интерфейса выполняется в таком формате: указывается имя интерфейса, точка, ключевое слово super, точка и, собственно, инструкция вызова метода. Например, если нужно вызвать версию метода hello() из интерфейса First, то соответствующая инструкция выглядит как First.super.hello(). Аналогично, инструкция Second.super.hello() означает вызов версии метода hello() из интерфейса Second.

Рассмотрим небольшой пример, представленный в листинге 7.6.

Листинг 7.6. Программный код проекта MoreDefaultMethodsApplication

```
// Первый интерфейс:
interface First{
  default void hello(){
```

```
    System.out.println("Метод из интерфейса First");
  }
}
// Второй интерфейс:
interface Second{
  default void hello(){
    System.out.println("Метод из интерфейса Second");
  }
}
// Класс реализует два интерфейса:
class MyClass implements First, Second{
  // Описание метода:
  public void hello(){
    // Вызов версии метода из интерфейса First:
    First.super.hello();
    // Вызов версии метода из интерфейса Second:
    Second.super.hello();
  }
}
// Главный класс:
class MoreDefaultMethodsDemo{
  public static void main(String[] args){
    // Создание объекта:
    MyClass obj=new MyClass();
    // Вызов метода:
    obj.hello();
  }
}
```

Ниже показано, как выглядит результат выполнения программы.

Результат выполнения программы (из листинга 7.6)

```
Метод из интерфейса First
Метод из интерфейса Second
```

В этом примере в интерфейсах First и Second описан метод с названием hello(). Класс MyClass реализует оба эти интерфейса. В классе MyClass метод hello() описывается, и при описании метода использованы инструкции First.super.hello() и Second.super.hello(), которыми последовательно вызываются версии метода hello() из интерфейсов First и Second.

В главном методе программы создается объект класса MyClass, из которого вызывается метод hello(). Результат, думается, особых комментариев не требует.

Расширение интерфейсов

Наследование применимо не только к классам, но и к интерфейсам: один интерфейс может наследовать другой интерфейс. В результате интерфейс-наследник «получает» описание методов (и статических констант) из наследуемого интерфейса. Технически наследование интерфейсов (которое называется *расширением* интерфейса) реализуется так же, как и наследование классов. В частности, в описании интерфейса-наследника после его имени указывается ключевое слово extends, после которого указывается имя наследуемого интерфейса. Простой пример, в котором иллюстрируется расширение интерфейсов, представлен в листинге 7.7.

Листинг 7.7. Программный код проекта ExtendingInterfaceApplication

```
// Наследуемый интерфейс:
interface First{
  // Метод с кодом по умолчанию:
  default void alpha(){
    System.out.println("Интерфейс First: метод alpha()");
  }
  // Метод с кодом по умолчанию:
  default void bravo(){
    System.out.println("Интерфейс First: метод bravo()");
  }
  // Метод с кодом по умолчанию:
  default void charlie(){
    System.out.println("Интерфейс First: метод charlie()");
  }
```

```
  // Метод без кода по умолчанию:
  void delta();
}
// Интерфейс-наследник:
interface Second extends First{
  // Объявление метода:
  void bravo();
  // Метод с кодом по умолчанию:
  default void charlie(){
    System.out.println("Интерфейс Second: метод charlie()");
  }
  // Объявление метода:
  void echo();
}
// Класс реализует интерфейс:
class MyClass implements Second{
  // Описание методов:
  public void bravo(){
    System.out.println("Класс MyClass: метод bravo()");
  }
  public void delta(){
    System.out.println("Класс MyClass: метод delta()");
  }
  public void echo(){
    System.out.println("Класс MyClass: метод echo()");
  }
}
// Главный класс:
class ExtendingInterfaceDemo{
  public static void main(String[] args){
    // Создание объекта:
    MyClass obj=new MyClass();
    // Вызов методов:
```

```
    obj.alpha();
    obj.bravo();
    obj.charlie();
    obj.delta();
    obj.echo();
  }
}
```

В результате выполнения программы получаем следующее:

Результат выполнения программы (из листинга 7.7)

```
Интерфейс First: метод alpha()
Класс MyClass: метод bravo()
Интерфейс Second: метод charlie()
Класс MyClass: метод delta()
Класс MyClass: метод echo()
```

Мы использовали довольно простую схему. В частности, в программе описан интерфейс First. В интерфейсе объявлены методы alpha(), bravo(), charlie() и delta(). Методы не возвращают результат, и у них нет аргументов. Более того, методы alpha(), bravo() и charlie() имеют код, используемый по умолчанию. Код очень простой: при вызове метода выводится сообщение с названием интерфейса First и названием метода. Метод delta() просто объявлен.

НА ЗАМЕТКУ

Прочие методы в интерфейсе Second и классе MyClass описаны аналогичным образом: они не возвращают результат, у них нет аргументов, а при вызове метода отображается название интерфейса или класса, в котором описан метод, и название метода.

Класс Second наследует (расширяет) интерфейс First. Это означает, что интерфейс Second автоматически включает в себя объявления (и определения) методов из интерфейса First. Но в интерфейсе Second есть и «собственные» определения. А именно в интерфейсе Second объявлен метод bravo() (с такой же сигнатурой, как в интерфейсе First), описан метод charlie() и объявлен метод echo(). Интерфейс Second, в свою очередь, реализуется в классе MyClass.

В классе MyClass описаны методы bravo(), delta() и echo(). Что получается в итоге? Есть всего пять методов. Разберем ситуацию с каждым из них.

Метод alpha() описан с кодом по умолчанию в интерфейсе First, явно не описывается и не объявляется в интерфейсе Second и классе MyClass. В итоге в классе MyClass используется версия метода alpha() с кодом по умолчанию из интерфейса First.

Метод bravo() описан с кодом по умолчанию в интерфейсе First. Он наследуется в интерфейсе Second, но там объявлен метод с такой же сигнатурой и без кода по умолчанию. Поэтому в интерфейсе Second метод bravo() становится абстрактным. В результате в классе MyClass метод bravo() необходимо описать — в противном случае класс был бы абстрактным. Другими словами, код по умолчанию для метода bravo() из интерфейса First до класса MyClass «не доходит». И все из-за объявления метода bravo() в интерфейсе Second.

Метод charlie() описан с кодом по умолчанию в интерфейсе First. Но в интерфейсе Second код по умолчанию для данного метода переопределяется. Именно код по умолчанию метода charlie() из интерфейса Second используется в классе MyClass.

Метод delta() объявлен в интерфейсе First, а метод echo() объявлен в интерфейсе Second. Методы только объявлены, и код по умолчанию для них не задан. Поэтому в классе MyClass оба метода следует описать (или объявить класс MyClass как абстрактный).

Еще одна вариация на тему предыдущего примера представлена в листинге 7.8.

Листинг 7.8. Программный код проекта MoreExtendingInterfaceApplication

```
// Первый интерфейс:
interface First{
  // Описание метода:
  default void alpha(){
    System.out.println("Интерфейс First: метод alpha()");
  }
}
// Второй интерфейс:
interface Second extends First{
  // Описание методов:
```

```
  default void alpha(){
    // Вызов версии метода из интерфейса First:
    First.super.alpha();
    System.out.println("Интерфейс Bravo: метод alpha()");
  }
  default void bravo(){
    System.out.println("Интерфейс Bravo: метод bravo()");
  }
}
// Класс реализует интерфейс:
class MyClass implements Second{
  // Описание метода:
  public void bravo(){
    // Вызов версии метода из интерфейса Second:
    Second.super.bravo();
    System.out.println("Класс MyClass: метод bravo()");
  }
}
// Главный класс:
class MoreExtendingInterfaceDemo{
  public static void main(String[] args){
    // Создание объекта:
    MyClass obj=new MyClass();
    // Вызов методов:
    obj.alpha();
    obj.bravo();
  }
}
```

Ниже показан результат выполнения программы.

Результат выполнения программы (из листинга 7.8)

```
Интерфейс First: метод alpha()
Интерфейс Bravo: метод alpha()
```

```
Интерфейс Bravo: метод bravo()
Класс MyClass: метод bravo()
```

Мы описываем интерфейс First с методом alpha(). Метод имеет код по умолчанию (при вызове метода отображается имя класса и название метода). Интерфейс Second расширяет интерфейс First, причем в интерфейсе Second описывается метод bravo() и переопределяется метод alpha(). При описании кода метода alpha() в интерфейсе Second использована инструкция First.super.alpha(), означающая вызов версии метода alpha() из интерфейса First.

Класс MyClass наследует (реализует) интерфейс Second. В классе MyClass описывается метод bravo(). В теле метода командой Second.super.bravo() вызывается версия метода bravo(), описанная в интерфейсе Second.

В главном методе программы создается объект класса MyClass, и из него вызываются методы alpha() и bravo(). Желающие могут проанализировать результат выполнения методов в контексте того, как они описывались в интерфейсах First и Second и в классе MyClass.

Наследование классов и реализация интерфейсов

Это мелочи. Но нет ничего важнее мелочей!

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Класс может наследовать суперкласс и одновременно реализовывать несколько интерфейсов. Пример такой ситуации представлен в листинге 7.9. В представленной там программе описаны два интерфейса First и Second, а также суперкласс Base. Путем реализации интерфейсов и наследования суперкласса создается класс MyClass. Чтобы рассмотреть вопрос более детально, рассмотрим программный код примера.

Листинг 7.9. Программный код проекта ExtendingAndImplementingApplication

```
// Первый интерфейс:
interface First{
  // Объявление метода:
  String getFirst();
  // Описание метода:
  default void show(){
```

```
    System.out.println("Интерфейс First: метод show()");
  }
}
// Второй интерфейс:
interface Second{
  // Объявление метода:
  String getSecond();
  // Описание метода:
  default void show(){
    System.out.println("Интерфейс Second: метод show()");
  }
}
// Суперкласс:
class Base{
  // Описание метода:
  String getBase(){
    return "Класс Base";
  }
  // Описание метода:
  void show(){
    System.out.println("Класс Base: метод show()");
  }
}
// Подкласс наследует суперкласс и реализует интерфейсы:
class MyClass extends Base implements First, Second{
  // Описание метода:
  public String getFirst(){
    return "Интерфейс First";
  }
  // Описание метода:
  public String getSecond(){
    return "Интерфейс Bravo";
  }
```

```
  // Описание метода:
  public void show(){
    System.out.println("Мы используем:");
    System.out.println(getFirst());
    System.out.println(getSecond());
    System.out.println(getBase());
    // Вызов версии метода из интерфейса First:
    First.super.show();
    // Вызов версии метода из интерфейса Second:
    Second.super.show();
    // Вызов версии метода из суперкласса Base:
    super.show();
  }
}
// Главный класс:
class ExtendingAndImplementingDemo{
  public static void main(String[] args){
    // Создание объекта подкласса:
    MyClass obj=new MyClass();
    // Вызов метода из объекта:
    obj.show();
  }
}
```

При выполнении программы получаем следующее:

Результат выполнения программы (из листинга 7.9)

```
Мы используем:
Интерфейс First
Интерфейс Bravo
Класс Base
Интерфейс First: метод show()
Интерфейс Second: метод show()
Класс Base: метод show()
```

В интерфейсе First объявлен метод getFirst() (без аргументов, возвращающий текстовый результат) и описан (с кодом по умолчанию) метод show() (без аргументов и не возвращающий результат). Аналогично в интерфейсе Second объявлен метод (без аргументов и с текстовым результатом) getSecond() и описан не возвращающий результат и не имеющий аргументов метод show(). Такой же метод show() (имеется в виду метод с такой же сигнатурой) описан в суперклассе Base. Еще в суперклассе Base описан метод getBase(). Метод вызывается без аргументов и возвращает результатом текстовое значение.

Все это «богатство» наследуется и реализуется в классе MyClass. В частности, в классе в явном виде определяются методы getFirst() (из интерфейса First) и getSecond() (из интерфейса Second). Эти методы в классе MyClass описать необходимо, поскольку они объявлены в интерфейсах, которые реализуются в классе. Ситуация с методом show() не очень тривиальная: метод с соответствующей сигнатурой описан в суперклассе Base, а также есть реализация с кодом по умолчанию метода в интерфейсах First и Second. Чтобы избежать неоднозначности, мы описываем (используя ключевое слово public) явно метод show() в классе MyClass. При этом в теле метода вызываются его версии из суперкласса Base (инструкция super.show()) и интерфейсов First и Second (инструкции First.super.show() и Second.super.show()).

В главном методе программы создается объект класса MyClass, и из этого объекта вызывается метод show(). При вызове метода вызываются, кроме прочего, версии метода из суперкласса и интерфейсов, чем и объясняется результат выполнения программы.

Резюме

Холмс, это исключено. Сразу видно, что вы мало читаете.

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

- Если метод в классе только объявлен, но не описан, то такой метод называется абстрактным. Абстрактный метод описывается с ключевым словом abstract. Класс, содержащий хотя бы один абстрактный метод, называется абстрактным классом и описывается с ключевым словом abstract. Если класс описан с ключевым словом abstract, то он является абстрактным, даже если в нем нет ни одного абстрактного метода.

- На основе абстрактного класса нельзя создать объект, но абстрактный класс можно использовать при наследовании в качестве суперкласса. В подклассе, созданном на основе абстрактного суперкласса, описываются все абстрактные методы из абстрактного суперкласса. В противном случае (если описаны не все абстрактные методы) подкласс также будет абстрактным. Объектная переменная абстрактного суперкласса может ссылаться на объекты подклассов.
- Интерфейс напоминает собой абстрактный класс. При описании интерфейса используется ключевое слово interface. В интерфейсе объявляются методы и статические константы (поля с постоянным значением). Интерфейсы реализуются в классах. Если класс реализует интерфейс (или интерфейсы), то в описании класса после имени класса указывается ключевое слово implements, после которого указывается имя реализуемого в классе интерфейса (если интерфейсов несколько, то их названия перечисляются через запятую). В классе должны быть описаны (с ключевым словом public) все методы, объявленные в интерфейсах, которые реализуются в классе — в противном случае класс будет абстрактным.
- Интерфейс может содержать не только объявление методов, но и описание методов (метод с кодом по умолчанию). Описание методов с кодом по умолчанию начинается с ключевого слова default. Если метод в интерфейсе описан с кодом по умолчанию, а в классе, который реализует такой интерфейс, метод не описан, то для метода будет использоваться код из интерфейса.
- Один интерфейс может наследовать другой интерфейс. В таком случае говорят о расширении интерфейса. Интерфейс-наследник в таком случае получает описание (и определение) методов и полей из наследуемого интерфейса. Описывается наследование (расширение) интерфейсов так же, как наследование классов — с использованием ключевого слова extends.
- Наследование класса и реализацию интерфейсов можно объединять: подкласс может наследовать суперкласс и одновременно реализовывать один или более интерфейсов. В описании подкласса после его имени через ключевое слово extends указывается имя суперкласса и через ключевое слово implements, через запятую перечисляются имена реализуемых в подклассе интерфейсов.

Глава 8
ИСПОЛЬЗОВАНИЕ КЛАССОВ И ОБЪЕКТОВ

Очень убедительно. Мы подумаем, к кому это применить.

Из к/ф «31 июня»

В этой главе мы рассмотрим некоторые вопросы, связанные с использованием классов и объектов, которым ранее мы не уделили должного внимания, но которые важны с практической точки зрения.

Методы и объекты

Хотите обмануть мага? Боже, какая детская непосредственность. Я же вижу вас насквозь.

Из к/ф «31 июня»

Мы знаем, что методам при вызове могут передаваться аргументы. Вместе с тем механизм передачи аргументов методам не самый тривиальный, поэтому имеет смысл его обсудить.

Механизм передачи аргументов методам

Базовое положение состоит в том, что при передаче аргументов методам на самом деле передается техническая, автоматически создаваемая копия аргументов. Чтобы пояснить и раскрыть суть проблемы, рассмотрим небольшой пример в листинге 8.1.

Листинг 8.1. Программный код проекта MethodArgumentsApplication

```
class MethodArgumentsDemo{
  // Статический метод с двумя целочисленными аргументами,
  // которые "обмениваются" значениями:
  static void swap(int a,int b){
```

```
    System.out.println("Выполняется метод swap()");
    // Значения аргументов метода до
    // изменения значений:
    System.out.println("Аргументы до изменения значений: "+a+" и "+b);
    // Аргументы "обмениваются" значениями:
    int x=b;
    b=a;
    a=x;
    // Значения аргументов метода после
    // изменения значений:
    System.out.println("Аргументы после изменения значений: "+a+" и "+b);
    System.out.println("Завершено выполнение метода swap()");
  }
  // Главный метод программы:
  public static void main(String[] args){
    // Целочисленные переменные:
    int m=100,n=200;
    // Значения переменных до вызова метода swap():
    System.out.println("Переменные до вызова метода swap(): "+m+" и "+n);
    // Вызов метода swap():
    swap(m,n);
    // Значения переменных после вызова метода swap():
    System.out.println("Переменные после вызова метода swap(): "+m+" и "+n);
  }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 8.1)

```
Переменные до вызова метода swap(): 100 и 200
Выполняется метод swap()
Аргументы до изменения значений: 100 и 200
Аргументы после изменения значений: 200 и 100
```

```
Завершено выполнение метода swap()
Переменные после вызова метода swap(): 100 и 200
```

В данном случае программа очень простая. В классе MethodArgumentsDemo, кроме главного метода программы, описан статический метод swap(). Метод не возвращает результат и у метода два целочисленных аргумента. При вызове метода аргументы «обмениваются» значениями — во всяком случае, мы предпринимаем такую попытку. Но перед началом «обмена» исходные значения аргументов отображаются в окне вывода. После того как аргументы «обменялись» значениями, значения аргументов опять выводятся в окно вывода.

В главном методе программы объявляются две целочисленные переменные. Эти переменные передаются аргументами методу swap(). Мы проверяем значения переменных до вызова метода swap() и после вызова метода. Что показывают результаты выполнения программы? Проверка значений аргументов при выполнении метода swap() показывает, что «обмен» значениями произошел. Но когда проверяются значения переменных после вызова метода swap(), то оказывается, что значения переменных не изменились. Причина как раз в том, что на самом деле аргументами методу передаются не сами переменные, а их копии. Поэтому когда при выполнении метода swap() мы проверяем значения аргументов и производим «обмен» значениями, в действительности проверяются значения копий аргументов. Эти же копии «обмениваются» значениями. Исходные переменные, которые формально передаются аргументами методу, остаются неизменными.

НА ЗАМЕТКУ

Копии аргументов создаются автоматически. После завершения выполнения метода копии аргументов (как и прочие локальные переменные) удаляются из памяти.

Передача аргументом объекта

Теперь рассмотрим фактически тот же пример, что и выше, но только метод swap() будет описан так, что аргументами ему передаются не целые числа, а объекты (если быть более точным, то объектные переменные, или ссылки на объекты). Рассмотрим программный код в листинге 8.2.

Листинг 8.2. Программный код проекта SwapFieldsApplication

```
// Класс для создания объектов:
class MyClass{
  // Целочисленное поле:
  int number;
  // Конструктор:
  MyClass(int n){
    number=n;
  }
}
// Класс со статическим методом swap() и главным
// методом программы:
class SwapFieldsDemo{
  // Статический метод с двумя аргументами, являющимися
  // объектами класса MyClass:
  static void swap(MyClass A,MyClass B){
    System.out.println("Выполняется метод swap()");
    // Значения поля number объектов, переданных
    // аргументами методу:
    System.out.println("Объект A: "+A.number);
    System.out.println("Объект B: "+B.number);
    // Объекты "обмениваются" значениями полей:
    int number=B.number;
    B.number=A.number;
    A.number=number;
    // Значения поля number объектов, переданных
    // аргументами методу swap()
    // (после "обмена" значениями полей):
    System.out.println("Значения полей изменены");
    System.out.println("Объект A: "+A.number);
    System.out.println("Объект B: "+B.number);
    System.out.println("Завершено выполнение метода swap()");
```

```
  }
  // Главный метод программы:
  public static void main(String[] args){
    // Создание объектов:
    MyClass A=new MyClass(100);
    MyClass B=new MyClass(200);
    // Значения поля number объектов A и B до
    // вызова метода swap():
    System.out.println("До вызова метода swap(): A.number="+A.number+" и B.number="+B.
number);
    // Вызов метода swap():
    swap(A,B);
    // Значения поля number объектов A и B после
    // вызова метода swap():
    System.out.println("После вызова метода swap(): A.number="+A.number+" и B.
number="+B.number);
  }
}
```

Ниже представлен результат выполнения программы.

Результат выполнения программы (из листинга 8.2)

```
До вызова метода swap(): A.number=100 и B.number=200
Выполняется метод swap()
Объект A: 100
Объект B: 200
Значения полей изменены
Объект A: 200
Объект B: 100
Завершено выполнение метода swap()
После вызова метода swap(): A.number=200 и B.number=100
```

Мы описываем класс MyClass с целочисленным полем number и конструктором, позволяющим задать значение поля при создании объекта. В статическом

методе swap(), описанном в классе SwapFieldsDemo, аргументами являются объекты класса MyClass. В начале выполнения метода отображаются значения полей объектов, затем полю number первого объекта присваивается значение поля number второго объекта, и наоборот, после чего новое значение поля number каждого из объектов отображается в окне вывода. В главном методе программы создается два объекта A и B класса MyClass, которые передаются аргументами методу swap(). Перед вызовом метода swap() и после вызова метода проверяется значение поля number объектов A и B. Результат выполнения программы свидетельствует о том, что значение поля number объектов A и B до вызова метода и после вызова метода swap() различны. Проще говоря, попытка в рамках метода swap() произвести «обмен» значениями для полей объектов, переданных аргументами методу, проходит успешно в том смысле, что «эффект» остается и после завершения выполнения метода. Почему так происходит? Чтобы понять «механику» дела, следует учесть, что аргументами методу swap() передаются на самом деле *объектные переменные*, которые значением содержат ссылку на объекты. Как и в случае с целочисленными аргументами (см. листинг 8.1), здесь при передаче аргументов методу swap() на самом деле передаются копии переменных, формально указанных аргументами. Значения у копий такие же, как и у «оригиналов». Но поскольку значение объектной переменной — это ссылка на объект, то копия объектной переменной ссылается на тот же объект, что и исходная переменная. В результате, несмотря на то что в теле метода swap() операции выполняются с копиями аргументов, изменения происходят с теми же самыми объектами, на которые ссылаются исходные переменные.

Если посмотреть на ситуацию шире, то легко понять, что при вызове метода нельзя изменить значения аргументов, которые передаются методу. Причина — в механизме передачи аргументов методу: аргументы передаются по значению, поэтому на самом деле метод для обработки получает копии аргументов. В данном утверждении нет противоречия с рассмотренным выше примером (см. листинг 8.2). Тем аргументами методу передавались объектные переменные, а изменения в теле метода вносились в поля объектов, на которые ссылаются аргументы метода. Поэтому об изменении значений аргументов метода там речь не шла — значениями аргументов являются ссылки на объекты. Зато в следующем примере выполняется попытка (отметим сразу — неудачная) изменить как раз аргументы метода, в роли которых, как и ранее, выступают объектные ссылки. В представленном далее программном коде, по сравнению с примером из листинга 8.2, принципиально изменился лишь код метода swap(): если ранее мы изменяли значения полей объектов, то теперь пытаемся изменить ссылки на объекты. Рассмотрим код

в листинге 8.3 (поскольку он во многом похож на код из листинга 8.2, то для сокращения объема кода основная часть комментариев удалена, а наиболее важное место кода выделено жирным шрифтом).

Листинг 8.3. Программный код проекта SwapObjectsApplication

```
class MyClass{
  int number;
  MyClass(int n){
    number=n;
  }
}
class SwapObjectsDemo{
  static void swap(MyClass A,MyClass B){
    System.out.println("Выполняется метод swap()");
    System.out.println("Объект A: "+A.number);
    System.out.println("Объект B: "+B.number);
    // Аргументы "обмениваются" значениями:
    MyClass X=B;
    B=A;
    A=X;
    System.out.println("Значения аргументов изменены");
    System.out.println("Объект A: "+A.number);
    System.out.println("Объект B: "+B.number);
    System.out.println("Завершено выполнение метода swap()");
  }
  public static void main(String[] args){
    MyClass A=new MyClass(100);
    MyClass B=new MyClass(200);
    System.out.println("До вызова метода swap(): A.number="+A.number+"
        и B.number="+B.number);
    swap(A,B);
    System.out.println("После вызова метода swap(): A.number="+A.number+"
        и B.number="+B.number);
  }
}
```

Результат выполнения программы будет таким:

Результат выполнения программы (из листинга 8.3)

```
До вызова метода swap(): A.number=100 и B.number=200
Выполняется метод swap()
Объект A: 100
Объект B: 200
Значения аргументов изменены
Объект A: 200
Объект B: 100
Завершено выполнение метода swap()
После вызова метода swap(): A.number=100 и B.number=200
```

Методу swap() при вызове передаются две объектные переменные, а в процессе выполнения метода аргументы «обмениваются» значениями. Но на самом деле значениями «обмениваются» копии переменных, переданных аргументами. Поэтому после выполнения метода значения объектных переменных остаются такими же, как и было до вызова метода.

Объект как результат метода

Нередко результатом метода должен возвращаться объект. Схема, по которой результатом метода возвращается объект, сводится к тому, что при выполнении метода создается объект, а результатом метода возвращается ссылка на этот объект, которая «технически» реализуется через объектную переменную. Простой пример описания метода, возвращающего результатом объект, представлен в листинге 8.4.

Листинг 8.4. Программный код проекта ObjectAsResultApplication

```
// Класс:
class MyClass{
  // Закрытые поля:
  private int code;
  private String name;
  // Конструктор:
  MyClass(int n,String s){
```

```
    System.out.println("Создание объекта:");
    // Присваивание значений полям
    // и отображение этих значений:
    set(n,s).show();
  }
  // Метод для присваивания значения целочисленному полю,
  // возвращающий результатом ссылку на объект:
  MyClass set(int n){
    // Присваивание значения целочисленному полю:
    code=n;
    // Результат метода:
    return this;
  }
  // Метод для присваивания значения текстовому полю,
  // возвращающий результатом ссылку на объект:
  MyClass set(String s){
    // Присваивание значения текстовому полю:
    name=s;
    // Результат метода:
    return this;
  }
  // Метод для присваивания значений полям,
  // возвращающий результатом ссылку на объект:
  MyClass set(int n,String s){
    // Присваивание значений полям и результат метода:
    return set(n).set(s);
  }
  // Метод для отображения значений полей:
  void show(){
    System.out.println("Поле code="+code);
    System.out.println("Поле name="+name);
    System.out.println("-----------------");
```

```
  }
}
// Главный класс:
class ObjectAsResultDemo{
  // Статический метод для создания объекта:
  static MyClass createObject(int n,String s){
    // Результат метода:
    return new MyClass(n,s);
  }
  // Главный метод программы:
  public static void main(String[] args){
    // Создание объекта:
    MyClass obj=createObject(100,"alpha");
    // Изменение значения целочисленного поля
    // и отображение значений полей объекта:
    obj.set(200).show();
    // Изменение значения текстового поля
    // и отображение значений полей объекта:
    obj.set("bravo").show();
    // Отображение значений полей:
    obj.show();
    // Создание объекта, изменение значений его полей
    // и вызов метода для отображения значений полей:
    createObject(300,"charlie").set(400,"delta").show();
  }
}
```

Результат выполнения метода такой:

Результат выполнения программы (из листинга 8.4)

```
Создание объекта:
Поле code=100
Поле name=alpha
```

```
----------
Поле code=200
Поле name=alpha
----------
Поле code=200
Поле name=bravo
----------
Поле code=200
Поле name=bravo
----------
Создание объекта:
Поле code=300
Поле name=charlie
----------
Поле code=400
Поле name=delta
----------
```

В классе MyClass описано два закрытых поля: текстовое и целочисленное. У класса есть конструктор с двумя аргументами и метод show(), которым в окне вывода отображаются значения полей объекта. Кроме этого, в классе описаны три версии метода set(). Метод предназначен для присваивания значений полям (соответственно, только целочисленному полю, только текстовому полю, и обоим полям). Все версии метода set() описаны так, что результатом возвращается ссылка на объект, из которого вызывается метод. Все самое интересное в классе MyClass связано именно с методом set() (разными его версиями). Поэтому метод заслуживает отдельный комментарий.

В версии метода set() с одним целочисленным аргументом сначала полю code присваивается соответствующее значение, после чего командой return this ссылка на объект, из которого вызывается метод, возвращается результатом метода. Аналогичным образом описана версия метода set() с одним текстовым аргументом. А вот код версии метода set() с двумя аргументами (целочисленным n и текстовым s) выглядит несколько иначе. В частности, в теле метода выполняется всего одна команда return set(n).

set(s), которой результатом метода возвращается значение выражения set(n).set(s). Это выражение имеет смысл. Значение выражения — это результат вызова метода set() с аргументом s из объекта, который возвращается при вызове метода set() с аргументом n. Но результатом выражения set(n) является объект, из которого вызывается метод, и при этом полю code данного метода присваивается значение n. Из этого же объекта вызывается метод set() с аргументом s. Как следствие поле name объекта получает значение s, а результатом возвращается ссылка на все тот же объект. Итог такой: при вызове из объекта метода set() с двумя аргументами полям объекта присваиваются значения, а результатом возвращается ссылка на объект.

Похожая ситуация имеет место в конструкторе. В теле конструктора класса MyClass командой set(n,s).show() полям объекта присваиваются значения, и из этого объекта вызывается метод show(). Поэтому при создании объекта класса MyClass автоматически отображаются значения полей объекта.

В классе ObjectAsResultDemo описан статический метод createObject(), возвращающий результатом объект класса MyClass. У метода два аргумента (целочисленный n и текстовый s), определяющие значения полей объекта, возвращаемого результатом метода. В теле метода выполняется всего одна команда return new MyClass(n,s), которой создается новый объект, и ссылка на него возвращается результатом.

В главном методе программы сначала с помощью команды MyClass obj=cr eateObject(100,"alpha") создаем объект путем вызова статического метода createObject(), а результат метода записывается в объектную переменную obj класса MyClass. Далее инструкцией obj.set(200).show() изменяется значения целочисленного поля, и отображаются значения полей объекта obj. Затем командой obj.set("bravo").show() изменяется значение текстового поля, и снова отображаются значения полей объекта obj. Наконец, для проверки выполняется команда obj.show().

Еще одна команда createObject(300,"charlie").set(400,"delta").show() дает пример вызова метода show() из объекта, возвращаемого при вызове метода set() из объекта, создаваемого при вызове метода createObject() (понятно, что речь идет об одном и том же объекте).

НА ЗАМЕТКУ

В языке Java используется специальная система «сборки мусора», которой, кроме прочего, из памяти удаляются те объекты, созданные

в процессе выполнения программы и на которые в программе нет ссылок. Например, допустим, что в теле метода создается объект, ссылка на объект записывается в локальную объектную переменную, и это единственная переменная, содержащая ссылку на объект. После завершения выполнения метода локальная переменная удаляется из памяти. В результате на созданный в теле метода объект больше не будет ссылок в программе, поэтому объект из памяти удаляется. С другой стороны, если в теле метода создается объект и ссылка на него возвращается результатом метода, то даже после завершения выполнения метода в программе на объект имеется ссылка, поэтому такой объект из памяти не удаляется.

Объекты и наследование

— А рекомендацию нашего венценосного брата короля Эдуарда этот Мальгрим имеет?
— Имеет, Ваше Величество!
— Хорошая рекомендация?
— Плохая, Ваше Величество!

Из к/ф «31 июня»

Очень многие нетривиальные свойства объектов проявляются при наследовании. Далее мы рассмотрим несколько простых (но показательных примеров).

Фабрика объектов

Напомним, что объектная переменная суперкласса может ссылаться на объект подкласса. Мы воспользуемся этим замечательным обстоятельством для того, чтобы создать метод, возвращающий, в зависимости от фактического значения своего аргумента, объекты разных классов. Рассмотрим программный код, представленный в листинге 8.5.

Листинг 8.5. Программный код проекта ObjectFactoryApplication

```
// Абстрактный суперкласс:
abstract class Base{
  // Объявление абстрактного метода:
  abstract void show();
}
// Производные классы:
```

```
class Alpha extends Base{
  void show(){
    System.out.println("Объект класса Alpha");
  }
}
class Bravo extends Base{
  void show(){
    System.out.println("Объект класса Bravo");
  }
}
class Charlie extends Base{
  void show(){
    System.out.println("Объект класса Charlie");
  }
}
// Главный класс:
class ObjectFactoryDemo{
  // Статический метод для создания объектов
  // разных классов:
  static Base createObject(int n){
    if(n==1) return new Alpha();
    if(n==2) return new Bravo();
    return new Charlie();
  }
  // Главный метод программы:
  public static void main(String[] args){
    // Объектная переменная абстрактного суперкласса:
    Base obj;
    for(int k=1;k<=3;k++){
      // Создание объекта:
      obj=createObject(k);
      // Вызов из объекта метода:
```

```
      obj.show();
    }
  }
}
```

В результате выполнения программы получим следующее:

Результат выполнения программы (из листинга 8.5)

```
Объект класса Alpha
Объект класса Bravo
Объект класса Charlie
```

В программе мы описываем абстрактный суперкласс Base, в котором объявлен абстрактный метод show(). На основе суперкласса Base путем наследования создаются подклассы Alpha, Bravo и Charlie, в каждом из которых метод show() определяется таким образом, что в консольном окне отображается название соответствующего класса.

В классе ObjectFactoryDemo, кроме главного метода, описывается статический метод createObject(). Идентификатором типа результата указано название класса Base. Это означает, что результатом метод возвращает ссылку на объект. Понятно, что это не может быть объект класса Base, поскольку класс Base абстрактный. Но это может быть объект одного из производных классов объекта Base — то есть объект класса Alpha, Bravo или Charlie. Причина такого «демократизма» как раз в том, что объектная переменная суперкласса может ссылаться на объект подкласса.

У метода createObject() один целочисленный аргумент. Если значение аргумента метода равно 1, результатом возвращается ссылка на объект класса Alpha. Если значение аргумента равно 2, то результатом возвращается ссылка на объект класса Bravo. При всех прочих значениях аргумента результатом метод возвращает ссылку на объект класса Charlie. Понятно, что в каждом из перечисленный случаев ссылка-результат выполняется на объект, созданный в теле метода.

ДЕТАЛИ

Нелишним будет выделить основные моменты, связанные с возвращением результата методом. Итак, если метод возвращает

результат, то при вызове метода в памяти автоматически выделяется место для записи результата метода. Место выделяется для переменной с типом, совпадающим с типом результат метода. В нашем случае, поскольку идентификатором типа для метода createObject() указано название класса Base, то под результат выделяется место, как для объектной переменной класса Base. Далее, когда в теле метода выполняется return-инструкция с некоторым значением, то данное значение записывается в область памяти, выделенную под результат метода. Так, если метод createObject() возвращает результатом ссылку на объект класса Alpha, то данная ссылка (фактически адрес объекта) записывается в область памяти, выделенную под результат метода. В принципе результатом должна быть объектная переменная класса Base, но поскольку класс Alpha является подклассом суперкласса Base, то объектная переменная класса Base может содержать значением ссылку на объект класса Alpha. То же происходит при возвращении методом createObject() ссылок на объекты классов Bravo и Charlie.

В главном методе программы объявляется объектная переменная obj абстрактного суперкласса Base. Затем запускается оператор цикла, в котором индексная переменная k последовательно принимает значения 1, 2 и 3. Каждый раз переменная передается аргументом методу createObject(), а результат метода записывается в объектную переменную obj. Затем через эту переменную вызывается метод show(). И хотя объектная переменная одна и та же, но она последовательно ссылается на объекты разных классов. Версия метода show(), вызываемого через переменную obj, определяется объектом, на который на момент вызова метода ссылается переменная. Поэтому в окне вывода появляется три разных сообщения.

Конструктор создания копии

Еще один, важный с практической точки зрения, пример иллюстрирует использования в классе *конструктора создания копии*. Речь идет об описанном в классе конструкторе, который позволяет создавать на основе уже существующего объекта другой объект (обычно с такими же значениями полей, но это не обязательно). В принципе, ситуация достаточно тривиальная, но если при этом используется наследование, то могут возникнуть некоторые не очень очевидные «моменты». Чтобы не быть голословными, сразу перейдем к рассмотрению примера. Интересующий нас программный код представлен в листинге 8.6.

Листинг 8.6. Программный код проекта CopyConstructorApplication

```
// Суперкласс:
class Base{
  // Текстовое поле:
  String name;
  // Конструктор с текстовым аргументом:
  Base(String txt){
    name=txt;
  }
  // Конструктор создания копии:
  Base(Base obj){
    name=obj.name;
  }
}
// Подкласс:
class MyClass extends Base{
  // Целочисленное поле:
  int code;
  // Конструктор с текстовым и целочисленным полем:
  MyClass(String txt,int n){
    // Вызов конструктора суперкласса:
    super(txt);
    // Присваивание значения целочисленному полю:
    code=n;
  }
  // Конструктор создания копии:
  MyClass(MyClass obj){
    // Вызов конструктора суперкласса:
    super(obj);
    // Присваивание значения целочисленному полю:
    code=obj.code;
  }
  // Метод для отображения значений полей:
```

```
  void show(){
    // Значение текстового поля:
    System.out.println("Текстовое поле: "+name);
    // Значение целочисленного поля:
    System.out.println("Целочисленное поле: "+code);
  }
}
// Главный класс:
class CopyConstructorDemo{
  public static void main(String[] args){
    // Создание объекта с вызовом конструктора
    // с двумя аргументами:
    MyClass A=new MyClass("Желтый",200);
    // Создание объекта с вызовом конструктора
    // создания копии:
    MyClass B=new MyClass(A);
    // Изменение значений полей первого объекта:
    A.name="Красный";
    A.code=100;
    // Создание объекта с вызовом конструктора
    // создания копии (аргумент — анонимный объект):
    MyClass C=new MyClass(new MyClass("Зеленый",300));
    // Отображение значений полей:
    System.out.println("Объект A");
    A.show();
    System.out.println("Объект B");
    B.show();
    System.out.println("Объект C");
    C.show();
  }
}
```

Ниже показано, каким будет результат выполнения программы.

Результат выполнения программы (из листинга 8.6)

```
Объект A
Текстовое поле: Красный
Целочисленное поле: 100
Объект B
Текстовое поле: Желтый
Целочисленное поле: 200
Объект C
Текстовое поле: Зеленый
Целочисленное поле: 300
```

В данной программе описывается суперкласс Base, в котором есть текстовое поле name и две версии конструктора: с текстовым аргументом и конструктор создания копии, аргументом которому передается объект класса Base. Конструкторы простые. В каждом из них присваивается значение полю name. Но только в конструкторе с текстовым аргументом значение поля name создаваемого объекта определяется аргументом конструктора, а в конструкторе создания копии значение поля name такое же, как значение поля name объекта, переданного аргументом конструктору.

На основе суперкласса Base наследованием создается подкласс MyClass. В этом подклассе дополнительно появляется целочисленное поле code, метод show(), предназначенный для отображения значений полей name и code, а также описаны две версии конструктора: имеется версия с двумя аргументами (текстовым и целочисленным), а также конструктор создания копии, аргументом которому передается объект класса MyClass. Именно конструктор создания копии представляет для нас наибольший интерес. Дело в том, что командой super(obj) в теле конструктора вызывается конструктор суперкласса. Вместе с тем объект obj (аргумент конструктора создания копии) объявлен как относящийся к классу MyClass. Но в классе Base конструктор создания копии описывается с аргументом, являющимся объектом класса Base. Фактически, мы вместо объекта класса Base передаем объект класса MyClass. Однако благодаря тому, что класс MyClass является подклассом суперкласса Base, ошибки здесь нет.

ДЕТАЛИ

Когда в теле конструктора создания копии, описанном в подклассе MyClass, происходит вызов конструктора суперкласса Base и этому

конструктору аргументом передается объект подкласса MyClass, то для аргумента конструктора суперкласса выделяется место в памяти. Место выделяется как для объектной переменной класса Base. Но записывается туда ссылка на объект класса MyClass. Поскольку класс MyClass является подклассом суперкласса Base, то проблем с этим не возникает. В результате конструктор создания копии суперкласса Base «оперирует» с объектом подкласса MyClass. Однако здесь уместно напомнить, что при получении доступа к объекту подкласса через переменную суперкласса доступны только те члены, которые объявлены в суперклассе.

В главном методе программы есть несколько команд создания объектов класса MyClass, в том числе и с использованием конструктора создания копии. В частности, сначала создается объект A. При создании объекта вызывается конструктор класса MyClass с двумя аргументами. Затем на основе объекта A создается объект B. Это копия объекта: значения полей у объекта B такие же, как у объекта A, но все равно это разные объекты. Поэтому после внесения изменений в значения полей объекта A значения полей объекта B остаются неизменными.

Объект C создается с использованием конструктора создания копии. Аргументом конструктору передается анонимный объект, который, в свою очередь, создается командой new MyClass("Зеленый",300).

ДЕТАЛИ

Происходит следующее: создается объект со значениями полей "Зеленый" и 300, а на его основе — копия объекта (объект C). Исходный анонимный объект из памяти автоматически удаляется системой «сборки мусора» (поскольку объект анонимный и на него в программе нет ссылки), а объект C продолжает свое существование.

Массивы и объекты

— Какая гадость.

— Это не гадость. Это последние достижения современной науки.

Из к/ф «31 июня»

Далее мы рассмотрим ряд вопросов, связанных с совместным использованием массивов и объектов. В первую очередь обсудим некоторые

особенности, связанные с использованием классов (и объектов, созданных на их основе), в которых полем является массив.

Массив как поле

Итак, допустим, что необходимо описать класс, у которого полем является массив. Как это можно сделать? В принципе, достаточно просто: в классе полем объявляется переменная массива, а создание и заполнение собственно массива обычно переносится в конструктор. Небольшой пример, поясняющий, как такое происходит, представлен в листинге 8.7. Там описывается класс Binomial, у которого есть закрытое поле, представляющее собой целочисленный массив. Массив при создании объект класса заполняется биномиальными коэффициентами.

НА ЗАМЕТКУ

Напомним, что по определению биномиальные коэффициенты $C_n^k = n!/(k!(n-k)!)$. При вычислении коэффициентов мы исходим из того, что $C_n^0 = 1$ и для всех $k = 1, 2, 3, \ldots, n$ имеет место соотношение $C_n^k = C_n^{k-1} \cdot (n-k+1)/k$.

Также для удобства в классе Binomial переопределен метод toString(), благодаря чему при попытке «напечатать» объект класса Binomial в окне вывода отображается содержимое массива, являющегося полем соответствующего объекта. Теперь рассмотрим программный код примера.

Листинг 8.7. Программный код проекта ArrayAsFieldApplication

```
// Класс с полем-массивом:
class Binomial{
  // Закрытое поле-массив:
  private int[] binoms;
  // Конструктор:
  Binomial(int n){
    // Создание массива:
    binoms=new int[n+1];
    // Значение начального элемента массива:
    binoms[0]=1;
```

```
    // Заполнение массива:
    for(int k=1;k<=n;k++){
      binoms[k]=binoms[k-1]*(n-k+1)/k;
    }
  }
  // Переопределение метода toString():
  public String toString(){
    // Текстовая переменная для формирования результата:
    String txt="| ";
    // Добавление к тексту значений элементов массива:
    for(int k=0;k<binoms.length;k++){
      txt+=binoms[k]+" | ";
    }
    // Результат метода:
    return txt;
  }
}
// Главный класс:
class ArrayAsFieldDemo{
  public static void main(String[] args){
    // Создание объектов:
    Binomial A=new Binomial(5);
    Binomial B=new Binomial(10);
    // Отображение биномиальных коэффициентов:
    System.out.println(A);
    System.out.println(B);
  }
}
```

В классе Binomial поле-массив объявляется инструкцией private int[] binoms. Но на самом деле массив здесь не создается. Поле binoms может содержать ссылку на массив, который предстоит создать. Создается массив в конструкторе.

НА ЗАМЕТКУ

Поле binoms объявлено с ключевым словом private и является закрытым. Причина в том, что элементы массива должны содержать значения биномиальных коэффициентов — то есть это определенная последовательность значений, которая вычисляется по установленным правилам. Объявление поля закрытым ограничивает доступ к элементам массива и уменьшает вероятность «случайного» изменения значения того или иного элемента массива. Хотя в принципе наличие ключевого слова private в данном примере не является принципиальным.

Конструктору передается один целочисленный аргумент (обозначен как n) — значение нижнего индекса для биномиальных коэффициентов C_n^k. Поскольку верхний индекс k пробегает значения от 0 до n включительно, то количество биномиальных коэффициентов при заданном значении n равно $n + 1$. Поэтому командой binoms=new int[n+1] в теле конструктора создается массив из n+1 элементов, а ссылка на него записывается в поле binoms.

Далее, командой binoms[0]=1 присваивается значение начальному (с нулевым индексом) элементу массива, после чего запускается оператор цикла, в котором командой binoms[k]=binoms[k-1]*(n-k+1)/k значение очередного биномиального коэффициента binoms[k] вычисляется на основе значения предыдущего коэффициента binoms[k-1]. При этом индексная переменная k пробегает значения от 1 до n включительно.

В методе toString() формируется текстовая строка, содержащая значения всех элементов массива binoms. Данная строка возвращается результатом метода.

НА ЗАМЕТКУ

Метод toString() автоматически вызывается каждый раз, когда объект класса Binomial должен быть приведен к текстовому формату.

В главном методе программы командами Binomial A=new Binomial(5) и Binomial B=new Binomial(10) создаются объекты A и B класса Binomial, после чего командами System.out.println(A) и System.out.println(B) содержимое полей-массивов этих объектов отображается в окне вывода.

Результат выполнения программы такой, как показано ниже.

Результат выполнения программы (из листинга 8.7)

```
| 1 | 5 | 10 | 10 | 5 | 1 |
| 1 | 10 | 45 | 120 | 210 | 252 | 210 | 120 | 45 | 10 | 1 |
```

Первая строка представляет собой набор биномиальных коэффициентов C_n^k при $n = 5$, а вторая строка содержит значения биномиальных коэффициентов при $n = 10$. В обоих случаях верхний индекс k принимает значения от 0 до n.

Массив объектов

При создании массива объектов массив создается из объектных переменных, а уже затем в каждую из переменных записывается ссылка на объект. Далее мы рассмотрим небольшой пример, связанный, как и в предыдущем случае, с вычислением биномиальных коэффициентов. Только если в примере из листинга 8.7 описывался класс с полем-массивом, то теперь будет создаваться массив из объектов, у каждого из которых имеется числовое поле, значение которого вычисляется как значение биномиального коэффициента. Рассмотрим программный код в листинге 8.8.

Листинг 8.8. Программный код проекта ArrayOfObjectsApplication

```
// Класс с целочисленным полем:
class MyClass{
  // Закрытое целочисленное поле:
  private int number;
  // Конструктор:
  MyClass(int n){
    number=n;
  }
  // Метод для считывания значения поля:
  int get(){
    return number;
  }
}
```

```
// Главный класс:
class ArrayOfObjectsDemo{
  // Статический метод для создания массива объектов:
  static MyClass[] createBinoms(int n){
    // Создается массив из объектных переменных:
    MyClass[] bins=new MyClass[n+1];
    // Создание объекта, ссылка на который записывается
    // в начальный элемент массива:
    bins[0]=new MyClass(1);
    // Создание объектов и заполнение массива:
    for(int k=1;k<=n;k++){
      // Создается новый объект и ссылка на него
      // присваивается значением элементу массива:
      bins[k]=new MyClass(bins[k-1].get()*(n-k+1)/k);
    }
    // Результат метода:
    return bins;
  }
  // Статический метод для отображения значений полей
  // объектов, формирующих массив:
  static void show(MyClass[] objs){
    // Начальное значение текстовой переменной:
    String txt="| ";
    // В текст дописываются значения полей объектов,
    // которые формируют массив, переданный
    // аргументом методу:
    for(int k=0;k<objs.length;k++){
      txt+=objs[k].get()+" | ";
    }
    // Отображение сообщения в окне вывода:
    System.out.println(txt);
  }
```

```
  // Главный метод программы:
  public static void main(String[] args){
    // Создание массивов из объектов:
    MyClass[] A=createBinoms(5);
    MyClass[] B=createBinoms(10);
    // Отображение значений полей объектов из массивов:
    show(A);
    show(B);
  }
}
```

Результат выполнения программы фактически такой же, как и в предыдущем случае:

Результат выполнения программы (из листинга 8.8)

```
| 1 | 5 | 10 | 10 | 5 | 1 |
| 1 | 10 | 45 | 120 | 210 | 252 | 210 | 120 | 45 | 10 | 1 |
```

Проанализируем программный код примера. Во-первых, мы используем класс MyClass, в котором объявлено закрытое целочисленное поле number, есть конструктор с одним аргументом (значение, присваиваемое полю) и еще предусмотрен метод get(), возвращающий результатом значение целочисленного поля.

НА ЗАМЕТКУ

Таким образом, значение поля number определяется при создании объекта, после чего значение поля неизменно. Для считывания значения поля используется метод get().

В классе ArrayOfObjectsDemo описывается главный метод main() и еще два статических метода. Метод createBinoms() предназначен для создания массива объектов. Аргументом методу передается целочисленный параметр n, который соответствует нижнему индексу биномиальных коэффициентов, которыми заполняются поля объектов, формирующих массив. Результатом метод возвращает ссылку на массив, состоящий из объектных переменных класса MyClass (поэтому тип результата метода определяется

инструкцией MyClass[], в которой пустые квадратные скобки свидетельствуют о том, что речь идет не об объектной переменной класса MyClass, а о переменной массива, элементами которого являются объектные переменные класса MyClass). В теле метода командой MyClass[] bins=new MyClass[n+1] создается массив размера n+1, а элементами массива являются объектные переменные класса MyClass. Но пока что элементам массива не присвоены значения. Командой bins[0]=new MyClass(1) создается объект класса MyClass, полю number которого присваивается значение 1 (аргумент конструктора), а ссылка на созданный объект присваивается значением начальному элементу (с нулевым индексом) массива bins. Прочие элементы массива заполняются при выполнении оператора цикла. Там индексная переменная k пробегает значения от 1 до n включительно, и за каждый цикл выполняется команда bins[k]=new MyClass(bins[k-1].get()*(n-k+1)/k). С ее помощью создается новый объект класса MyClass, причем аргументом конструктору передается выражение bins[k-1].get()*(n-k+1)/k. В этом выражении инструкцией bins[k-1].get() считывается значение поля number предыдущего объекта в массиве, после чего оно умножается на (n-k+1)/k и в результате получается значение для очередного биномиального коэффициента. После заполнения массива bins он возвращается результатом метода createBinoms().

Статический метод show() не возвращает результат, а аргументом ему передается массив из объектных переменных класса MyClass (аргумент описан инструкцией MyClass[] objs — то есть на самом деле это переменная массива, состоящего из объектных переменных класса MyClass). При вызове метода отображаются значения полей объектов из массива, который передан аргументом методу. При этом для считывания значения поля number из объекта, ссылка на который записана в элемент objs[k] массива мы используем инструкцию objs[k].get(), то есть вызываем из соответствующего объекта метод get().

В главном методе программы командами MyClass[] A=createBinoms(5) и MyClass[] B=createBinoms(10) создается два массива, а командами show(A) и show(B) в окне вывода отображаются биномиальные коэффициенты.

Цепочка объектов

Еще один способ организации объектов, не подразумевающий применения массива, состоит в том, что создается «цепочка» объектов: это группа объектов, в которой каждый объект (за исключением последнего) содержит ссылку на другой объект. Применение такого подхода проиллюстрируем на примере. Рассмотрим программный код из листинга 8.9.

Листинг 8.9. Программный код проекта ListOfObjectsApplication

```
// Класс для создания объектов в цепочке:
class MyClass{
  // Целочисленное поле:
  int number;
  // Ссылка на следующий объект в цепочке:
  MyClass next;
}
// Главный класс:
class ListOfObjectsDemo{
  // Статический метод для создания цепочки объектов:
  static MyClass createList(int n){
    // Создание первого объекта:
    MyClass obj=new MyClass();
    // Целочисленное поле первого объекта:
    obj.number=1;
    // Ссылка на последний (и пока что единственный)
    // объект в цепочке:
    MyClass t=obj;
    // Создание цепочки объектов:
    for(int k=1;k<=n;k++){
      // Создается новый объект и ссылка на него
      // записывается в поле next последнего (на
      // данный момент) объекта в цепочке:
      t.next=new MyClass();
      // Вычисление значения числового поля вновь
      // созданного объекта:
      t.next.number=t.number*(n-k+1)/k;
      // Созданный объект становится последним объектом
      // в цепочке объектов:
      t=t.next;
    }
    // Пустая ссылка для поля next последнего
```

```
    // объекта в цепочке:
    t.next=null;
    // Результат метода — ссылка на первый объект
    // в цепочке:
    return obj;
  }
  // Статический метод для отображения значений числовых
  // полей объектов из цепочки:
  static void showList(MyClass obj){
    // Начальное значение текстовой переменной:
    String txt="| ";
    // Ссылка на первый объект в цепочке:
    MyClass t=obj;
    // Добавление к тексту значений числовых полей:
    do{
      // К тексту дописывается значение числового поля
      // объекта, на который ссылается переменная t:
      txt+=t.number+" | ";
      // Переменная t указывает на следующий объект:
      t=t.next;
    }while(t!=null);
    // Отображение значений биномиальных коэффициентов:
    System.out.println(txt);
  }
  // Главный метод программы:
  public static void main(String[] args){
    // Создание цепочек объектов:
    MyClass A=createList(5);
    MyClass B=createList(10);
    // Отображение биномиальных коэффициентов:
    showList(A);
    showList(B);
  }
}
```

Результат выполнения программы имеет вполне знакомый вид:

Результат выполнения программы (из листинга 8.9)

```
| 1 | 5 | 10 | 10 | 5 | 1 |
| 1 | 10 | 45 | 120 | 210 | 252 | 210 | 120 | 45 | 10 | 1 |
```

Чтобы упростить программный код и не отвлекаться на второстепенные моменты, мы в некоторых местах существенно упростили программный код. В частности, мы используем очень простой код для класса MyClass, на основе которого создаются объекты для цепочки объектов. В классе объявлены всего два открытых поля: целочисленное поле number для записи значения биномиального коэффициента и поле next, являющееся ссылкой на объект класса MyClass. Таким образом, каждый объект класса MyClass может содержать числовое значение и ссылку на другой объект того же класса.

В классе ListOfObjectsDemo описывается три статических метода: один из них главный метод программы main(), а кроме него еще методы createList() (метод для создания цепочки объектов) и showList() (метод для отображения значений числовых полей объектов, формирующих цепочку объектов).

Статический метод createList() позволяет создать цепочку объектов класса MyClass. Результатом метод возвращает ссылку на первый объект в цепочке. Цепочка формируется по следующему принципу: первый объект содержит ссылку (в поле next) на второй объект, второй объект содержит ссылку (в своем поле next) на третий объект, третий объект ссылается на четвертый объект, и так далее. Последний объект в цепочке не ссылается ни на какой иной объект, поэтому его полю next значением присваивается пустая ссылка null. Числовые поля объектов в цепочке содержат значения биномиальных коэффициентов. Нижний индекс для биномиальных коэффициентов определяется целочисленным аргументом n метода createList() (соответственно, количество объектов в цепочке равно n+1).

В теле метода createList() командой MyClass obj=new MyClass() создается первый объект в цепочке. Командой obj.number=1 присваивается значение полю number данного объекта (биномиальный коэффициент $C_n^0 = 1$). Кроме этого, для выполнения итерационной процедуры, нам понадобится объектная переменная класса MyClass, в которую мы будем записывать ссылку на последний элемент в цепочке. Поэтому такая переменная объявляется и инициализируется командой MyClass t=obj. Начальным значением переменной t является ссылка на объект obj, поскольку пока что он является

не только первым, но одновременно и последним элементом в цепочке объектов. Затем запускается оператор цикла, в котором индексная переменная k пробегает значения от 1 до n включительно. За каждый цикл командой t.next=new MyClass() создается новый объект в цепочке и ссылка на него записывается в поле next объекта, на который в данный момент ссылается переменная t. После создания нового объекта с помощью команды t.next.number=t.number*(n-k+1)/k полю number этого объекта присваивается значение очередного биномиального коэффициента. В данном случае мы исходим из соотношения $C_n^k = C_n^{k-1} \cdot (n - k + 1)/k$ для биномиальных коэффициентов. В поле number объекта t записано значение коэффициента C_n^{k-1}, а значение коэффициента C_n^k записывается в поле number того объекта, ссылка на который содержится в поле next объекта t (ссылка на этот объект дается выражением t.next). Наконец, в результате выполнения команды t=t.next переменная t «перебрасывается» на тот объект, ссылка на который содержится в поле next — то есть мы создали новый объект, и теперь он последний объект в цепочке.

После завершения оператора цикла все объекты в цепочке созданы. У этих объектов заполнены числовые поля. Не присвоено только поле next самого последнего объекта в цепочке. Ссылка на последний объект в цепочке содержится в переменной t. Поэтому при выполнении команды t.next=null после оператора цикла поле next последнего объекта в цепочке получает значением пустую ссылку null. Последней командой return obj в теле метода createList() результатом возвращается ссылка на первый объект в цепочке.

Статический метод showList() не возвращает результат, а аргументом методу передается первый объект в цепочке объектов (обозначен как obj). Формально это просто объект класса MyClass, но поскольку каждый объект данного класса имеет поле next, то здесь проблем не возникает.

В теле метода формируется текстовое значение (объектная переменная txt класса String), содержащее значения числовых полей объектов из цепочки, а затем это текстовое значение отображается в окне вывода. Для перебора объектов в цепочке использован оператор цикла do-while и объектная переменная t класса MyClass, начальное значение которой является ссылкой на первый объект в цепочке. В теле оператора цикла командой txt+=t.number+" | " к тексту дописывается значение поля number объекта, на который (на данный момент) ссылается переменная t, после чего с помощь команды t=t.next данная переменная «перебрасывается» на следующий объект в цепочке. Оператор цикла выполняется до тех пор, пока не будет достигнут последний объект в цепочке — то есть пока истинно условие t!=null. Здесь мы учли, что после добавления в текст числового значения из последнего

объекта в цепочке переменная t получит новое значение из поля next объекта, а для последнего объекта в поле next записано значение null.

После того как текстовое значение сформировано, командой System.out.println(txt) оно отображается в окне вывода.

В главном методе программы мы создаем две цепочки объектов, для чего использованы команды MyClass A=createList(5) и MyClass B=createList(10). Отображаются значения числовых полей объектов из цепочек посредством команд showList(A) и showList(B).

Внутренние классы

У меня есть предложение. Пока мы окончательно не свихнулись — пойдем, посидим в «Вороном коне».

Из к/ф «31 июня»

Класс может быть описан внутри другого класса. Такой класс называется *внутренним*. Класс, в котором описан внутренний класс, будем называть *внешним* или *классом-контейнером*. Главная и далеко не очевидная особенность внутреннего состоит в том, что он имеет доступ к полям и методам внешнего класса. При этом из внешнего класса нет прямого доступа к полям и методам внутреннего класса (однако доступ к таким полям и методам можно получить только через объект внутреннего класса).

> **НА ЗАМЕТКУ**
>
> Далее мы рассмотрим достаточно простой подход относительно использования внутреннего нестатического класса. Кроме нестатических, существуют и статические внутренние (*вложенные*) классы (объявляются с ключевым словом static). Главная их особенность состоит в том, что доступ в таких классах есть только к статическим полям внешнего класса. Мы рассмотрим внутренние нестатические классы. Также следует отметить, что класс в Java может объявляться и в методах. Такие классы обычно называют *локальными*.

В качестве примера использования внутренних классов рассмотрим следующую задачу, в которой вычисляется итоговая сумма по банковскому вкладу. Итак, некоторый пользователь размещает определенную денежную сумму на депозит в банк. Необходимо вычислить сумму, которую он получит в итоге. Мы исходим из того, что если известна начальная сумма

вклада m, годовая процентная ставка r (в процентах) и время t (в годах), на которое размещается вклад, то итоговая сумма равна $m(1 + r/100)^t$. Задача простая, но решать ее мы будем нетривиальным образом. В частности, мы опишем внешний класс BankAccount, в котором опишем внутренний класс Person. У класса BankAccount, кроме прочего, будет числовое поле rate, определяющее годовую процентную ставку. Также в классе BankAccount будет поле fellow, являющееся объектом (объектной переменной) внутреннего класса Person. Информация об имени вкладчика, начальной сумме вклада и времени, на которое размещается вклад, определяется полями класса Person. Теперь рассмотрим программный код в листинге 8.10.

Листинг 8.10. Программный код проекта UsingInnerClassApplication

```
// Статический импорт:
import static javax.swing.JOptionPane.*;
// Внешний класс:
class BankAccount{
  // Числовое поле (процентная ставка):
  double rate;
  // Поле — объект внутреннего класса:
  Person fellow;
  // Конструктор:
  BankAccount(String name,double money,int time,double r){
    // Процентная ставка:
    rate=r;
    // Создание объекта внутреннего класса:
    fellow=new Person(name,money,time);
  }
  // Метод для отображения диалогового кона с сообщением:
  void show(){
    showMessageDialog(null,
      // Текст сообщения (автоматически вызывается
      // метод toString() для класса Person):
      fellow,
      // Название окна:
      "Депозит",
      // Тип окна:
```

```
      INFORMATION_MESSAGE
    );
  }
  // Внутренний класс:
  class Person{
    // Текстовое поле (имя вкладчика):
    String name;
    // Начальная сумма:
    double money;
    // Время, на которое размещается вклад:
    int time;
    // Метод для вычисления итоговой суммы:
    double getMoney(){
      // Переменная для записи результата:
      double s=money;
      // Вычисление результата:
      for(int k=1;k<=time;k++){
        s*=(1+rate/100);
      }
      // Уточнение результата (округление до
      // двух цифр после десятичной точки):
      s=Math.round(s*100)/100.0;
      // Результат метода:
      return s;
    }
    // Переопределение метода toString():
    public String toString(){
      String txt="Имя: "+name+"\n";
      txt+="Начальная сумма: "+money+"\n";
      txt+="Процентная ставка: "+rate+"\n";
      txt+="Время (лет): "+time+"\n";
      txt+="Итоговая сумма: "+getMoney();
      return txt;
    }
```

```
    // Конструктор внутреннего класса:
    Person(String txt,double m,int t){
      name=txt; // Имя вкладчика
      money=m;  // Начальная сумма вклада
      time=t;   // Время размещения вклада
    }
  }
}
// Главный класс:
class UsingInnerClassDemo{
  public static void main(String[] args){
    // Создание объекта внешнего класса:
    BankAccount ivanov=new BankAccount(
      "Иван Иванов", // Имя вкладчика
      1000.0,       // Начальная сумма
      5,           // Время размещения вклада
      8.0          // Процентная ставка
    );
    // Отображение информации по вкладу:
    ivanov.show();
  }
}
```

В результате выполнения программы появляется диалоговое окно, представленное на рис. 8.1.

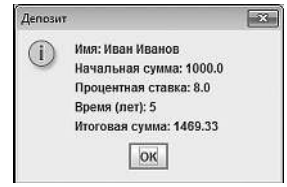


Рис. 8.1. Диалоговое окно с информацией по вкладу

Анализ программного кода начнем с внутреннего класса Person, описанного в классе BankAccount. Класс Person достаточно простой. В нем есть:

- текстовое поле name для запоминания имени вкладчика;
- поле money типа double для записи значения начальной суммы вклада;
- поле time типа int для записи значения интервала времени (в годах), на которое размещается вклад;
- конструктор с тремя аргументами;
- метод getMoney() для определения итоговой суммы по вкладу;
- в классе переопределяется метод toString().

Особенность кода класса в том, что в методах getMoney() и toString() используется ссылка rate (процентная ставка) на поле внешнего класса BankAccount.

Что касается класса BankAccount, то в нем кроме поля rate и внутреннего класса Person описан конструктор с четырьмя аргументами и метод show(), предназначенный для отображения информации по вкладу. Чтобы понять «взаимодействие» внутреннего и внешнего классов, имеет смысл проанализировать процесс создания объекта класса BankAccount (например, объекта ivanov в главном методе программы). Понятно, что вызывается конструктор класса. Исходя из кода конструктора, присваивается значение полю rate и еще создается объект класса Person, ссылка на который записывается в поле fellow. У объекта fellow имеются поля name, money, time, а также метод getMoney() (и переопределенный метод toString()). Когда из объекта класса BankAccount вызывается метод show(), то в качестве отображаемого в диалоговом окне текста указана ссылка на поле fellow. Для приведения объекта fellow к текстовому формату вызывается метод toString(), в котором, в свою очередь, вызывается метод getMoney() и выполняется прямое обращение к полю rate объекта класса BankAccount. Таким образом, объект fellow внутреннего класса Person имеет доступ к полю rate того объекта класса BankAccount, в котором объект fellow является полем.

Анонимные классы

— Это безрассудство! Тебя могли увидеть!
— Ничего страшного, сочтут за обыкновенное привидение.

Из к/ф «Тот самый Мюнхгаузен»

Достаточно часто возникает необходимость в создании класса, который используется в программе один-единственный раз. В таких случаях

нередко используют *анонимные классы* — классы, у которых нет «собственного имени». Мы рассмотрим два способа создания анонимных классов: путем наследования абстрактного класса и путем реализации интерфейса.

Создание анонимного класса путем наследования абстрактного суперкласса

Допустим, имеется некоторый абстрактный класс, который назовем условно Суперкласс. Анонимный класс будем создавать наследование Суперкласса. Но на самом деле создание анонимного класса подразумевает, что на основе этого анонимного класса сразу же создается объект. Другими словами, нет смысла говорить об описании анонимного класса, если речь не идет о создании объекта. В принципе, объект тоже может быть анонимным. Но может быть и так, что ссылка на объект, созданный на основе анонимного класса, записывается в объектную переменную. Ссылка на объект анонимного класса может быть записана в переменную Суперкласса, на основе которого «конструируется» анонимный класс. Используется следующий шаблон:

```
Суперкласс переменная=new Суперкласс(аргументы){
  // Описание методов
};
```

То есть сама конструкция создания объекта на основе анонимного класса начинается, как и положено, с инструкции new, после которой указан конструктор суперкласса, а в фигурных скобках — код, содержащий описание абстрактных методов. Если речь идет об анонимном объекте, созданном на основе анонимного класса, то, скорее всего, такой анонимный объект передается аргументом какому-то методу или из него самого вызывается метод. В любом случае используется конструкция вида:

```
new Суперкласс(аргументы){
  // Описание методов
}
```

Небольшой пример, в котором использованы объекты, созданные на основе анонимного класса, который, в свою очередь, создается на основе абстрактного суперкласса, приведен в листинге 8.11.

Листинг 8.11. Программный код проекта AnonymousClassApplication

```
// Абстрактный суперкласс:
abstract class Base{
  // Текстовое поле:
  private String name;
  // Конструктор:
  Base(String txt){
    name=txt;
  }
  // Метод для отображения значения текстового поля:
  void show(){
    System.out.println("Имя объекта: "+name);
  }
  // Объявление абстрактного метода:
  abstract void hello();
}
// Главный класс:
class AnonymousClassDemo{
  public static void main(String[] args){
    // Создание объекта анонимного класса:
    Base obj=new Base("Красный"){
      // Описание абстрактного метода из суперкласса:
      void hello(){
        System.out.println("Объект анонимного класса");
      }
    }; // Завершение инструкции создания объекта
      // анонимного класса
    // Вызов методов из объекта, созданного на основе
    // анонимного класса:
    obj.show();
    obj.hello();
    // Создание анонимного объекта анонимного класса
    // и вызов их этого объекта метода showAll(),
```

```
    // описанного в анонимном классе:
    new Base("Зеленый"){
      // Описание абстрактного метода из суперкласса:
      void hello(){
        System.out.println("Анонимный объект");
      }
      // Описание нового метода:
      void showAll(){
        hello();
        show();
      }
    }.showAll(); // Вызов метода
  }
}
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 8.11)

```
Имя объекта: Красный
Объект анонимного класса
Анонимный объект
Имя объекта: Зеленый
```

Мы описываем абстрактный суперкласс Base, у которого есть закрытое текстовое поле name, конструктор с одним аргументом, обычный (не абстрактный метод) show(), которым отображается значение поля name, а также объявлен абстрактный метод hello() (не возвращающий результат и не имеет аргументов).

В главном методе программы путем наследования класса Base дважды создается анонимный класс и, соответственно, два объекта: не анонимный и анонимный. Например, в программе есть такая команда (приводится без комментариев):

```
Base obj=new Base("Красный"){
  void hello(){
    System.out.println("Объект анонимного класса");
```

```
  }
};
```

В данном случае ссылка на созданный объект записывается в объектную переменную obj класса Base. При создании объекта вызывается конструктор суперкласса с аргументом "Красный", поэтому у объекта поле name будет иметь именно такое значение. Метод hello() определен так, что при его вызове выполняется команда System.out.println("Объект анонимного класса"). Подтверждением служит результат выполнения команд obj.show() и obj.hello().

Следующая инструкция является более замысловатой:

```
new Base("Зеленый"){
  void hello(){
    System.out.println("Анонимный объект");
  }
  void showAll(){
    hello();
    show();
  }
}.showAll();
```

Данной командой создается анонимный объект анонимного класса, и из этого объекта вызывается метод showAll(), описанный в анонимном классе (в суперклассе Base такой метод не объявлен). Метод showAll() описан так, что в его теле вызываются методы hello() и show() из суперкласса, причем метод hello() описан в анонимном классе (поскольку в суперклассе он только объявлен).

Создание анонимного класса через реализацию интерфейса

Еще один часто используемый способ создания объектов анонимного класса базируется на реализации в анонимном классе интерфейса. Подход здесь абсолютно такой же, как и в случае с наследованием класса — с поправкой на то, что имеем дело с интерфейсом. Ниже представлен шаблон создания объекта на основе анонимного класса, реализующего Интерфейс, при условии, что ссылка на созданный объект записывается в интерфейсную переменную:

```
Интерфейс переменная=new Интерфейс(){
  // Описание методов
};
```

Анонимный объект анонимного класса при реализации в последнем интерфейса может создаваться кодом следующего вида:

```
new Интерфейс(){
  // Описание методов
}
```

Пример, аналогичный рассмотренному выше, приведен в листинге 8.12. Но только на это раз вместо абстрактного класса использован интерфейс.

Листинг 8.12. Программный код проекта MoreAnonymousClassApplication

```
// Интерфейс:
interface Base{
  // Метод с кодом по умолчанию:
  default void show(){
    System.out.println("Интерфейс Base");
  }
  // Объявление метода:
  void hello();
}
// Главный класс:
class MoreAnonymousClassDemo{
  public static void main(String[] args){
    // Создание объекта анонимного класса:
    Base obj=new Base(){
      // Описание метода из интерфейса:
      public void hello(){
        System.out.println("Объект анонимного класса");
      }
```

```
    }; // Завершение инструкции создания объекта
      // анонимного класса
    // Вызов методов из объекта, созданного на основе
    // анонимного класса:
    obj.show();
    obj.hello();
    // Создание анонимного объекта анонимного класса
    // и вызов их этого объекта метода showAll(),
    // описанного в анонимном классе:
    new Base(){
      // Описание метода из интерфейса:
      public void hello(){
        System.out.println("Анонимный объект");
      }
      // Описание нового метода:
      void showAll(){
        hello();
        show();
      }
    }.showAll(); // Вызов метода
  }
}
```

Ниже показано, как выглядит результат выполнения программы.

Результат выполнения программы (из листинга 8.12)

```
Интерфейс Base
Объект анонимного класса
Анонимный объект
Интерфейс Base
```

Программные коды в листинге 8.11 и листинге 8.12 достаточно схожи, так что хочется верить, что особых комментариев приведенный здесь

пример не требует. Что касается анонимных классов, то с ними мы еще будем иметь дело при обсуждении вопросов, связанных с созданием приложений с графическим интерфейсом.

Резюме

Я предупреждал. У джентльменов нет оснований обижаться на меня.

Из к/ф «В поисках капитана Гранта»

- При передаче аргументов методам на самом деле передаются копии аргументов, которые создаются автоматически. Поэтому в теле метода нельзя изменить значение аргумента.
- При передаче аргументом методу объекта на самом деле передается ссылка на объект. Если нужно, чтобы метод возвращал объект результатом, то в теле метода создается соответствующий объект, а ссылка на него возвращается результатом метода.
- Если необходимо описать класс с полем-массивом, то полем является переменная массива, значением которой присваивается ссылка на массив. Массив обычно создается при вызове конструктора. Аналогичным образом поступают, если необходимо описать класс с полем-объектом: полем является объектная переменная, а значением ей присваивается ссылка на объект.
- Для создания массива объектов создается массив из объектных переменных, значениями которым присваиваются ссылки на объекты.
- Класс может быть описан в классе. Такой класс называется внутренним. Внутренний класс имеет доступ к полям внешнего класса.
- Объекты могут создаваться на основе анонимных классов. Такие анонимные классы создают, как правило, или наследованием абстрактного класса, или реализацией интерфейса.

Глава 9
ОБОБЩЕННЫЕ ТИПЫ ДАННЫХ

Может где-нибудь высоко в горах, но не в нашем районе, вы что-нибудь обнаружите для вашей науки.

Из к/ф «Кавказская пленница»

В *обобщенных классах* тип данных выступает параметром. Обычно обобщенные классы используют в тех случаях, когда с помощью класса реализуется некоторая «структура» или «конструкция», которая имеет универсальные характеристики и слабо зависит от конкретного типа данных, с которыми приходится оперировать. Это же замечание относится и к методам: нередко приходится решать задачи, в которых алгоритм обработки данных мало зависит от типа данных. В таких случаях удобно использовать *обобщенные методы*, в которых тип данных является параметром. Далее обсуждаются методы создания и использования обобщенных классов и методов, а также ряд смежных вопросов.

НА ЗАМЕТКУ

Для тех читателей, кто знаком с языком программирования C++ и, в частности, с концепцией шаблонов, следует заметить, что обобщенные классы в языке Java реализуются иначе, пор сравнению с шаблонами языка C++. Возможности обобщенных классов в языке Java намного «скромнее». Поэтому не следует автоматически переносить свойства и возможности шаблонов языка C++ на свойства и возможности обобщенных классов в Java.

Знакомство с обобщенными классами

И мы с пути кривого ни разу не свернем, а надо будет — снова пойдем кривым путем.

Из к/ф «Айболит-66»

Итак, мы начинаем знакомство с обобщенными классами. В первую очередь нам необходимо разрешить две фундаментальных задачи:

- выяснить, как описывается обобщенный класс;
- разобраться с тем, как на основе обобщенного класса создаются объекты.

Собственно этим далее мы и займемся.

Общие принципы использования обобщенных классов

Обобщенный класс в Java описывается так же просто, как и обычный класс. Но, разумеется, имеются некоторые особенности. Подход достаточно простой: при описании класса некий формальный параметр (или параметры) декларируются как такие, что обозначают тип данных, а при создании объекта класса идентификатор типа передается как параметр. Параметры, которые отождествляются в описании обобщенного класса с типом данных (будем их называть *обобщенными параметрами*), указываются в угловых скобках в описании класса после имени класса. Другими словами, используется следующий шаблон описания обобщенного класса:

```
class имя_класса<параметр(ы) типа>{
  // Описание обобщенного класса
}
```

Параметр (или параметры) типа просто перечисляются через запятую, и эти параметры могут использоваться в описании обобщенного класса (в качестве идентификаторов типа). В качестве параметра типа может использоваться в принципе любой идентификатор — главное, чтобы это не было одно из зарезервированных ключевых слов. Например, ниже приведен шаблонный код, которым объявляется обобщенный класс с названием MyClass, в котором использовано два обобщенных параметра X и Y:

```
class MyClass<X,Y>{
  // Описание обобщенного класса
}
```

При создании объекта на основе обобщенного класса в угловых скобках в команде объявления объектной переменной указывают идентификаторы типа, которые следует использовать вместо обобщенных параметров.

НА ЗАМЕТКУ

Значениями обобщенных параметров могут быть названия классов, но не могут быть идентификаторы для примитивных (базовых) типов. Другими словами, значениями обобщенных параметров такие идентификаторы, например, как int или char, быть не могут. Вместо них следует использовать классы-оболочки (в данном случае Integer и Character).

Также идентификаторы указываются в угловых скобках в инструкции создания объекта — между названием класса и круглыми скобками с аргументами конструктора. Так, команда создания объекта на основе обобщенного класса MyClass с двумя обобщенными параметрами могла бы иметь такой вид:

```
MyClass<Integer,Character> obj=new MyClass<Integer,Character>(аргументы);
```

Команда означает, что при создании объекта на основе обобщенного класса вместо параметра X следует использовать идентификатор Integer, а вместо параметра Y следует использовать идентификатор типа Character. Также приемлемо использовать упрощенный синтаксис: идентификаторы типов во вторых угловых скобках (в команде вызова конструктора) можно не указывать (но при этом угловые скобки должны присутствовать). То есть вместо приведенной выше команды создания объекта на основе обобщенного класса допустимо было бы использовать такую:

```
MyClass<Integer,Character> obj=new MyClass<>(аргументы);
```

Здесь подстановка для обобщенных параметров при создании объекта определяется на основе идентификаторов, указанных в первых угловых скобках (в инструкции объявления объектной переменной). Так дела обстоят в теории. А чтобы теория не представлялась слишком оторванной от жизни, далее рассмотрим простенький пример.

Пример создания обобщенного класса

Допустим, что перед нами стоит задача по описанию класса, у которого есть поле определенного типа, конструктор с одним аргументом и метод, которым отображается значение поля. Понятно, что каким бы ни был фактический тип поля, код класса достаточно универсальный. Разные версии класса для разных типов (простых) поля отличались бы

только идентификатором типа в инструкции объявления поля и описании конструктора. Это как раз тот случай, когда удобно использовать обобщенный класс. Именно так и поступим. Соответствующий программный код представлен в листинге 9.1.

Листинг 9.1. Программный код проекта UsingGenClassApplication

```
// Обобщенный класс с одним параметром типа:
class MyClass<X>{
  // Поле обобщенного типа:
  X data;
  // Конструктор с аргументом обобщенного типа:
  MyClass(X data){
    // Присваивание значения полю:
    this.data=data;
  }
  // Метод для отображения значения поля:
  void show(){
    System.out.println("Значение поля: "+data);
  }
}
// Главный класс:
class UsingGenClassDemo{
  public static void main(String[] args){
    // Создание объекта на основе обобщенного класса.
    // Вместо обобщенного параметра используется
    // идентификатор Integer:
    MyClass<Integer> A=new MyClass<Integer>(100);
    // Создание объекта на основе обобщенного класса.
    // Вместо обобщенного параметра используется
    // идентификатор Character:
    MyClass<Character> B=new MyClass<Character>('B');
    // Создание объекта на основе обобщенного класса.
    // Вместо обобщенного параметра используется
```

```
    // идентификатор String:
    MyClass<String> C=new MyClass<String>("Зеленый");
    // Вызов метода show() из объектов, созданных
    // на основе обобщенного класса:
    A.show();
    B.show();
    C.show();
  }
}
```

Результат выполнения программы будет таким:

Результат выполнения программы (из листинга 9.1)

```
Значение поля: 100
Значение поля: B
Значение поля: Зеленый
```

Итак, инструкция <X> в описании класса MyClass означает, что класс обобщенный и через X в нем обозначен некоторый тип данных (какой именно — станет понятно при создании объекта на основе данного класса). У класса есть поле data, тип которого задан через обобщенный параметр X.

Аргумент конструктора класса также относится к типу X. Отображение значения поля data выполняется с помощью метода show(), описанного в классе MyClass.

В главном методе программы создается три разных объекта. Все они создаются на основе обобщенного класса MyClass, но в качестве обобщенного параметра каждый раз используются разные значения. Так, команда MyClass<Integer> A=new MyClass<Integer>(100) означает, что при создании объекта вместо параметра X используется имя класса Integer, а аргументом конструктору передается значение 100. Соответственно, поле data объекта A получает значение 100, и оно же отображается при вызове из объекта A метода show() (команда A.show()). Аналогично создаются объекты B и C, но только вместо параметра X используются соответственно идентификаторы Character и String (и, разумеется, конструктору передаются разные значения).

НА ЗАМЕТКУ

Вместо команды `MyClass<Integer> A=new MyClass<Integer>(100)` можно использовать команду `MyClass<Integer> A=new MyClass<>(100)`. Это же замечание относится и к прочим командам, которыми в программе на основе обобщенного класса `MyClass` создаются объекты.

Обобщенный класс с несколькими параметрами

Следующий пример, который мы рассмотрим, содержит описание нескольких обобщенных классов и в некоторых из них используется более одного обобщенного параметра. Программный код примера представлен в листинге 9.2.

Листинг 9.2. Программный код проекта UsingGenericsApplication

```
// Обобщенный класс с одним параметром:
class Alpha<X>{
  // Закрытое поле обобщенного типа:
  private X first;
  // Конструктор с аргументом обобщенного типа:
  Alpha(X first){
    // Вызов метода с аргументом обобщенного типа:
    set(first);
  }
  // Метод с аргументом обобщенного типа
  // для присваивания значения полю:
  void set(X first){
    this.first=first;
  }
  // Метод возвращает результатом значение поля:
  X get(){
    return first;
  }
}
// Обобщенный класс с двумя параметрами:
```

```
class Bravo<X,Y>{
  // Поле — объект обобщенного класса:
  Alpha<X> obj;
  // Поле обобщенного типа:
  Y second;
  // Конструктор с двумя аргументами обобщенного типа:
  Bravo(X first,Y second){
    // Создание объекта на основе обобщенного класса:
    obj=new Alpha<X>(first);
    // Присваивание значения полю:
    this.second=second;
  }
  // Метод для отображения значений полей:
  void show(){
    System.out.println("Значения "+obj.get()+" и "+second);
  }
}
// Главный класс:
class UsingGenericsDemo{
  // Главный метод:
  public static void main(String[] args){
    // Создание объекта на основе обобщенного класса:
    Bravo<Integer,Character> A=new Bravo<Integer,Character>(100,'A');
    // Отображение значений полей объекта:
    A.show();
    // Создание объекта на основе обобщенного класса:
    Bravo<String,Double> B=new Bravo<>("BRAVO",12.345);
    // Отображение значений полей объекта:
    B.show();
  }
}
```

В результате выполнения программы получаем следующее:

Результат выполнения программы (из листинга 9.2)

```
Значения 100 и A
Значения BRAVO и 12.345
```

В программе описывается обобщенный класс Alpha с одним параметром (параметр обозначен как X). В классе объявлено закрытое поле first обобщенного типа X. У конструктора, описанного в классе, один аргумент обобщенного типа, который определяет значение поля first. Для присваивания значения полю first предназначен метод set(), который не возвращает результат и которому передается один аргумент обобщенного типа. Наконец, метод get() возвращает значение обобщенного типа — это значение поля first объекта, из которого вызывается метод.

Класс Bravo является обобщенным классом с двумя обобщенными параметрами (параметры обозначены как X и Y).

В классе Bravo объявлены два поля. Поле obj является переменной обобщенного класса Alpha с обобщенным параметром X. Поле объявлено инструкцией Alpha<X> obj. Также в классе Bravo объявляется поле second обобщенного типа Y.

У конструктора, описанного в классе, два аргумента обобщенного типа: один типа X, и другой типа Y. В теле конструктора командой obj=new Alpha<X>(first) создается объект обобщенного класса Alpha с параметром X, и ссылка на объект записывается в поле obj. После этого присваивается значение полю second.

Наконец, в классе Bravo описан метод show(), с помощью которого отображаются значения поле объекта, из которого вызывается метод.

В главном методе программы создаются объекты на основе обобщенного класса Bravo (с использованием разных значений для обобщенных параметров), и из созданных объектов вызывается метод show().

НА ЗАМЕТКУ

Вообще, следует отметить, что в Java реализации обобщенного класса при разных значениях обобщенных параметров не интерпретируются как разные классы. Поэтому в вопросе явного указания значений для обобщенных параметров при создании объектов на основе обобщенных классов прослеживается некоторый «демократизм».

Обобщенные методы

— Благородная Нинэт, я вам предлагаю маленький заговор.
— А большой нельзя?
— Маленький, но с большими последствиями.
— Что надо делать? Я готова на все.

Из к/ф «31 июня»

Помимо обобщенных классов, в Java можно создавать *обобщенные методы*. В описании обобщенного метода тип определяется через обобщенный параметр (или параметры), а при вызове метода значения обобщенных параметров определяются на основе типов аргументов, переданных методу. Обычно (но не обязательно) обобщенные методы являются статическими.

Создание статического обобщенного метода

При описании обобщенного метода обобщенный параметр типа в угловых скобках указывается перед идентификатором типа результата метода. Шаблон объявления статического обобщенного метода выглядит следующим образом:

```
static <параметр> тип_результата имя_метода(аргументы){
  // Код метода
}
```

Параметр (или параметры) используются в описании аргументов метода и/или типа результата метода. При вызове метода это дает возможность идентифицировать значения обобщенных параметров, которые следует использовать при выполнении кода метода. Небольшой пример, в котором объявляются и используются статические обобщенные методы, представлен в листинге 9.3.

Листинг 9.3. Программный код проекта UsingStaticGenMethodApplication

```
// Класс со статическими обобщенными методами:
class UsingStaticGenMethodDemo{
  // Метод с аргументом обобщенного типа:
  static <X> void show(X arg){
    System.out.println(arg);
  }
```

```
// Аргумент метода — обобщенный массив:
static <X> void show(X[] array){
  System.out.print("| ");
  // Отображение значений элементов массива:
  for(int k=0;k<array.length;k++){
    System.out.print(array[k]+" | ");
  }
  System.out.println("");
}
// Методу аргументом передается обобщенный массив
// и целочисленный индекс, а результатом возвращается
// значение элемента с соответствующим индексом:
static <X> X getElement(X[] array,int index){
 return array[index];
}
// Главный метод:
public static void main(String[] args){
  // Целочисленный массив:
  Integer[] nums={1,3,7,2};
  // Символьный массив:
  Character[] symbs={'A','W','L','O','B'};
  System.out.println("Вызов метода show()");
  System.out.print("С текстовым аргументом: ");
  show("обобщенный метод");
  System.out.print("С int-аргументом: ");
  show(123);
  System.out.print("С double-аргументом: ");
  show(123.45);
  System.out.print("С char-аргументом: ");
  show('A');
  System.out.print("Целочисленный массив: ");
  show(nums);
  System.out.print("Символьный массив: ");
```

```
    show(symbs);
    // Поэлементное отображение массивов:
    System.out.println("Вызов метода getElement()");
    System.out.print("Целочисленный массив: *");
    for(int k=0;k<nums.length;k++){
      System.out.print(getElement(nums,k)+"*");
    }
    System.out.print("\nСимвольный массив: *");
    for(int k=0;k<symbs.length;k++){
      System.out.print(getElement(symbs,k)+"*");
    }
    System.out.println("");
  }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 9.3)

```
Вызов метода show()
С текстовым аргументом: обобщенный метод
С int-аргументом: 123
С double-аргументом: 123.45
С char-аргументом: A
Целочисленный массив: | 1 | 3 | 7 | 2 |
Символьный массив: | A | W | L | O | B |
Вызов метода getElement()
Целочисленный массив: *1*3*7*2*
Символьный массив: *A*W*L*O*B*
```

В программе, в классе UsingStaticGenMethodDemo, описываются две версии обобщенного метода show() и обобщенный метод getElement(). В этом же классе описывается главный метод программы.

В первой версии метода show() ему передается аргумент обобщенного типа X. При вызове метода в окне вывода отображается значение аргумента.

Второй версии метода show() передается массив, элементы которого относятся к обобщенному типу. При вызове метода отображаются значения элементов массива.

У метода getElement() два аргумента: массив с элементами обобщенного типа и целочисленное значение, определяющее индекс элемента массива. Результатом метод возвращает значение элемента массива (первый аргумент) с данным индексом (второй аргумент).

В главном методе программы есть примеры вызова методов show() и getElement() с разными аргументами. Хочется обратить внимание на несколько обстоятельств. Во-первых, массивы, которые передаются аргументами методам show() и getElement(), создаются на основе контейнерных классов (в частности, Integer и Character). Во-вторых, метод show() вызывается, кроме прочего, с аргументами (литералами), относящимися к базовым (простым) типам (int и char). Ошибки в этом случае не возникает, поскольку автоматически примитивные типы расширяются до типов классов-оболочек.

ДЕТАЛИ

Работа с обобщенными типами полна сюрпризов. Причина «неожиданностей» во многом связана с механизмом «стирания», который используется при компиляции программного кода с обобщенными методами и/или классами. Ситуация такая, что значения, используемые для обобщенных параметров, известны на мент компиляции, но после компиляции программы в байт-код эта информация «стирается». Проще говоря, на этапе выполнения программного кода в рамках обобщенного класса или метода нельзя явно идентифицировать используемое значение обобщенного параметра (но этом можно сделать на этапе компиляции). Поэтому, например, мы можем объявить в теле метода или в классе объектную переменную, относящуюся к обобщенному типу, но не можем создать объект на основе обобщенного типа. Причина в том, что переменная создается на этапе компиляции, а объект — в процессе выполнения программы. По той же причине мы можем использовать переменные массива с элементами обобщенного типа, но не можем создать массив с элементами обобщенного типа.

Создание нестатического обобщенного метода

Нестатический обобщенный метод создается практически точно так же, как и статический. Принципиальное отличие в том, что нестатический метод вызывается из объекта.

НА ЗАМЕТКУ

Мы можем описать обобщенный класс, в котором используется метод с обобщенными параметрами типа. Но можно поступить и иначе: класс описывать как обычный, но при этом один или более методов описать как обобщенные. Именно такая ситуация рассматривается далее.

Небольшой пример, в котором создается и используется нестатический метод, представлен в листинге 9.4.

Листинг 9.4. Программный код проекта UsingGenMethodsApplication

```
// Класс с обобщенным методом:
class MyClass{
  // Текстовое поле:
  String name;
  // Обобщенный метод:
  <X> void show(X arg){
    System.out.println(name+": "+arg);
  }
  // Конструктор:
  MyClass(String txt){
    name=txt;
  }
}
// Главный класс:
class UsingGenMethodsDemo{
  // Главный метод:
  public static void main(String[] args){
    // Создание объектов:
    MyClass A=new MyClass("Объект A");
    MyClass B=new MyClass("Объект B");
    // Вызов обобщенных методов из разных объектов:
    A.show(123);
    A.show("Alpha");
    A.show('A');
```

```
    B.show(123);
    B.show("Bravo");
    B.show('B');
  }
}
```

В результате выполнения программы получаем следующее:

Результат выполнения программы (из листинга 9.4)

```
Объект A: 123
Объект A: Alpha
Объект A: A
Объект B: 123
Объект B: Bravo
Объект B: B
```

В данном случае описывается класс MyClass, который содержит текстовое поле name, конструктор с текстовым аргументом, а также обобщенный метод show().

Методу передается аргументом значение обобщенного типа (обозначен как X), а при выполнении метода отображается значение поля name и, через двоеточие, значение аргумента метода.

В главном методе программы создается два объекта класса MyClass, а затем из каждого объекта вызывается метод show() с разными аргументами.

Обобщенные классы и наследование

— Нет, ей об этом думать еще рано.
— Об этом думать никому не рано, и никогда не поздно.

Из к/ф «Кавказская пленница»

Далее обсудим некоторые аспекты, имеющие отношение к обобщенным классам и методам в контексте использования такого механизма, как наследование.

Суперкласс на основе обобщенного класса

Конкретные реализации обобщенного суперкласса могут использоваться в качестве суперкласса при наследовании. Проще говоря, при создании подкласса в качестве суперкласса можно указать обобщенный класс — но не как таковой, а с конкретными значениями для обобщенных параметров. Пример такого подхода представлен в программе в листинге 9.5.

Листинг 9.5. Программный код проекта ExtendingGenClassApplication

```
// Обобщенный класс:
class Base<X>{
  // Обобщенное поле:
  X data;
  // Конструктор:
  Base(X data){
    this.data=data;
  }
  // Метод для отображения значения поля:
  void show(){
    System.out.println(data);
  }
}
// Подкласс на основе обобщенного класса с целочисленным
// типом вместо обобщенного:
class Alpha extends Base<Integer>{
  // Конструктор:
  Alpha(Integer n){
    // Вызов конструктора суперкласса:
    super(n);
  }
  // Переопределение метода:
  void show(){
    System.out.print("Целочисленное поле: ");
    // Вызов версии метода из суперкласса:
```

```
    super.show();
  }
}
// Подкласс на основе обобщенного класса с текстовым
// типом вместо обобщенного:
class Bravo extends Base<String>{
  // Конструктор:
  Bravo(String txt){
    // Вызов конструктора суперкласса:
    super(txt);
  }
  // Переопределение метода:
  void show(){
    System.out.print("Текстовое поле: ");
    // Вызов версии метода из суперкласса:
    super.show();
  }
}
// Подкласс на основе обобщенного класса с символьным
// типом вместо обобщенного:
class Charlie extends Base<Character>{
  // Конструктор:
  Charlie(Character s){
    // Вызов конструктора суперкласса:
    super(s);
  }
  // Переопределение метода:
  void show(){
    System.out.print("Символьное поле: ");
    // Вызов версии метода из суперкласса:
    super.show();
  }
}
```

```
// Главный класс:
class ExtendingGenClassDemo{
  // Главный метод:
  public static void main(String[] args){
    // Создание объектов:
    Alpha A=new Alpha(123);
    Bravo B=new Bravo("объект B");
    Charlie C=new Charlie('C');
    // Вызов метода show() из разных объектов:
    A.show();
    B.show();
    C.show();
  }
}
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 9.5)

```
Целочисленное поле: 123
Текстовое поле: объект B
Символьное поле: C
```

Мы использовали очень простую схему: описали обобщенный класс Base с обобщенным параметром X, а в классе описали поле data типа X, метод show(), при вызове которого отображается значение поля data, а также у класса есть конструктор с одним аргументом обобщенного типа. На основе класса Base путем наследования создаются классы Alpha, Bravo и Charlie. При этом в качестве обобщенного параметра указываются соответственно типы Integer, String и Character соответственно. Также в каждом из классов переопределяется метод show(). При этом с помощью инструкции super.show() вызывается версия метода из обобщенного суперкласса Base.

Таким образом, в классе Alpha типом поля data является Integer, в классе Bravo типом поля data является String, а в классе Charlie типом поля data является Character. Объекты данных классов создаются в главном методе программы, а затем из каждого из них вызывается метод show().

ДЕТАЛИ

Допустим, что обобщенный суперкласс описан следующим образом:

```
class Base<X>{
  X data;
}
```

Рассмотрим два подкласса. Подкласс Alpha описан так:

```
class Alpha extends Base<Integer>{}
```

Еще один подкласс Bravo опишем следующим образом:

```
class Bravo extends Base{}
```

Принципиальное отличие классов Alpha и Bravo в том, что при создании Alpha для обобщенного суперкласса Base указано значение Integer для обобщенного типа, а при создании класса Bravo значение для обобщенного типа не указано. В результате у объектов класса Alpha поле data будет целочисленным (типа Integer), а у объектов класса Bravo поле data будет относиться к типу Object (суперкласс в вершине иерархии всех классов).

Ограничение наследования для обобщенного типа

При использовании обобщенных классов и методов в инструкции объявления обобщенного типа может использоваться выражение вида X extends Суперкласс, означающее, что обобщенный тип X должен быть таким, что является подклассом (прямым или опосредованным) класса Суперкласс. В некоторых случаях такой подход бывает достаточно удобен. Небольшая иллюстрация представлена в листинге 9.6.

Листинг 9.6. Программный код проекта GenTypeExtendingApplication

```
// Суперкласс:
class Base{
  // Текстовое поле:
  String name;
  // Конструктор:
  Base(String txt){
    name=txt;
  }
  // Метод для отображения значения поля:
  void show(){
    System.out.println("Текстовое поле: "+name);
```

```
  }
}
// Подкласс суперкласса Base:
class Alpha extends Base{
  // Целочисленное поле:
  int number;
  // Конструктор:
  Alpha(String txt,int n){
    // Вызов конструктора суперкласса:
    super(txt);
    // Присваивание значения целочисленному полю:
    number=n;
  }
  // Переопределение метода:
  void show(){
    // Вызов версии метода из суперкласса:
    super.show();
    // Отображение значения целочисленного поля:
    System.out.println("Целочисленное поле: "+number);
  }
}
// Подкласс суперкласса Alpha:
class Bravo extends Alpha{
  // Символьное поле:
  char symbol;
  // Конструктор:
  Bravo(String txt,int n,char s){
    // Вызов конструктора суперкласса:
    super(txt,n);
    // Присваивание значения символьному полю:
    symbol=s;
  }
  // Переопределение метода:
  void show(){
```

```
    // Вызов версии метода из суперкласса:
    super.show();
    // Отображение значения символьного поля:
    System.out.println("Символьное поле: "+symbol);
  }
}
// Обобщенный класс:
class MyClass<X extends Base>{
  // Поле обобщенного типа:
  X obj;
  // Конструктор:
  MyClass(X obj){
    // Значение поля:
    this.obj=obj;
  }
  // Метод обобщенного класса:
  void show(){
    System.out.println("Объект класса MyClass");
    // Вызов метода из поля обобщенного типа:
    obj.show();
  }
}
// Главный класс:
class GenTypeExtendingDemo{
  // Главный метод:
  public static void main(String[] args){
    // Объекты создаются на основе обобщенного класса:
    MyClass<Alpha> A=new MyClass<>(new Alpha("Alpha",123));
    MyClass<Bravo> B=new MyClass<>(new Bravo("Bravo",321,'B'));
    // Вызов метода show() из объектов:
    A.show();
    B.show();
  }
}
```

В программе описывается класс Base с текстовым полем name, конструктором и методом show() для отображения значения поля. На основе класса Base создается класс Alpha. В классе Alpha к полю name добавляется целочисленное поле number. Там же переопределяется метод show(). На основе класса Alpha создается класс Bravo. В классе Bravo дополнительно появляется символьное поле symbol и соответствующим образом переопределяется метод show().

НА ЗАМЕТКУ

Инструкция super.show() в классе Alpha означает вызов версии метода show() из класса Base, поскольку он является суперклассом для класса Alpha. Инструкция super.show() в классе Bravo означает вызов версии метода show() из класса Alpha, поскольку для класса Bravo суперклассом является класс Alpha.

Обобщенный класс MyClass описывается с инструкцией <X extends Base>, означающей, что значением обобщенного параметра типа X при создании объекта класса MyClass может быть лишь класс, который является прямым или опосредованным (через несколько классов) наследником класса Base. Классы Alpha и Bravo для этой цели подходят.

В классе MyClass объявлено поле obj обобщенного типа X. Значение полю присваивается в конструкторе класса MyClass. Аргументом конструктору передается значение типа X, и именно это значение присваивается полю obj.

ДЕТАЛИ

Полю obj присваивается значение, переданное аргументом конструктору класса MyClass. Учитывая, что значением параметра X может быть имя класса (наследующего Base), легко понять, что аргументом конструктору передается объектная переменная, значением которой является ссылка на некоторый объект и, следовательно, полю obj присваивается ссылка на этот объект. Важно здесь то, что объект для поля obj не создается в конструкторе, а ссылка на него лишь передается в конструктор. Это принципиально важный момент, поскольку в обобщенном классе нельзя создать объект на основе обобщенного типа.

В классе MyClass описывается метод show(), в котором вызывается метод show() из поля-объекта obj. Здесь нужно отметить два момента. Во-первых,

о переопределении метода show() речь не идет, поскольку один метод определяется для класса MyClass, а другой вызывается из объекта obj обобщенного типа X. Во-вторых, мы можем вызвать метод show() из объекта obj обобщенного типа X только потому, что обобщенный параметр X описан как «наследующий» класс Base, в котором есть метод show(). Если бы вместо инструкции <X extends Base> в описании обобщенного класса MyClass мы использовали инструкцию <X>, то при компиляции команды obj.show() в теле метода show() (в классе MyClass) возникла бы ошибка.

В главном методе программы командами MyClass<Alpha> A=new MyClass<>(new Alpha("Alpha",123)) и MyClass<Bravo> B=new MyClass<>(new Bravo("Bravo",321,'B')) на основе обобщенного класса MyClass создаются объект A и B. Объект A создается на основе класса MyClass с использованием в качестве обобщенного параметра типа класса Alpha. Объект B создается на основе класса MyClass с использованием в качестве обобщенного параметра класса Bravo. В обоих случаях аргументом конструктору класса MyClass передается анонимный объект: в первом случае речь идет об объекте класса Alpha, а во втором случае аргументом передается анонимный объект класса Bravo. Именно на эти объекты будет содержаться ссылка в поле obj для объектов A и B соответственно.

После создания объектов A и B командами A.show() и B.show() из каждого объекта вызывается метод show(). В результате при выполнении программы получаем такой результат:

Результат выполнения программы (из листинга 9.6)

```
Объект класса MyClass
Текстовое поле: Alpha
Целочисленное поле: 123
Объект класса MyClass
Текстовое поле: Bravo
Целочисленное поле: 321
Символьное поле: B
```

Таким образом, использование инструкции объявления обобщенного типа с ключевым словом extends не только ограничивает «спектр» возможных значений обобщенного типа, но и «конкретизирует» тип используемых в обобщенном классе данных. А это может быть очень кстати.

Обобщенные интерфейсы

Наше повеление. Этот танец не вяжется с королевской честью. Мы запрещаем его на веки веков.

Из к/ф «31 июня»

Обобщенным может быть не только класс или метод, но и интерфейс. Объявляется обобщенный интерфейс аналогично к обобщенному классу: после имени интерфейса в угловых скобках указываются названия для обобщенных типов. Такой обобщенный интерфейс может использоваться как для создания обобщенного класса, так и для создания класса с конкретными значениями для обобщенных параметров типа.

Создание обобщенного класса на основе интерфейса

Если на основе обобщенного интерфейса создается обобщенный класс, то обобщенный параметр типа указывается после имени класса и после имени реализуемого в классе интерфейса. Пример такой ситуации представлен в листинге 9.7.

Листинг 9.7. Программный код проекта GenInterfaceApplication

```
// Обобщенный интерфейс:
interface MyMethods<X>{
  X get();
  void set(X arg);
}
// Создание обобщенного класса на основе
// обобщенного интерфейса:
class MyClass<X> implements MyMethods<X>{
  // Закрытое поле обобщенного типа:
  private X value;
  // Описание методов из интерфейса:
  public X get(){
    return value;
  }
  public void set(X arg){
```

```
    value=arg;
  }
  // Метод для отображения значения поля:
  void show(){
    System.out.println("Значение поля: "+get());
  }
  // Конструктор:
  MyClass(X arg){
    value=arg;
  }
}
// Главный класс:
class GenInterfaceDemo{
  // Главный метод:
  public static void main(String[] args){
    // Интерфейсная переменная:
    MyMethods ref;
    // Создание объектов на основе обобщенного класса:
    MyClass<Integer> A=new MyClass<>(123);
    MyClass<Character> B=new MyClass<>('A');
    // Вызов метода из объекта класса:
    A.show();
    // Интерфейсной переменной присваивается значение:
    ref=A;
    // Вызов метода через интерфейсную переменную:
    ref.set(321);
    // Вызов методов из объектов класса:
    A.show();
    B.show();
    // Интерфейсной переменной присваивается значение:
    ref=B;
    // Вызов метода через интерфейсную переменную:
    ref.set('B');
```

```
    // Вызов метода из объекта класса:
    B.show();
  }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 9.7)

```
Значение поля: 123
Значение поля: 321
Значение поля: A
Значение поля: B
```

Код программы достаточно простой. В обобщенном интерфейсе MyMethods параметр X обозначает обобщенный тип данных. В интерфейсе объявлены два метода: метод get() не имеет аргументов и возвращает результатом значение обобщенного типа X, а метод set() не возвращает результат, а аргументом методу передается значение обобщенного типа X.

Обобщенный интерфейс MyMethods реализуется в обобщенном классе MyClass. В этом классе описано закрытое поле обобщенного типа, а методы set() и get() определены так, что ими соответственно присваивается значение полю и возвращается значение поля. Также в классе описан конструктор с одним аргументом и метод show(), которым отображается значение закрытого поля.

В главном методе программы объявляется интерфейсная переменная ref интерфейса MyMethods, а также создаются переменные A и B на основе обобщенного класса MyClass соответственно со значениями Integer и Character для обобщенного параметра типа. Для каждого из объектов выполняется такая процедура:

- из объекта вызывается метод show();
- переменной ref присваивается ссылка на соответствующий объект;
- через интерфейсную переменную ref вызывается метод set(), благодаря чему закрытому полю объекта присваивается новое значение;
- через объектную переменную снова вызывается метод show().

Результат всех этих действий приведен выше.

Создание обычного класса на основе обобщенного интерфейса

На основе обобщенного интерфейса можно создавать обычные классы. Небольшая программная вариация, отдаленно напоминающая предыдущий пример. Рассмотрим программный код в листинге 9.8.

Листинг 9.8. Программный код проекта MoreGenInterfaceApplication

```
// Обобщенный интерфейс:
interface MyMethods<X>{
  X get();
  void set(X arg);
}
// Создание первого класса на основе
// обобщенного интерфейса:
class Alpha implements MyMethods<Integer>{
  // Закрытое поле целочисленного типа:
  private Integer value;
  // Описание методов из интерфейса:
  public Integer get(){
    return value;
  }
  public void set(Integer arg){
    value=arg;
  }
  // Метод для отображения значения поля:
  void show(){
    System.out.println("Целочисленное поле: "+get());
  }
  // Конструктор:
  Alpha(Integer arg){
    value=arg;
  }
}
```

```
// Создание второго класса на основе
// обобщенного интерфейса:
class Bravo implements MyMethods<Character>{
  // Закрытое поле символьного типа:
  private Character value;
  // Описание методов из интерфейса:
  public Character get(){
    return value;
  }
  public void set(Character arg){
    value=arg;
  }
  // Метод для отображения значения поля:
  void show(){
    System.out.println("Символьное поле: "+get());
  }
  // Конструктор:
  Bravo(Character arg){
    value=arg;
  }
}
// Главный класс:
class MoreGenInterfaceDemo{
  // Главный метод:
  public static void main(String[] args){
    // Интерфейсная переменная:
    MyMethods ref;
    // Создание объектов:
    Alpha A=new Alpha(123);
    Bravo B=new Bravo('A');
    // Вызов метода из объекта класса:
    A.show();
    // Интерфейсной переменной присваивается значение:
```

```
    ref=A;
    // Вызов метода через интерфейсную переменную:
    ref.set(321);
    // Вызов методов из объектов класса:
    A.show();
    B.show();
    // Интерфейсной переменной присваивается значение:
    ref=B;
    // Вызов метода через интерфейсную переменную:
    ref.set('B');
    // Вызов метода из объекта класса:
    B.show();
  }
}
```

Ниже показано, каким будет результат выполнения программы.

Результат выполнения программы (из листинга 9.8)

```
Целочисленное поле: 123
Целочисленное поле: 321
Символьное поле: A
Символьное поле: B
```

Как видим, результат выполнения программы такой же, как и в предыдущем случае.

ДЕТАЛИ

Хотя интерфейс MyMethods является обобщенным, интерфейсная переменная ref объявляется командой MyMethods ref, в которой указано только основное имя интерфейса без указания значения для обобщенного типа. Благодаря этому значением переменной ref можно присвоить ссылку как на объект A, так и на объект B — несмотря на то, что объект A создавался на основе класса Alpha, реализующего интерфейс MyMethods со значением Integer для обобщенного типа, а объект B создается на основе класса Bravo, реализующего интерфейс MyMethods со значением Character для обобщенного типа. У такой «универсальности»

есть цена: обобщенный тип при обращении к объектам через интерфейсную переменную ref не специфицирован. Поэтому, например, метод get(), который в соответствии со своим объявлением в интерфейсе, возвращает значение обобщенного типа, а для объектов A и B описан возвращающим соответственно значения типа Integer и Character, при вызове его через переменную ref интерпретируется как возвращающий значение типа Object. Это не мешает, скажем, отображать в консольном окне значение, возвращаемое методом get(), но накладывает ограничения на допустимые операции с результатом метода (правда, обычно на выручку может прийти явное приведение типа).
С другой стороны, мы могли бы, скажем, объявить переменную ref с помощью команды MyMethods<Integer> ref, в которой для интерфейса MyMethods явно указано значение Integer для параметра обобщенного типа. Но в таком случае значением переменной ref можно было бы присвоить лишь ссылку на объект A, но не на объект B. Зато при вызове метода get() через переменную ref результат метода интерпретируется как значение типа Integer.

Обобщенные подстановки

Что бы мы делали без науки? Подумать страшно!

Из к/ф «31 июня»

Использование параметра обобщенного типа в классе или методе подразумевает, что при создании объекта класса или вызове метода происходит определение (идентификация) обобщенного типа. Но нередко значение для обобщенного параметра играет формальную роль. В таких случаях удобно использовать *обобщенные подстановки*.

НА ЗАМЕТКУ

На самом деле тема обобщенных подстановок многогранна и достаточно нетривиальна. Здесь мы во многом поверхностно рассматриваем только некоторые вводные аспекты данной темы.

Знакомство с обобщенными подстановками

Идея обобщенной подстановки в принципе проста: используя в качестве определения обобщенного параметра инструкцию вида <?> можно сообщить компилятору, что в соответствующем месте используется

некоторый неизвестный обобщенный тип. Например, представим себе такую ситуацию: в программе определен обобщенный класс MyClass с одним обобщенным параметром (обозначим его как T). В классе объявлено поле value обобщенного типа T и конструктор с одним аргументом обобщенного типа. Аргумент конструктора присваивается значением полю value.

В классе UsingWildcardDemo описывается главный метод main() и два статических не возвращающих результат метода: show() и display(). Оба метода выполняют практически одинаковую «работу», но описаны по-разному. Оба метода предназначены для отображения значения поля value объекта, созданного на основе обобщенного класса MyClass и переданного аргументом методу. В главном методе программы приведены примеры создания объектов на основе обобщенного класса MyClass с последующей их передачей аргументами методам show() и display().

Листинг 9.9. Программный код проекта UsingWildcardApplication

```
// Обобщенный класс:
class MyClass<T>{
  // Поле обобщенного типа:
  T value;
  // Конструктор:
  MyClass(T val){
    value=val;
  }
}
// Главный класс:
class UsingWildcardDemo{
  // Статический обобщенный метод
  // (используется параметр обобщенного типа):
  static <T> void show(MyClass<T> obj){
    System.out.println("Вызов метода show():");
    // Отображение значения поля:
    System.out.println(obj.value);
  }
  // Статический метод, в котором используется
  // обобщенная подстановка:
```

```
  static void display(MyClass<?> obj){
    System.out.println("Вызов метода display():");
    // Отображение значения поля:
    System.out.println(obj.value);
  }
  // Главный метод:
  public static void main(String[] args){
    // При создании объекта явно указано значение
    // для обобщенного типа:
    MyClass<Integer> A=new MyClass<>(100);
    // При создании объекта не указано значение
    // для обобщенного типа:
    MyClass B=new MyClass<>('B');
    // При создании объекта использована
    // обобщенная подстановка:
    MyClass<?> C=new MyClass<>("Объект C");
    // Примеры вызова методов show() и display()
    // с разными аргументами:
    show(A);
    display(A);
    show(B);
    display(B);
    show(C);
    display(C);
  }
}
```

Итак, метод show() описан как обобщенный. Параметр обобщенного типа обозначен как T. Тип аргумента метода определен выражением MyClass<T>. Сам аргумент обозначен как obj. В теле метода отображается текстовое сообщение и значение поля value объекта obj. Таким образом, обобщенный тип T здесь в явном виде не используется. Это «намек» на то, что можно использовать обобщенную подстановку. Такой подход использован при описании метода display(). Аргумент obj этого метода описан с типом

MyClass<?>. Данная инструкция означает, что аргумент является объектом обобщенного класса MyClass с некоторым значением для параметра обобщенного типа, но это значение не конкретизируется (не вычисляется). Можно сказать и так: какой-то тип использован, но какой именно — не важно. Все остальное в методе display() практически такое же, как в методе show().

В главном методе программы создается три объекта класса MyClass. При создании объекта A значение для обобщенного параметра типа указано явно. При создании объекта B значение для обобщенного параметра типа не указано совсем, а при создании объекта C использована обобщенная подстановка: в угловых скобках после имени класса MyClass указан вопросительный знак (инструкция <?>). Затем вызываются методы show() и display() с передачей аргументами методам объектов A, B и C. В результате выполнения программы получаем такое:

Результат выполнения программы (из листинга 9.9)

```
Вызов метода show():
100
Вызов метода display():
100
Вызов метода show():
B
Вызов метода display():
B
Вызов метода show():
Объект C
Вызов метода display():
Объект C
```

Следует отметить разницу между способами объявления объектов A, B и C. В частности, поскольку при создании объекта A мы явно указали значение Integer для параметра обобщенного типа, то поле value созданного объекта интерпретируется как Integer-значение.

При создании объекта B для объектной переменной указан только основной тип — имя класса MyClass. Поэтому поле value созданного объекта интерпретируется как значение типа Object.

Что касается объекта C, при создании которого использована обобщенная подстановка, то для этого объекта значение обобщенного параметра не идентифицируется. Технически в таком случае используется формальное «рабочее» обозначение для значения параметра обобщенного типа — то есть для параметра обобщенного типа выполняется «подстановка», которая играет роль типа.

Во всех трех случаях мы можем отобразить значение поля, но операции, допустимые для выполнения с полем value, зависят от способа создания объекта и различны для объектов A, B и C.

Обобщенные подстановки с ограничениями

При использовании обобщенных подстановок можно использовать *ограничения*: с помощью ключевого слова extends определяют подстановки, обозначающие подклассы для данного класса, а с помощью ключевого слова super определяют подстановки для суперклассов данного класса. В частности, выражение вида <? extends SuperClass> означает, что для параметра обобщенного типа используется неустановленный тип, являющийся подклассом (прямым или опосредованным) класса SuperClass. Выражение вида <? super SubClass> означает, что в качестве параметра обобщенного типа может быть класс, являющийся (прямо или через цепочку наследования) суперклассом для класса SubClass. Небольшой пример использования таких обобщенных подстановок приведен в листинге 9.10.

Листинг 9.10. Программный код проекта BoundedWildcardsApplication

```
// Первый класс:
class Alpha{
  // Закрытое текстовое поле:
  private String name;
  // Конструктор:
  Alpha(String txt){
    name=txt;
  }
  // Переопределение метода toString():
  public String toString(){
    return name;
  }
```

```
}
// Второй класс:
class Bravo extends Alpha{
  // Конструктор:
  Bravo(String txt){
    // Вызов конструктора суперкласса:
    super(txt);
  }
}
// Третий класс:
class Charlie extends Bravo{
  Charlie(String txt){
    super(txt);
  }
}
// Четвертый класс:
class Delta extends Charlie{
  Delta(String txt){
    super(txt);
  }
}
// Пятый класс:
class Echo extends Delta{
  Echo(String txt){
    super(txt);
  }
}
// Обобщенный класс:
class MyClass<T>{
  // Закрытое поле обобщенного типа:
  private T obj;
  // Переопределение метода toString():
  public String toString(){
```

```
    return obj.toString();
  }
  // Конструктор:
  MyClass(T arg){
    obj=arg;
  }
}
// Главный класс:
class BoundedWildcardsDemo{
  // Статический метод для отображения текстового
  // представления объекта, созданного с использованием
  // подкласса для класса Charlie:
  static void show(MyClass<? extends Charlie> obj){
    System.out.println(obj);
  }
  // Статический метод для отображения текстового
  // представления объекта, созданного с использованием
  // суперкласса для класса Charlie:
  static void display(MyClass<? super Charlie> obj){
    System.out.println(obj);
  }
  // Главный метод:
  public static void main(String[] args){
    // Создание объектов:
    MyClass<Alpha> A=new MyClass<>(new Alpha("Объект A"));
    MyClass<Bravo> B=new MyClass<>(new Bravo("Объект B"));
    MyClass<Charlie> C=new MyClass<>(new Charlie("Объект C"));
    MyClass<Delta> D=new MyClass<>(new Delta("Объект D"));
    MyClass<Echo> E=new MyClass<>(new Echo("Объект E"));
    // Вызов методов display() и show() с передачей
    // аргументом одного из созданных объектов:
    display(A);
    display(B);
```

```
    display(C);
    show(C);
    show(D);
    show(E);
  }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 9.10)

```
Объект A
Объект B
Объект C
Объект C
Объект D
Объект E
```

Программный код хоть и немного длинный, но все же простой. В программе создается пять классов (Alpha, Bravo, Charlie, Delta и Echo): класс Bravo является подклассом класса Alpha, класс Charlie является подклассом класса Bravo, и так далее — вплоть до класса Echo. В классе Alpha описано закрытое текстовое поле name, и значение полю присваивается при вызове конструктора. Метод toString() в классе Alpha переопределен так, что результатом возвращается значение текстового поля name. Во всех остальных классах в цепочке наследования просто определяется конструктор. Поэтому все классы однотипны: объект каждого из перечисленных выше пяти классов создается с передачей конструктору текстового значения, и это значение используется при преобразовании объекта к текстовому значению.

Обобщенный класс MyClass описан с одним параметром обобщенного типа (обозначен как T). В классе есть закрытое поле obj обобщенного типа, конструктор с одним аргументом, определяющим значение поля obj, а также переопределен метод toString(), результатом которого возвращается выражение obj.toString() — результат преобразования объекта obj к текстовому формату (здесь мы в явном виде вызываем метод toString() из объекта obj).

В главном классе BoundedWildcardsDemo описываются, кроме главного метода, еще и статические методы show() и display(). Тип аргумента для метода show() описан выражением MyClass<? extends Charlie>, означающим, что аргументом методу можно передавать лишь такие объекты обобщенного класса MyClass, которые создавались с использованием в качестве значения параметра обобщенного типа класса, являющегося подклассом класса Charlie. Под этот критерий подпадают классы Charlie, Delta и Echo. В методе display() тип аргумента определяется выражением MyClass<? super Charlie>. Аргументом методу display() может передаваться объект, созданный на основе обобщенного класса MyClass с использованием в качестве значения для параметра обобщенного типа имени класса, являющегося суперклассом (прямо или через цепочку наследования) для класса Charlie. Данному критерию удовлетворяют классы Alpha, Bravo и Charlie.

В главном методе программы с использованием классов Alpha, Bravo, Charlie, Delta и Echo и обобщенного класса MyClass создаются объекты A, B, C, D и E, а затем данные объекты передаются аргументами методам display() и show(). С учетом указанных выше ограничений, методу display() аргументом можно передавать только объекты A, B и C, а аргументом методу show() можно передавать только объекты C, D и E.

ДЕТАЛИ

Напомним, что при работе с обобщенными классами отсутствует возможность в теле класса создавать объект класса, определяемого параметром обобщенного типа. При создании объекта на основе класса MyClass аргументом конструктору передается анонимный объект одного из классов Alpha, Bravo, Charlie, Delta или Echo. Но на самом деле здесь объект создается в явном виде вне тела обобщенного класса, а ссылка на объект передается аргументом конструктору.

Резюме

Вы видите, мистер Холмс, как я ловко расставил сеть. И сеть, как вы видите, сужается.

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

- В обобщенных классах, методах и интерфейсах тип может передаваться через параметр. Параметры обобщенного типа для обобщенного класса или интерфейса указываются в угловых скобках после

имени класса. Параметры обобщенного типа для обобщенного метода указываются в угловых скобках перед идентификатором типа результата метода. Параметры типа могут использоваться в теле обобщенного класса, интерфейса или метода.

- При создании объекта на основе обобщенного класса значения для параметров обобщенного типа, используемые при создании объекта, указываются в угловых скобках после основного имени обобщенного класса. Значениями параметров обобщенного типа могут быть только ссылочные типы (классы), и ими не могут быть базовые (примитивные) типы. Значения параметров обобщенного типа при вызове обобщенного метода определяются исходя из контекста команды вызова метода.
- На параметры обобщенного типа можно накладывать ограничение. С использованием ключевого слова `extends` значения для параметра обобщенного типа можно ограничить подклассами определенного класса.
- Класс может создаваться на основе обобщенного интерфейса.
- Обобщенная подстановка позволяет задекларировать использование обобщенного типа, не конкретизируя его значение. С использованием ключевых слов `extends` и `super` можно ограничить возможные значения для типа, определяемого через обобщенную подстановку: соответственно, возможные значения параметра типа ограничиваются подклассами определенного класса или суперклассами (по цепочке наследования) определенного класса.
- Существует ряд ограничений, которые следует иметь в виду при работе с обобщенными классами и методами. Так, например, в теле обобщенного класса нельзя создать объект на основе класса, определяемого обобщенным параметром (но можно использовать объектную переменную соответствующего класса).

Глава 10
ЛЯМБДА-ВЫРАЖЕНИЯ

— Ладно, все. Надо что-то делать. Давай-ка, может быть, сами изобретем.
— Витя, не надо! Я прошу тебя. Не дразни начальство!

Из к/ф «Чародеи»

В этой главе обсуждаются *лямбда-выражения*. Это новая технология в Java 8, не самая тривиальная, но достаточно интересная. Собственно, ее мы и обсудим далее.

Знакомство с лямбда-выражениями

— Ученый совет должен быть в полном составе!
— Кота ученого приглашать будем?

Из к/ф «Чародеи»

В отношении лямбда-выражений нам предстоит выяснить два момента:

- во-первых, желательно понять, что такое лямбда-выражение;
- во-вторых, важно уяснить, как лямбда-выражения можно использовать.

Хотя вопросы, на которые нам предстоит дать ответы, достаточно простые, ответы на них весьма неординарные.

Синтаксис лямбда-выражения

Лямбда-выражение можно рассматривать как некоторый блок кода, определяющий *действие*. Также можно рассматривать лямбда-выражение как определение *анонимного метода* — то есть метода, у которого нет имени (аналогия не совсем точная, то во многих случаях помогает понять логику происходящего). Используются лямбда-выражения в основном как альтернатива к созданию анонимных классов.

Чтобы разобраться со способами описания лямбда-выражений, удобно исходить из представления о лямбда-выражении как об описании некоторого метода, у которого нет имени и не указывается тип возвращаемого результата. Синтаксис описания лямбда-выражения подразумевает, что задаются аргументы, с которыми выполняются определенные операции, и собственно сами операции или команды. Блок из команд в лямбда-выражении для удобства будем называть телом лямбда-выражения. Соответственно, лямбда-выражение состоит из аргументов и тела. То есть здесь все очень близко к тому, как описывается метод, но только нет названия и идентификатора типа результата. В общем случае аргументы в лямбда-выражении описываются в круглых скобках. Если аргументов несколько, то между собой они разделяются запятыми. Для аргументов указывается тип (хотя может и не указываться — это мы еще обсудим). Команды в теле лямбда-выражения разделяются между собой точкой с запятой, а весь блок команд заключается в фигурные скобки. Между аргументами и блоком команд (телом лямбда-выражения) указывается «стрелка» ->. Таким образом, используется следующий шаблон описания лямбда-выражения:

```
(аргументы)->{команды}
```

Например, ниже приведен пример описания лямбда-выражения с двумя аргументами целочисленного типа, а тело выражения состоит из двух команд, которыми должна вычисляться сумма аргументов и результат вычислений выводится в консольное окно:

```
(int x,int y)->{int z=x+y; System.out.println(z);}
```

Если для большего удобства команды в теле лямбда-выражения разместить построчно, то внешнее сходство лямбда-выражения с описанием метода становится особенно очевидным:

```
(int x,int y)->{
  int z=x+y;
  System.out.println(z);
}
```

Правда, здесь мы пока не затрагиваем вопрос об использовании лямбда-выражений, поскольку эта тема отдельная. То есть мы не можем так же просто использовать лямбда-выражение, как обычный метод. Но формально с точки зрения синтаксиса аналогия есть.

Есть некоторые правила, которые позволяют упрощать синтаксис описания лямбда-выражений. Вот они.

- Тип аргумента можно не указывать, если он быть идентифицируется исходя из контекста команды использования лямбда-выражения.
- Если в лямбда-выражении один аргумент, тип которого не указан, то круглые скобки можно не использовать.
- Если в лямбда-выражении нет аргументов, то используются пустые круглые скобки.
- Если в теле лямбда-выражения всего одна команда, то фигурные скобки можно не использовать. В случае, когда единственная команда в теле лямбда-оператора состоит из return-инструкции, ключевое слово return можно не использовать.

Дальнейшее обсуждение лямбда-выражений имеет смысл продолжать в контексте их использования. А для этого нам придется познакомиться еще с «конструкцией» языка Java — *функциональными интерфейсами*.

Функциональные интерфейсы

Функциональный интерфейс — это интерфейс с одним и только одним абстрактным методам. То есть ничего особенного в функциональном интерфейсе нет. Просто должно быть выдержано одно условие: в интерфейсе объявляется всего один абстрактный метод.

ДЕТАЛИ

В функциональном интерфейсе методов может быть больше, чем один. Ограничение в один метод относится именно к абстрактным методам. Здесь имеет смысл напомнить, что в соответствии с новыми стандартами языка Java в интерфейсах можно не только объявлять методы, но и задавать для методов код по умолчанию. В функциональном интерфейсе должен быть только один абстрактный метод — то есть метод без кода по умолчанию. Количество методов с кодом по умолчанию в функциональном интерфейсе не регламентируется.

Если описывается функциональный интерфейс, в строке над описанием интерфейса может использоваться аннотация @FunctionalInterface. Наличие такой аннотации перед описанием интерфейса приводит к тому, что на этапе компиляции программного кода выполняется проверка на предмет того, действительно ли интерфейс функциональный. Если не так, то возникает ошибка.

Особенность функционального интерфейса в плане использования лямбда-выражений связана с тем, что лямбда-выражение может быть присвоено значением переменной интерфейсного типа — но при условии, что интерфейс функциональный, а сигнатура абстрактного метода в функциональном интерфейсе соответствует параметрам лямбда-выражения (количество и тип параметров).

Что же происходит при присваивании интерфейсной переменной значением лямбда-выражения? А происходит следующее: создается объект на основе класса (анонимного), реализующего данный интерфейс, а в качестве кода для определения абстрактного метода используется код лямбда-выражения. Простой пример использования функционального интерфейса и лямбда-выражений представлен в листинге 10.1.

Листинг 10.1. Программный код проекта UsingLambdaApplication

```
// Функциональный интерфейс:
interface MyNums{
  // Метод с кодом по умолчанию:
  default void show(int n){
    System.out.println("Аргумент: "+n);
    System.out.println("Результат: "+get(n));
  }
  // Абстрактный метод:
  int get(int n);
}
// Главный класс:
class UsingLambdaDemo{
  // Главный метод:
  public static void main(String[] args){
    // Присваивание лямбда-выражения значением
    // переменной интерфейсного типа:
    MyNums A=(int n)->{
      // Локальные переменные:
      int k,s=0;
      // Вычисление суммы натуральных чисел:
      for(k=1;k<=n;k++){
```

```
        s+=k;
      }
      // Результат:
      return s;
    };
    System.out.println("Используем лямбда-выражение:");
    // Вызов методов из интерфейсной переменной:
    A.show(10);
    System.out.println("Проверка: "+A.get(10));
    // Альтернативная ссылка на объект:
    MyNums B=A;
    System.out.println("Новое лямбда-выражение:");
    // Интерфейсной переменой значением присваивается
    // новое лямбда-выражение:
    A=n->n*n;
    System.out.println("Вызов метода show() из A:");
    A.show(10);
    System.out.println("Вызов метода show() из B:");
    B.show(10);
  }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 10.1)

```
Используем лямбда-выражение:
Аргумент: 10
Результат: 55
Проверка: 55
Новое лямбда-выражение:
Вызов метода show() из A:
Аргумент: 10
Результат: 100
Вызов метода show() из B:
```

```
Аргумент: 10
Результат: 55
```

Проанализируем программный код и результаты его выполнения. Важное место занимает описание функционального интерфейса MyNums. Функциональным он имеет право называться, поскольку в нем ровно один абстрактный метод. Абстрактный метод называется get(), у него целочисленный аргумент и он возвращает целочисленный результат. Кроме абстрактного метода get(), в интерфейсе MyNums описывается метод show() с кодом по умолчанию. Метод show() с кодом по умолчанию не возвращает результат, а аргументом ему передается целое число. При выполнении кода метода отображается значение целочисленного аргумента метода и результат, возвращаемый методом get(), вызванный с таким же аргументом, как и метод show().

В главном классе UsingLambdaDemo в методе main() выполняется такая команда (комментарии удалены):

```
MyNums A=(int n)->{
  int k,s=0;
  for(k=1;k<=n;k++){
    s+=k;
  }
  return s;
};
```

Этой командой объявляется переменная A интерфейсного типа MyNums, и значением этой переменной присваивается лямбда-выражение (все, что находится справа от оператора присваивания). Лямбда-выражение представляет собой код метода, вычисляющего сумму натуральных чисел. Верхняя граница суммы определяется целочисленным аргументом метода.

При присваивании значением переменной A лямбда-выражения происходит следующее.

- Автоматически создается анонимный класс, реализующий интерфейс MyNums. Для абстрактного кода get() «применяется» код, определяемый присваиваемым лямбда-выражением.
- На основе анонимного класса создается объект, ссылка на который присваивается переменной A.

Поэтому у объекта, на который ссылается переменная A, есть метод get(), вычисляющий сумму натуральных чисел и возвращающий вычисленное значение результатом. Также у объекта имеется метод show() (в котором вызывается метод get()). Вызов методов show() и get() через переменную A подтверждает сказанное.

Командой MyNums B=A объявляется переменная B интерфейсного типа MyNums. Значением переменной B присваивается ссылка на то же объект, на который ссылается переменная A. После этого командой A=n->n*n переменной A присваивается новое значение, и это значение определяется лямбда-выражением n->n*n. Данное лямбда-выражение определяет метод, который по указанному аргументу (обозначен как n), возвращается квадрат аргумента (значение n*n). При определении лямбда-выражения n->n*n мы, во-первых, не указали тип аргумента, во-вторых, не использовали круглые скобки для выделения аргументов, в-третьих, не использовали фигурные скобки для выделения кода с командами и, в-четвертых, не использовали ключевое слово return. При присваивании лямбда-выражения переменной A создается новый объект анонимного класса, реализующего интерфейс MyNums, но теперь метод get() определен так, что результатом метода возвращается квадрат аргумента. Что касается неуказанного типа аргумента в лямбда-выражении, то тип аргумента автоматически определяется на основе того обстоятельства, что в интерфейсе MyNums метод get() объявлен с целочисленным аргументом.

В итоге переменная A ссылается на объект, который создан на основе анонимного класса, реализующего интерфейс MyNums, в котором метод get() по указанному значению аргумента возвращает значением квадрат аргумента. Переменная B ссылается на объект, созданный на основе анонимного класса, реализующего интерфейс MyNums, в котором методом get() возвращается сумма натуральных чисел. Переменная B ссылается на объект, на который до присваивания ей нового значения n->n*n ссылалась переменная A.

Альтернативный подход

Чтобы более наглядно проиллюстрировать смысл происходящего при присваивании переменной интерфейсного типа значением лямбда-выражения, рассмотрим практически тот же пример, что и выше, но только на этот раз вместо лямбда-выражений используем анонимные объекты. Новая версия примера представлена в листинге 10.2 (основная часть комментариев удалена).

Листинг 10.2. Программный код проекта UsingAnonymousClassApplication

```
// Интерфейс:
interface MyNums{
  default void show(int n){
    System.out.println("Аргумент: "+n);
    System.out.println("Результат: "+get(n));
  }
  int get(int n);
}
class UsingAnonymousClassDemo{
  public static void main(String[] args){
    // Присваивание интерфейсной переменной ссылки
    // на объект анонимного класса:
    MyNums A=new MyNums(){
      // Описание метода из интерфейса:
      public int get(int n){
        int k,s=0;
        for(k=1;k<=n;k++){
          s+=k;
        }
        return s;
      }
    };
    System.out.println("Используем анонимный класс:");
    A.show(10);
    System.out.println("Проверка: "+A.get(10));
    MyNums B=A;
    System.out.println("Новый анонимный класс:");
    // Переменой значением присваивается ссылка
    // на объект нового анонимного класса:
    A=new MyNums(){
      // Описание метода из интерфейса:
      public int get(int n){
```

```
        return n*n;
      }
    };
    System.out.println("Вызов метода show() из A:");
    A.show(10);
    System.out.println("Вызов метода show() из B:");
    B.show(10);
  }
}
```

Ниже показано, как выглядит результат выполнения программы.

Результат выполнения программы (из листинга 10.2)

```
Используем анонимный класс:
Аргумент: 10
Результат: 55
Проверка: 55
Новый анонимный класс:
Вызов метода show() из A:
Аргумент: 10
Результат: 100
Вызов метода show() из B:
Аргумент: 10
Результат: 55
```

С точностью до некоторых текстовых значений, результат такой же, как в предыдущем примере (см. листинг 10.1). Вместе с тем совершенно очевидно, что использование лямбда-выражений вместо явного создания объектов на основе анонимных классов значительно упрощает программный код.

ДЕТАЛИ

На всякий случай напомним, как создается объект анонимного класса. Речь идет о классе, реализующем некоторый интерфейс (в рассмотренных примерах интерфейс `MyNums`). Итак, инструкция создания

объекта на основе анонимного класса начинается с ключевого слова new, после которого следует название реализуемого в классе интерфейса, круглые скобки, а в фигурных скобках — описание тех методов, которые объявлены в интерфейсе как абстрактные, и которые реализуются в данном классе. В применении к интерфейсу MyNums команда создания объекта на основе анонимного класса выглядит следующим образом:

```
new MyNums(){
  // Описание метода get()
}
```

Все это выражение присваивается значением интерфейсной переменной типа MyNums. Здесь используется свойство, состоящее в том, что интерфейсная переменная может содержать ссылку на объект класса, реализующего данный интерфейс.

Несколько интерфейсов и ссылка на метод

Одинаковые лямбда-выражения могут присваиваться переменным разных функциональных интерфейсов. Достаточно, чтобы абстрактный метод в каждом таком интерфейсе соответствовал параметрам (таким, как количество и тип аргументов) лямбда-выражения. Ниже приведен небольшой пример, в котором описывается три разных, но однотипных интерфейса, и в каждом из них объявляется метод, который не возвращает результат, и у которого нет аргументов. В главном методе программы объявляются интерфейсные переменные, значением которым присваивается одно и то же (по форме) лямбда-выражение ()->System.out.println("Используем лямбда-выражение"), соответствующее методу без аргументов, при выполнении которого в окне вывода отображается сообщение. Теперь рассмотрим программный код в листинге 10.3.

Листинг 10.3. Программный код проекта LambdaAndInterfacesApplication

```
// Первый функциональный интерфейс:
interface Alpha{
  void showA();
}
// Второй функциональный интерфейс:
interface Bravo{
  void showB();
```

```
}
// Третий функциональный интерфейс:
interface Charlie{
  void showC();
}
// Главный класс:
class LambdaAndInterfacesDemo{
  // Главный метод:
  public static void main(String[] args){
    // Значением интерфейсной переменной присваивается
    // лямбда-выражение:
    Alpha A=()->System.out.println("Используем лямбда-выражение");
    // Вызов метода из интерфейсной переменой:
    A.showA();
    // Значением интерфейсной переменной присваивается
    // лямбда-выражение:
    Bravo B=()->System.out.println("Используем лямбда-выражение");
    // Вызов метода из интерфейсной переменой:
    B.showB();
    // Значением интерфейсной переменной присваивается
    // ссылка на метод:
    Charlie C=A::showA;
    // Вызов метода из интерфейсной переменной:
    C.showC();
  }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 10.3)

```
Используем лямбда-выражение
Используем лямбда-выражение
Используем лямбда-выражение
```

Код программы простой. В интерфейсах Alpha, Bravo и Charlie объявлены, соответственно, методы showA(), showB() и showC(). Это единственные абстрактные методы в соответствующих интерфейсах, так что интерфейсы являются функциональными. Поэтому интерфейсным переменным данных интерфейсов значением можно присвоить лямбда-выражение. В главном методе программы интерфейсным переменным A и B интерфейсов Alpha и Bravo значением присваивается лямбда-выражение ()->System.out.println("Используем лямбда-выражение"). В результате создаются объекты анонимных классов, ссылки на которые присваиваются переменным. В результате при вызове метода showA() из переменной A в окне вывода отображается текст "Используем лямбда-выражение". Такое же сообщение появляется при вызове метода showB() из переменной B. Но, несмотря на то, что интерфейсным переменным A и B присваиваются одинаковые значения (одинаковые лямбда-выражения), присвоить значением одной переменной другую нельзя. Также нельзя присвоить переменной C интерфейса Charlie ни значение переменной A, ни значение переменной B. Причина в том, что переменные относятся к разным типам, то есть к разным интерфейсам. Здесь есть некоторая «нелогичность», поскольку мы имеем дело с формально разными методами в разных объектах, но код методов одинаков. Поэтому было бы вполне удобно, если бы можно было «воспользоваться» кодом метода в одном объекте для определения кода метода в другом объекте. Такая возможность имеется и состоит она в использовании *ссылки на метод*. Пример использования ссылки на метод есть в программе. Речь о команде Charlie C=A::showA. Командой объявляется интерфейсная переменная C интерфейса Charlie, а значением переменной присваивается ссылка на метод A::showA. Это ссылка на метод showA() из объекта, на который ссылается интерфейсная переменная A. В результате присваивания ссылки A::showA на метод showA() переменной C создается объект анонимного класса, реализующего интерфейс Charlie, в котором метод showC() имеет такой же код, как и метод showA() из объекта, на который ссылается переменная A. Далее мы более подробно обсудим подходы, связанные с использованием ссылок на методы.

НА ЗАМЕТКУ

Работа с лямбда-выражениями неразрывно связана с использованием функциональных интерфейсов. Для последних важным критерием является сигнатура абстрактного метода, объявленного в интерфейсе. Более того, только этот фактор и имеет значение (если речь идет о присваивании интерфейсной переменной в качестве значения лямбда-выражения). В пакете java.util.function содержится

описание многих функциональных интерфейсов (некоторые из них обобщенные), которые соответствуют наиболее часто встречающимся «типам» методов.

Ссылка на метод и конструктор

— Что за вздор. Как вам это в голову взбрело?
— Да не взбрело бы, но факты, как говорится, упрямая вещь.

Из к/ф «Чародеи»

Чтобы понять принципы выполнения и использования ссылок на методы, удобно представлять, что значением выражения со ссылкой на метод является инструкция, аналогичная лямбда-выражению, которое соответствует коду метода (на который выполняется ссылка). Проще говоря, ссылка на метод может рассматриваться как лямбда-выражение, соответствующее коду метода, на который выполняется ссылка. Далее рассмотрим некоторые примеры использования ссылок на методы.

Ссылка на метод объекта

Если речь идет о выполнении ссылки на нестатический метод конкретного объекта, то выполняется она просто (и пример выполнения такой ссылки мы видели): указывается имя объекта, и через оператор :: указывается название метода. То есть шаблон выполнения ссылки на нестатический метод следующий:

```
объект::имя_метода
```

Если ссылка на объект выполнена в формате `объект::метод`, и при этом `метод` описан с некоторыми `аргументами`, то ссылке `объект::метод` соответствует лямбда-выражение вида `аргументы->объект.метод(аргументы)`. Пример использования ссылки на метод объекта представлен в программе в листинге 10.4.

Листинг 10.4. Программный код проекта ObjMethReferenceApplication

```
// Класс:
class MyClass{
  // Закрытое целочисленное поле:
  private int number;
```

```
  // Конструктор:
  MyClass(int n){
    number=n;
  }
  // Метод для присваивания значения полю:
  void set(int n){
    number=n;
  }
  // Метод для считывания значения поля:
  int get(){
    return number;
  }
}
// Первый функциональный интерфейс:
interface MyGetter{
  int myget();
}
// Второй функциональный интерфейс:
interface MySetter{
  void myset(int n);
}
// Главный класс:
class ObjMethReferenceDemo{
  // Главный метод:
  public static void main(String[] args){
    // Создание объекта:
    MyClass obj=new MyClass(100);
    System.out.println("Создан объект с полем 100");
    // Используем ссылки на методы:
    MyGetter A=obj::get;
    MySetter B=obj::set;
    // Проверяем "значение поля" вызовом метода myget()
```

```
// из интерфейсной переменной A:
System.out.println("Переменная A: "+A.myget());
// Проверяем значение поля объекта:
System.out.println("Переменная obj: "+obj.get());
// Полю объекта присваивается значение:
obj.set(200);
System.out.println("Полю присвоено значение 200");
// Проверяем "значение поля" вызовом метода myget()
// из интерфейсной переменной A:
System.out.println("Переменная A: "+A.myget());
// Присваиваем "значение полю" вызовом метода myset()
// из интерфейсной переменной B:
B.myset(300);
System.out.println("Выполнена команда B.myset(300)");
// Проверяем "значение поля" вызовом метода myget()
// из интерфейсной переменной A:
System.out.println("Переменная A: "+A.myget());
// Проверяем значение поля объекта:
System.out.println("Переменная obj: "+obj.get());
// Создается новый объект:
obj=new MyClass(400);
System.out.println("Создан объект с полем 400");
// Проверяем "значение поля" вызовом метода myget()
// из интерфейсной переменной A:
System.out.println("Переменная A: "+A.myget());
// Проверяем значение поля объекта:
System.out.println("Переменная obj: "+obj.get());
// Присваиваем "значение полю" вызовом метода myset()
// из интерфейсной переменной B:
B.myset(500);
System.out.println("Выполнена команда B.myset(500)");
// Проверяем "значение поля" вызовом метода myget()
```

```
    // из интерфейсной переменной A:
    System.out.println("Переменная A: "+A.myget());
    // Проверяем значение поля объекта:
    System.out.println("Переменная obj: "+obj.get());
  }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 10.4)

```
Создан объект с полем 100
Переменная A: 100
Переменная obj: 100
Полю присвоено значение 200
Переменная A: 200
Выполнена команда B.myset(300)
Переменная A: 300
Переменная obj: 300
Создан объект с полем 400
Переменная A: 300
Переменная obj: 400
Выполнена команда B.myset(500)
Переменная A: 500
Переменная obj: 400
```

Мы описываем класс MyClass с закрытым целочисленным полем number, конструктором и двумя метолами: set() для присваивания значения полю и get() для считывания значения поля.

Есть еще два функциональных интерфейса. В интерфейсе MyGetter объявлен метод myget() без аргументов, возвращающий целочисленный результат. В интерфейсе MySetter объявлен метод myset(), у которого целочисленный аргумент и который не возвращает результат. Таким образом, сигнатура метода myget() из интерфейса MyGetter соответствует сигнатуре метода get() из класса MyClass, а сигнатура метода myset() из интерфейса MySetter соответствует сигнатуре метода set() из класса MyClass.

В методе main() из главного класса ObjMethReferenceDemo командой MyClass obj=new MyClass(100) создается объект класса MyClass со значением 100 для поля number. Также командами MyGetter A=obj::get и MySetter B=obj::set объявляются интерфейсные переменные A и B, значениями которым присваиваются ссылки на методы объекта obj: переменной A присваивается ссылка obj::get на метод get(), а переменной B присваивается ссылка obj::set на метод set(). Интрига здесь в том, что и метод get(), и метод set() обращаются к полю number объекта obj. Проанализируем, что происходит. Начнем с переменной A. При присваивании ей в качестве значения ссылки obj::get создается объект анонимного класса, в котором реализуется интерфейс MyGetter. При этом метод myget() определяется как вызов метода get() из объекта obj класса MyClass. Другими словами, при вызове метода myget() из переменной A на самом деле вызывается метод get() из объекта obj. Поэтому командой A.myget() можно узнать значение поля number объекта obj. Нечто похожее происходит при присваивании переменной B ссылки obj::set(), но только теперь речь идет о вызове метода set() из объекта obj. При выполнении команды вида B.myset(число) выполняется команда obj.set(число). Поэтому с помощью вызова метода myset() через переменную B можно присвоить значение полю number объекта obj.

В этом смысле становится понятен смысл происходящего в главном методе. Так, после создания объекта obj выполнение команды A.myget() дает значение 100 для поля number объекта obj — такое же значение возвращается при выполнении команды obj.get().

После выполнения команды obj.set(200) поле number объекта obj получает значение 200. Проверка значения поля number с помощью команды A.myget() подтверждает это. Новое значение 300 полю number присваивается командой B.myset(300). Проверка значения поля с помощью команд A.myget() и obj.get() дает такое же значение.

Ситуация меняется, когда командой obj=new MyClass(400) создается новый объект (со значением 400 для поля number) и ссылка на него записывается в переменную obj. Но при этом в определении методов myget() и myset() ссылка на объект остается неизменной: оба эти метода обращаются к полю number объекта, на который ранее ссылалась объектная переменная obj.

НА ЗАМЕТКУ

Нелишним будет напомнить, что в Java объектная переменная ссылается на объект: фактически значением объектной переменной является адрес объекта.

Поэтому после того, как ссылка в объектной переменной obj «переброшена» на новый объект, изменение поля number через переменную obj никак не влияет на значение поля number, с которым «оперируют» методы myget() и myset(). Верно и обратное замечание: вызов метода myset() из переменной B изменяет результат, возвращаемый при вызове метода myget() из переменной A, но не изменяет значение поля number объекта, на который теперь ссылается объектная переменная obj.

Ссылка на нестатический метод класса

Ссылка может выполняться не только на метод объекта, но и на метод класса. В последнем случае имеет значение, является ли метод статическим, или нет. Сначала рассмотрим случай, когда метод нестатический.

Ссылка на нестатический метод некоторого класса выполняется в следующем виде:

```
класс::имя_метода
```

Синтаксис в общем-то простой. Немного сложнее понять, что он означает. Правило такое. Если используется ссылка вида класс::метод, и при этом метод нестатический и описан с некоторыми аргументами, то указанная ссылка эквивалента лямбда-выражению следующего вида: (объект,аргументы)->объект.метод(аргумнты). Более конкретно: допустим, что имеется класс MyClass, и в нем описан некоторый нестатический метод method() с аргументом num типа int и аргументом symb типа char, то ссылке на метод MyClass::method соответствует лямбда-выражение вида (MyClass obj,int num,char symb)->{код_метода_method()}. Поэтому если мы хотим ссылку MyClass::method присвоить интерфейсной переменной, то в этом функциональном интерфейсе абстрактный метод должен быть объявлен с тремя аргументами соответственно типа MyClass, int и char (и тип результата абстрактного метода должен согласовываться с типом результата, возвращаемого методом method()).

Чтобы не быть голословными, рассмотрим пример. В листинге 10.5 представлена модификация предыдущего пример (см. листинг 10.4), но на этот раз вместо ссылки на метод объекта выполняется ссылка на нестатический метод класса.

Листинг 10.5. Программный код проекта MethReferenceApplication

```
// Класс с полем и методами:
class MyClass{
```

```
  private int number;
  MyClass(int n){
    number=n;
  }
  void set(int n){
    number=n;
  }
  int get(){
    return number;
  }
}
// Первый функциональный интерфейс:
interface MyGetter{
  int myget(MyClass obj);
}
// Второй функциональный интерфейс:
interface MySetter{
  void myset(MyClass obj,int n);
}
// Главный класс:
class MethReferenceDemo{
  public static void main(String[] args){
    // Создается объект:
    MyClass obj=new MyClass(100);
    System.out.println("Создан объект с полем 100");
    // Используем ссылки на методы:
    MyGetter A=MyClass::get;
    MySetter B=MyClass::set;
    System.out.println("Переменная A: "+A.myget(obj));
    System.out.println("Переменная obj: "+obj.get());
    obj.set(200);
    System.out.println("Полю присвоено значение 200");
    System.out.println("Переменная A: "+A.myget(obj));
```

```
    B.myset(obj,300);
    System.out.println("Выполнена команда B.myset(obj,300)");
    System.out.println("Переменная A: "+A.myget(obj));
    System.out.println("Переменная obj: "+obj.get());
    // Создается новый объект:
    obj=new MyClass(400);
    System.out.println("Создан объект с полем 400");
    System.out.println("Переменная A: "+A.myget(obj));
    System.out.println("Переменная obj: "+obj.get());
    B.myset(obj,500);
    System.out.println("Выполнена команда B.myset(obj,500)");
    System.out.println("Переменная A: "+A.myget(obj));
    System.out.println("Переменная obj: "+obj.get());
  }
}
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 10.5)

```
Создан объект с полем 100
Переменная A: 100
Переменная obj: 100
Полю присвоено значение 200
Переменная A: 200
Выполнена команда B.myset(obj,300)
Переменная A: 300
Переменная obj: 300
Создан объект с полем 400
Переменная A: 400
Переменная obj: 400
Выполнена команда B.myset(obj,500)
Переменная A: 500
Переменная obj: 500
```

Результат выполнения программы немного изменился, если сравнивать с предыдущим примером. Но многие блоки кода остались неизменными. Так, совершенно не изменилось описание класса MyClass. Зато изменилось объявление методов myget() из интерфейса MyGetter и myset() из интерфейса MySetter. Дело в том, что в главном методе программы, как и ранее, создается объект obj класса MyClass, но ссылку на методы get() и set() из класса MyClass мы выполняем не через объект, а через класс, соответственно, в формате MyClass::get и MyClass::set. Что означает, например, ссылка MyClass::get? Метод get() описан без аргументов и это нестатический метод, поэтому должен вызываться из объекта. С учетом реального кода метода get(), ссылка MyClass::get соответствует лямбда-выражению вида (MyClass obj)->{obj.get()}. Аналогично, метод set() описан с одним целочисленным аргументом. Поэтому ссылке MyClass::set соответствует лямбда-выражение вида (MyClass obj,int n)->{obj.set(n)}. В соответствии с новым «форматом» ссылок несколько иначе объявляются и методы myget() и myset() в функциональных интерфейсах MyGetter и MySetter: в общем и целом, у этих методов появляется дополнительный (первый) аргумент, который является объектом класса MyClass. Этот аргумент определяет объект, из которого вызывается соответственно метод get() и set().

В главном методе программы после выполнения команд MyGetter A=MyClass::get и MySetter B=MyClass::set для вызова методов myget() и myset() используем команды вида A.myget(obj)) или B.myset(obj,число). Командой A.myget(obj)) на самом деле из объекта obj вызывается метод get(), а, например, командой B.myset(obj,300) из объекта obj вызывается метод set() с аргументом 300. Здесь, собственно, и кроются истоки к пониманию того, почему у программы именно тот результат выполнения, который приведен выше.

НА ЗАМЕТКУ

Читателям рекомендуется подумать над тем, почему же все-таки в некоторых момента результат выполнения программы отличается от результата выполнения программы из предыдущего примера.

Ссылка на статический метод

Ссылка на статический метод класса формально выполняется так же, как и на нестатический метод класса, но последствия, так сказать, другие. В частности, если используется ссылка на метод вида класс::метод, и при этом метод является статическим и описан с некоторыми аргументами, то указанная выше ссылка эквивалентна лямбда-выражению вида

(аргументы)->класс.метод(аргументы). Например, ссылка System.out::println эквивалентна лямбда-выражению t->System.out.println(t). Соответствующим образом данная ссылка и должна использоваться. Небольшой пример по этому поводу приведен в листинге 10.6.

Листинг 10.6. Программный код проекта StatMethReferenceApplication

```
// Класс со статическими методами:
class MyClass{
  // Методом отображается сообщение:
  static void show(){
    System.out.println("Метод класса MyClass");
  }
  // Методом вычисляется сумма чисел:
  static int sum(int n){
    int k,s=0;
    for(k=1;k<=n;k++){
      s+=k;
    }
    return s;
  }
}
// Первый интерфейс:
interface MyShow{
  void myshow();
}
// Второй интерфейс:
interface MySum{
  int mysum(int n);
}
// Третий интерфейс:
interface MyPrinter{
  void myprint(Object t);
}
```

```
// Главный класс:
class StatMethReferenceDemo{
  // Главный метод:
  public static void main(String[] args){
    // Использование ссылок на статические методы:
    MyShow A=MyClass::show;
    MySum B=MyClass::sum;
    MyPrinter P=System.out::println;
    // Вызов методов из интерфейсных переменных:
    A.myshow();
    P.myprint("Сумма чисел: "+B.mysum(10));
  }
}
```

При выполнении программы получаем такой результат:

Результат выполнения программы (из листинга 10.6)

```
Метод класса MyClass
Сумма чисел: 55
```

Мы описываем класс MyClass с двумя статическими методами. Метод show() не имеет аргументов и не возвращает результат. При вызове метода отображается сообщение с названием класса MyClass. Методом sum() вычисляется сумма натуральных чисел, которая возвращается результатом метода. Аргументом методу передается количество слагаемых в сумме.

В программе есть три функциональных интерфейса. Каждый из интерфейсов содержит объявление абстрактного метода, сигнатура которых соответствует двум статическим методам из класса MyClass, и методу println(). В последнем случае речь идет об интерфейсе MyPrinter, в котором объявлен не возвращающий результат метод myprint() с аргументом типа Object. Поскольку класс Object является суперклассом для всех классов, то в итоге аргументом метода myprint() может быть значение практически любого типа.

В главном методе программы есть примеры присваивания интерфейсным переменным в качестве значений ссылок на статические методы. Речь о командах MyShow A=MyClass::show, MySum B=MyClass::sum и MyPrinter P=System.

out::println. В результате при выполнении команды A.myshow() на самом деле из класса MyClass вызывается метод show(). При выполнении команды B.mysum(10) из класса MyClass вызывается статический метод sum() с аргументом 10, а вызов метода myprint() из переменной P с аргументом "Сумма чисел: "+B.mysum(10) аналогичен выполнению команды System.out.println("Сумма чисел: "+B.mysum(10)).

Ссылка на конструктор

Ссылку можно выполнять не только на метод, но и на конструктор. Синтаксис выполнения ссылки на конструктор такой: после имени класса указывается оператор ::, после которого следует ключевое слово new. Вся конструкция выглядит следующим образом:

```
класс::new
```

Если использована ссылка на конструктор вида класс::new, то ее эквивалентом является лямбда-выражение вида (аргументы)->new класс(аргументы), где подразумевается, что при создании объекта класса конструктору передаются аргументы.

НА ЗАМЕТКУ

Фактически речь идет не просто о конструкторе, а о команде создания нового объекта с помощью конструктора. Результатом выражения вида new класс(аргументы), очевидно, является ссылка на созданный объект — то есть значение типа класс. Такой результат должен быть у абстрактного метода из функционального интерфейса, переменной которого значением присваивается ссылка на конструктор.

Совсем небольшой пример использования ссылки на конструктор приведен в программе в листинге 10.7.

Листинг 10.7. Программный код проекта ConstructorReferenceApplication

```
// Класс:
class MyClass{
  // Закрытое поле:
  private int number;
  // Конструктор:
```

```
  MyClass(int n){
    number=n;
  }
  // Открытые методы:
  void show(){
    System.out.println("Значение поля: "+number);
  }
  void set(int n){
    number=n;
  }
}
// Интерфейс:
interface MyInterface{
  MyClass create(int n);
}
// Главный класс:
class ConstructorReferenceDemo{
  // Главный метод:
  public static void main(String[] args){
    // Использование ссылки на конструктор:
    MyInterface ref=MyClass::new;
    // Создание объекта вызовом метода
    // из интерфейсной переменной:
    MyClass obj=ref.create(100);
    // Вызов методов объекта:
    obj.show();
    obj.set(200);
    obj.show();
  }
}
```

Результат выполнения программы представлен ниже.

Результат выполнения программы (из листинга 10.7)

```
Значение поля: 100
Значение поля: 200
```

Код совсем небольшой. Проанализируем его. В программе описан класс MyClass, у которого есть закрытое целочисленное поле, метод set(), предназначенный для присваивания значения полю, метод show() для отображения значения поля и конструктор с одним целочисленным аргументом. В главном методе программы командой MyInterface ref=MyClass::new переменной ref интерфейсного типа MyInterface значением присваивается ссылка MyClass::new на конструктор класса MyClass. Интерфейс MyInterface является функциональным. В нем объявлен только один абстрактный метод create(), у которого единственный целочисленный аргумент. Результатом метод возвращает значение типа MyClass. Поэтому значением интерфейсной переменной типа MyInterface может быть присвоена ссылка на метод с целочисленным аргументом, возвращающий значение типа MyClass. Этим критериям удовлетворяет выражение вида new MyClass(число), результатом которого является объектная ссылка типа MyClass. Таким образом, после выполнения команды MyInterface ref=MyClass::new вызов из переменной ref метода create() с целочисленным аргументом эквивалентен вызову (через оператор new) конструктора класса MyClass с таким же аргументом. Более конкретно, при выполнении команды MyClass obj=ref.create(100) создается объект класса MyClass со значением 100 для поля number, а ссылка на этот объект записывается в объектную переменную obj. В дальнейшем через эту переменную можно обращаться к объекту так, как если бы он было создан с помощью выражения new MyClass(100).

Ссылка на перегруженный метод

Вполне может сложиться ситуация, когда необходимо выполнить ссылку на метод, который перегружен. В таком случае в классе описано несколько версий метода или, если быть более точными, то несколько методов с одинаковыми названиями. Собственно по ссылке на метод нельзя однозначно определить, о какой версии метода идет речь. Вывод о том, какую версию метода следует использовать, делается на основе интерфейсной переменной, которой в качестве значения присваивается ссылка на метод. Как иллюстрацию к сказанному рассмотрим небольшой пример, в котором выполняется ссылка на перегруженный метод объекта. Рассмотрим программный код в листинге 10.8.

Листинг 10.8. Программный код проекта OverloadedMethRefApplication

```
// Класс:
class MyClass{
  // Поле:
  int number;
  // Перегруженный метод:
  void set(){
    number=0;
  }
  void set(int n){
    number=n;
  }
}
// Первый интерфейс:
interface Alpha{
  void none();
}
// Второй интерфейс:
interface Bravo{
  void one(int n);
}
// Главный класс:
class OverloadedMethRefDemo{
  // Главный метод:
  public static void main(String[] args){
    // Создание объекта:
    MyClass obj=new MyClass();
    // Использование ссылки на перегруженный метод:
    Alpha A=obj::set;
    Bravo B=obj::set;
    // Вызов метода через интерфейсную переменную:
    B.one(100);
    // Проверка значения поля объекта:
    System.out.println("Значение поля: "+obj.number);
```

```
    // Вызов метода через интерфейсную переменную:
    A.none();
    // Проверка значения поля объекта:
    System.out.println("Значение поля: "+obj.number);
  }
}
```

В результате выполнения программы получим следующее:

Результат выполнения программы (из листинга 10.8)

```
Значение поля: 100
Значение поля: 0
```

Класс MyClass имеет одно поле и две версии метода set() (без аргументов и с одним целочисленным аргументом). В функциональном интерфейсе Alpha объявлен метод none() без аргументов, а в функциональном интерфейсе Bravo описан метод one() с одним целочисленным аргументом. Оба метода, как и обе версии метода set(), не возвращают результат.

В главном методе программы сначала создается объект obj класса MyClass (команда MyClass obj=new MyClass()), а затем командами Alpha A=obj::set и Bravo B=obj::set одна и та же ссылка obj::set присваивается переменным разного типа. Поскольку в интерфейсе Alpha абстрактный метод описан без аргументов, то при выполнении команды Alpha A=obj::set используется версия метода set() без аргументов. А вот в интерфейсе Bravo абстрактный метод описан с целочисленным аргументом, поэтому в команде Bravo B=obj::set задействована версия метода set() с целочисленным аргументом. Как следствие, при выполнении команды B.one(100) из объекта obj вызывается метод set() с аргументом 100, а при выполнении команды A.none() из объекта obj вызывается метод set() без аргументов.

Использование лямбда-выражений

Это экспонаты. Отходы, так сказать, магического производства.

Из к/ф «Чародеи»

Выше мы познакомились с основными, наиболее элементарными операциями, выполняемыми с лямбда-выражениями и ссылками на методы.

Далее рассмотрим несколько конкретных ситуаций, когда лямбда-выражения используются при решении прикладных задач.

Передача лямбда-выражения аргументом методу

Если говорить о программировании вообще, то целый класс практически важных задач подразумевает передачу функции аргументом другой функции. В языке Java функции в «чистом виде» не встречаются. Вместо них используются методы. Поэтому в контексте языка программирования Java будем говорить о передаче метода аргументом методу. Задач, когда функции аргументом передается функция (методу аргументом передается метод) достаточно много. Мы рассмотрим одну, но «классическую» — *вычисление интеграла*.

ДЕТАЛИ

Не вдаваясь в несущественные детали, будем под интегралом $\int_a^b f(x)\,dx$ подразумевать площадь под кривой $f(x)$, при условии, что значения аргумента x попадают в диапазон $a \le x \le b$. Для вычисления интеграла диапазон интегрирования $a \le x \le b$ разбивается на n внутренних интервалов равной длины $h = (b - a)/n$, границы этих внутренних интервалов определяются узловыми точками $x_k = a + kh$, где индекс $k = 0, 1, 2, \ldots, n$ (и, таким образом, $x_0 \equiv a$ и $x_n \equiv b$). Вычисляемый интеграл представляется в виде сумма интегралов $\int_a^b f(x)dx = \sum_{k=0}^{n-1} \int_{x_k}^{x_{k+1}} f(x)dx$. Для каждого из интегралов в сумме используется приближенное выражение $\int_{x_k}^{x_{k+1}} f(x)dx \approx ((f(x_k) + f(x_{k+1}))/2)h$. В итоге получаем формулу $\int_a^b f(x)dx \approx ((f(a) + f(b))/2)h + h\sum_{k=1}^{n-1} f(x_k)$, которая называется *формулой трапеций*. Именно этой формулой мы воспользуемся для вычисления интегралов от разных функций.

Для вычисления интеграла необходимо задать:

- функцию $f(x)$, которая находится под знаком интеграла и называется подынтегральной функцией;
- границы a и b области интегрирования.

Решать задачу будем с помощью специального статического метода, аргументами которому передаются лямбда-выражение (определяет подынтегральную функцию $f(x)$) и два числовых значения (границы области интегрирования a и b). Результатом метод возвращает значение (приближенное) для интеграла. При вычислении результата используется

формула $\int_a^b f(x)dx \approx ((f(a)+f(b))/2)h + h\sum_{k=1}^{n-1} f(x_k) = ((f(a)+f(b))/2)h + h\sum_{k=1}^{n-1} f(a+kh)$. При этом параметр h вычисляется как $h = (b-a)/n$, а целочисленное значение n определяется через локальную переменную. Далее имеет смысл рассмотреть программный код в листинге 10.9.

Листинг 10.9. Программный код проекта IntegralCalcApplication

```
// Функциональный интерфейс:
interface MyFunction{
  // Метод с double-аргументом и double-результатом:
  double f(double x);
}
// Главный класс:
class IntegralCalcDemo{
  // Статический метод для вычисления интеграла:
  static double integrate(MyFunction obj,double a,double b){
    // Количество внутренних интервалов:
    int n=1000;
    // Длина внутреннего интервала:
    double h=(b-a)/n;
    // Переменная для записи интегральной суммы:
    double s=(obj.f(a)+obj.f(b))*h/2;
    // Вычисление интегральной суммы:
    for(int k=1;k<=n-1;k++){
      s+=h*obj.f(a+k*h);
    }
    // Результат метода — значение интеграла:
    return s;
  }
  // Главный метод:
  public static void main(String[] args){
    // Вычисление интеграла:
    System.out.print(integrate((double x)->{return x*(1-x);},0,1));
    // Значение для сравнения:
```

```
    System.out.println(" — точное значение "+1.0/6);
    // Вычисление интеграла:
    System.out.print(integrate((double x)->{return 1/x;},1,2));
    // Значение для сравнения:
    System.out.println(" — точное значение "+Math.log(2));
    // Вычисление интеграла:
    System.out.print(integrate((double x)->{return Math.exp(-x);},0,10));
    // Значение для сравнения:
    System.out.println(" — точное значение "+(1-Math.exp(-10)));
  }
}
```

Результат выполнения программы такой, как показано ниже:

Результат выполнения программы (из листинга 10.9)

```
0.16666650000000022 — точное значение 0.16666666666666666
0.6931472430599375 — точное значение 0.6931471805599453
0.9999629330113494 — точное значение 0.9999546000702375
```

У подынтегральной функции *f(x)*, какой бы она ни была, один числовой аргумент и результатом возвращает числовое значение. Поэтому для работы с лямбда-выражениями, через которые мы собираемся задавать подынтегральную функцию, описываем интерфейс MyFunction с объявленным в нем методе f(), у которого double-результат и которого есть аргумент типа double.

Для вычисления интегралов предназначен статический метод integrate(). Первый аргумент метода — интерфейсная переменная типа MyFunction. Мы предполагаем, что переменная содержит ссылку на объект, созданный на основе класса, реализующего интерфейс MyFunction. Метод f() в этом объекте определяет подынтегральную функцию. Второй и третий аргументы метода integrate() являются числовыми значениями типа double, определяющими границы диапазона интегрирования. Результатом метод integrate() возвращает числовое значение типа double. Это и есть значение интеграла.

Код метода integrate() достаточно простой: в теле метода объявляется локальная целочисленная переменная n со значением 1000 (количество

интервалов, на которые разбивается диапазон интегрирования), а еще одна числовая переменная h (длина внутреннего интервала интегрирования) типа double значением получает результат выражения (b-a)/n. Переменная s типа double, в которую записывается значение подынтегральной суммы, вначале имеет значение (obj.f(a)+obj.f(b))*h/2 (что соответствует слагаемому $((f(a) + f(b))/2)h$ в формуле $\int_a^b f(x)dx \approx ((f(a) + f(b))/2)h + h\sum_{k-1}^{n-1} f(a + kh)$). Затем запускается оператор цикла, в котором за каждый цикл к текущему значению переменной s прибавляется слагаемое h*obj.f(a+k*h) (соответствующее слагаемому $hf(a + kh)$ в формуле для вычисления интеграла). После завершения оператора цикла значение переменной s возвращается результатом метода integrate().

В главном методе программы с помощью метода integrate() вычисляется несколько интегралов. При этом первым аргументом методу integrate() передается лямбда-выражение. Например, с помощью инструкции integrate((double x)->{return x*(1-x);},0,1)) вычисляется интеграл $\int_0^{-1} x(1 - x)dx = 1/6$ (в программе для сравнения вычисленного значения приводится «точное» значение для интеграла). Подынтегральную функцию $f(x) = x(1 - x)$ мы задаем с помощью лямбда-выражения (double x)->{return x*(1-x);}. Кроме этого интеграла, в программе вычисляется интеграл $\int_1^2 dx/x = \ln(2)$ (подынтегральная функция $f(x) = 1/x$ задается лямбда-выражением (double x)->{return 1/x;}, а натуральный логарифм при проверке результата вычисляется с помощью статического метода log() класса Math), а также интеграл $\int_0^{10} \exp(-x) = 1 - \exp(-10)$ (подынтегральная функция $f(x) = \exp(-x)$ задается лямбда-выражением (double x)->{return Math.exp(-x);}, а экспонента вычисляется с помощью статического метода exp() класса Math).

Теперь проанализируем причины, по которым первым аргументом методу integrate() можно передавать лямбда-выражения, хотя в описании метода первым аргументом обозначена интерфейсная переменная типа MyFunction. Происходит следующее. Когда аргументом методу передается лямбда-выражение (формат выражения должен соответствовать функциональному интерфейсу MyFunction), то технической переменной (интерфейсная переменная типа MyFunction), предназначенной для запоминания значения аргумента метода, присваивается значением лямбда-выражение. В результате создается объект анонимного класса, реализующего интерфейс MyFunction, метод f() определяется в соответствии с лямбда-выражением, а ссылка на этот объект записывается в техническую переменную, «выделенную» для записи значения аргумента. В этом смысле то, что происходит, полностью укладываются в схему выполнения программного кода метода integrate(). Другими словами, в описании метода

предполагалось, что первый аргумент является ссылкой на объект. В реальности так и происходит. Более того, мы в принципе можем передавать первым аргументом методу явную ссылку на объект некоторого класса, который реализует интерфейс MyFunction. Преимущество в использовании лямбда-выражений состоит в том, что нет необходимости явно описывать класс и создавать на его основе объект. При передаче аргументом лямбда-выражения приходится определять только программный код, задающий подынтегральную функцию.

Лямбда-выражение и результат метода

Лямбда-выражение может использоваться при определении результата метода. Формальным результатом в таком случае является значение интерфейсного типа, но собственно в коде метода результат может быть «оформлен» в виде лямбда-выражения. Мы именно такой случай и рассмотрим. Как «учебную» рассмотрим задачу о вычислении производной для функции.

ДЕТАЛИ

Производной $f'(x)$ для функции $f(x)$ называется предел отношения приращения функции к приращению аргумента при стремлении последнего к нулю: $f'(x) = \lim_{\Delta x \to 0}(f(x + \Delta x) - f(x))/\Delta x$. Для нас в данном случае важно то обстоятельство, что при вычислении производной на основе одной исходной функции (дифференцируемая функция $f(x)$) по определенным правилам получают другую функцию (производная функция $f'(x)$).

При вычислении производной в числовом виде нередко используют приближенное выражение $f'(x) \approx (f(x + \Delta x) - f(x))/\Delta x$, в котором параметр Δx, обозначающий приращение аргумента, мал (в принципе, чем меньше — тем лучше), но отличен от нуля. Именно такой подходы мы используем далее для вычисления на основе дифференцируемой функции $f(x)$ ее производной $f'(x)$.

Таким образом, мы решаем задачу о «вычислении» на основе одной (дифференцируемой) функции другой функции (производной).

НА ЗАМЕТКУ

Речь не идет о вычислении значения функции в точке. Мы решаем задачу совершенно иного рода. Нам необходимо определить функцию

как таковую, то есть определить функциональную зависимость. Другими словами, «на входе» есть одна функция, а «на выходе» должна быть другая.

Задачу решаем с помощью статического метода. Аргументом методу будет передаваться лямбда-выражение, определяющее дифференцируемую функцию, а результатом будет возвращаться лямбда-выражение, определяющее производную.

Технически для реализации означенного подхода мы описываем функциональный интерфейс MyFunction, в котором объявлен метод f(). У метода один double-аргумент и метод возвращает такого же типа результат. Через переменные этого интерфейса мы планируем «оперировать» с лямбда-выражениями.

В главном классе DerivativeCalcDemo описан статический метод Derivative(), предназначенный для вычисления производной. Аргументом методу передается значение типа MyFunction (аргумент обозначен как ref), и результатом возвращается значение такого же типа.

Мы предполагаем, что метод f() объекта, на который ссылается аргумент ref, определяет исходную функциональную зависимость (которую следует дифференцировать). В теле метода объявляется локальная переменная dx со значением 1e-5 (литерал для обозначения числа 10^{-5}). Переменная определяет приращение по аргументу. Данный параметр, как отмечалось, используется при вычислении производной. Результатом метод возвращает лямбда-выражение (double x)->{return (ref.f(x+dx)-ref.f(x))/dx;}. Несложно догадаться, что этим выражением определяется метод, который соответствует производной функции.

ДЕТАЛИ

Напомним, что результат метода обозначен как значение интерфейсного типа MyFunction. При этом фактически в return-инструкции указано лямбда-выражение. Ситуация примерно такая же, как и при передаче аргументом лямбда-выражения. В частности, для записи результат метода в памяти выделяется место под интерфейсную переменную типа MyFunction. Этой технической переменной значением присваивается лямбда-выражение (указанное в return-инструкции). В итоге такой операции создается объект анонимного класса, реализующего интерфейс MyFunction, и ссылка на объект записывается в переменную, выделенную под результат. Именно эта ссылка на объект и есть результат метода.

В главном методе программы показано, как метод Derivative() может использоваться для вычисления производной от функции. Весь программный код представлен в листинге 10.10.

Листинг 10.10. Программный код проекта DerivativeCalcApplication

```
// Функциональный интерфейс:
interface MyFunction{
  // Метод с аргументом типа double возвращает
  // результатом значение типа double:
  double f(double x);
}
// Главный класс:
class DerivativeCalcDemo{
  // Статический метод для вычисления производной:
  static MyFunction Derivative(MyFunction ref){
    // Приращение по аргументу для вычисления
    // производной в точке:
    double dx=1e-5;
    // Результат метода — лямбда-выражения:
    return (double x)->{return (ref.f(x+dx)-ref.f(x))/dx;};
  }
  // Главный метод:
  public static void main(String[] args){
    // Производная для первой функции:
    MyFunction A=Derivative((double x)->{return x*(3-x);});
    // Производная для второй функции:
    MyFunction B=Derivative((double x)->{return x*Math.exp(-x);});
    // Проверка результатов вычислений:
    System.out.println("Производная для первой функции");
    System.out.println("Вычислено:\tТочно:");
    for(double t=0;t<=5;t++){
      System.out.printf("%8.5f\t%8.5f\n",A.f(t),(3-2*t));
    }
```

```
    System.out.println("Производная для второй функции");
    System.out.println("Вычислено:\tТочно:");
    for(double t=0;t<=5;t++){
      System.out.printf("%8.5f\t%8.5f\n",B.f(t),Math.exp(-t)*(1-t));
    }
  }
}
```

Результат выполнения программы показан ниже:

Результат выполнения программы (из листинга 10.10)

```
Производная для первой функции
Вычислено:  Точно:
 2,99999    3,00000
 0,99999    1,00000
-1,00001   -1,00000
-3,00001   -3,00000
-5,00001   -5,00000
-7,00001   -7,00000
Производная для второй функции
Вычислено:  Точно:
 0,99999    1,00000
-0,00000    0,00000
-0,13534   -0,13534
-0,09957   -0,09957
-0,05495   -0,05495
-0,02695   -0,02695
```

В методе main() командой MyFunction A=Derivative((double x)->{return x*(3-x);}) объявляется интерфейсная переменная A, значение которой вычисляется вызовом методе Derivative(). Аргументом методу передается лямбда-выражение, которое означает, что «дифференцируется» функция $f(x) = x(3 - x)$. Результатом возвращается ссылка на объект класса, реализующего интерфейс MyFunction, и в этом объекте метод f() определяет производную $f'(x)$

(точное значение для производной $f'(x) = 3 - 2x$). Аналогично, командой MyFunction B=Derivative((double x)->{return x*Math.exp(-x);}) определена переменная B, ссылающаяся на объект, метод f() которого соответствует производной от функции $f(x) = x \cdot \exp(-x)$ (точное значение для производной $f'(x) = \exp(-x)(1 - x)$). Далее запускаются операторы цикла, с помощью которых значения для производных вычисляются в нескольких точках (для нескольких значений аргумента). Для сравнения приводятся «точные» значения для производных в этих же точках.

НА ЗАМЕТКУ

При отображении числовых значений использован метод printf(). Например, команда System.out.printf("%8.5f\t%8.5f\n",A.f(t),(3-2*t)) означает, что отображается значение A.f(t), затем выполняется табуляция (инструкция \t), отображается значение (3-2*t), и выполняется переход к новой строке (инструкция \n). Инструкция формата %8.5f означает, что отображается действительное число (символ f), в котором в дробной части должно быть не меньше 5 цифр, а всего под число выделяется не менее 8 позиций (включая одну позицию под десятичную точку/запятую).

Лямбда-выражение и поле объекта

Лямбда-выражения можно успешно использовать при работе с полями объектов. Небольшой пример с иллюстрацией такой ситуации приведен в листинге 10.11.

Листинг 10.11. Программный код проекта LambdaAsFieldApplication

```
// Функциональный интерфейс:
interface MyInterface{
  // Метод с целочисленным аргументом возвращает
  // целочисленный результат:
  int getNumber(int n);
}
// Класс с полем интерфейсного типа:
class MyClass{
  // Закрытое поле интерфейсного типа:
  private MyInterface ref;
```

```
  // Конструктор:
  MyClass(MyInterface mi){
    set(mi);
  }
  // Метод для присваивания значения полю:
  void set(MyInterface mi){
    ref=mi;
  }
  // Метод с целочисленным аргументом возвращает
  // результатом целое число:
  int get(int n){
    // Вызов метода из объекта, на которое ссылается
    // поле интерфейсного типа:
    return ref.getNumber(n);
  }
}
// Главный класс:
class LambdaAsFieldDemo{
  // Главный метод:
  public static void main(String[] args){
    // Создание объекта класса с передачей аргументом
    // конструктору лямбда-выражения:
    MyClass obj=new MyClass((int n)->{return n*n;});
    // Проверка результата:
    System.out.println("Аргумент:");
    for(int k=0;k<=5;k++){
      System.out.print(k+"\t");
    }
    System.out.println("\nАргумент в квадрате:");
    for(int k=0;k<=5;k++){
      System.out.print(obj.get(k)+"\t");
    }
```

```
    // Полю объекта присваивается новое значение:
    obj.set((int n)->{return n*n*n;});
    // Проверка результата:
    System.out.println("\nАргумент в кубе:");
    for(int k=0;k<=5;k++){
      System.out.print(obj.get(k)+"\t");
    }
    System.out.println("");
  }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 10.11)

```
Аргумент:
0   1   2   3   4   5
Аргумент в квадрате:
0   1   4   9   16  25
Аргумент в кубе:
0   1   8   27  64  125
```

В программе описан функциональный интерфейс MyInterface, в котором объявлен абстрактный метод getNumber(), возвращающий результатом целое число. У метода должен быть один целочисленный аргумент.

Еще в программе описан класс MyClass. У класса есть закрытое поле ref интерфейсного типа MyInterface. Описанный в классе открытый метод get() возвращает результатом целое число: при вызове метода с целочисленным аргументом в теле метода с таким же аргументом через поле ref вызывается метод getNumber(). Метод set() предназначен для присваивания значения полю ref. Аргументом методу set() передается значение интерфейсного типа MyInterface. Метод set() также вызывается в теле конструктора.

В главном методе командой MyClass obj=new MyClass((int n)->{return n*n;}) создается объект obj класса MyClass. Аргументом конструктору передано лямбда-выражение. В результате поле ref объекта obj ссылается на объект,

в котором метод getNumber() по аргументу n возвращает квадрат аргумента (значение n*n). Но поскольку метод getNumber() вызывается не напрямую, а через метод get() объекта obj, то «иллюзия» такая, как если бы это метод get() был определен для возведения аргумента в квадрат.

После выполнения команды obj.set((int n)->{return n*n*n;}) технически происходит следующее. Создается новый объект класса, реализующего интерфейс MyInterface, в котором метод getNumber() вычисляет третью степень аргумента. Ссылка на этот объект записывается в поле ref объекта obj. Но внешний эффект такой, как будто мы «переопределили» метод get() объекта obj.

Резюме

Увы мне, кудесник. Отправляй меня назад!
Из к/ф «Иван Васильевич меняет профессию»

- Лямбда-выражение определяет некое подобие анонимного метода. Описание лямбда-выражения состоит из списка аргументов в круглых скобках, стрелки -> и тела выражения в фигурных скобках.
- Лямбда-выражение может присваиваться значением интерфейсной переменной при условии, что соответствующий интерфейс является функциональным — то есть в нем объявлен только один абстрактный метод (могут быть и другие методы, но с кодом по умолчанию).
- Если лямбда-выражение присваивается значением интерфейсной переменной, то создается объект анонимного класса, реализующего данный функциональный интерфейс, в котором абстрактный метод определен в соответствии с кодом лямбда-выражения. Ссылка на созданный метод записывается в интерфейсную переменную.
- Можно выполнять ссылки на нестатические методы объекта (в формате объект::метод), нестатические и статические методы класса (в формате класс::метод), а также конструкторы (в формате класс::new). Результатом ссылки является лямбда-выражение.
- Лямбда-выражения, кроме прочего, могут передаваться аргументом методу, использоваться при определении результата метода или значения поля объекта.

Глава 11
ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

— Простите, часовню тоже я развалил?
— Нет, это было до вас, в XIV веке.

Из к/ф «Кавказская пленница»

Далее мы обсудим *обработку исключительных ситуаций* — полезный и эффективный механизм, позволяющий создавать гибкие и надежные в плане (возникновения ошибок) программные коды. Под исключительными ситуациями будем подразумевать ошибки, возникающие при выполнении программы (такие ошибки называются *ошибками времени выполнения программы*). Если не предпринимать никаких дополнительных действий, то возникновение ошибки при выполнении программы естественным и не очень приятным образом приводит к досрочному прекращению выполнения программы. Механизм обработки исключений ориентирован в первую очередь на то, чтобы «научить» программу «правильно реагировать» в случае возникновения ошибки.

Перехват и обработка ошибок

— А почему он роет на дороге?
— Да потому, что в других местах все уже перерыто и пересеяно.

Из к/ф «31 июня»

Основу системы обработки исключительных ситуаций составляет иерархия классов, описывающих исключительные ситуации, и оператор try-catch. Изучение вопроса начнем с простого примера, на основе которого поясним некоторые важные «теоретические» моменты.

Пример обработки исключения

Ранее мы многократно сталкивались с ситуацией, когда по запросу программы в диалоговом окне с текстовым полем вводится числовое

значение, которое считывается в текстовом формате, после чего преобразуется в число. Напомним, что там возможны некоторые «неприятные» ситуации, вроде того, что пользователь передумал вводить значение или ввел значение, которое не является числом. В первом случае легко проверить, содержит ли переменная, в которую записывается результат считывания введенного значения, непустую ссылку. Во втором случае «неприятность» в виде некорректного значения отслеживать сложнее. Решением проблемы может стать применение механизма обработки исключительных ситуаций. Рассмотрим программный код, представленный в листинге 11.1. Назначение у представленной там программы простое: отображается окно с полем ввода, в которое пользователя просят ввести целое число. Число считывается и в новом диалоговом окне отображается три числа: то, которое ввел пользователь, и еще два соседних числа.

Листинг 11.1. Программный код проекта TryCatchExampleApplication

```
// Статический импорт:
import static javax.swing.JOptionPane.*;
import static java.lang.Integer.*;
// Главный класс:
class TryCatchExampleDemo{
  // Главный метод:
  public static void main(String[] args){
    // Текстовая переменная для записи
    // считываемого значения:
    String input;
    // Переменная для записи целого числа:
    int num;
    // Отображение диалогового кона с полем ввода:
    input=showInputDialog(null,
      "Введите число", // Текст над полем ввода
      "Число",       // Название окна
      QUESTION_MESSAGE // Тип окна
    );
    // Блок контролируемого кода:
```

```
    try{
      // Попытка преобразовать текст в число:
      num=parseInt(input);
      // Отображение диалогового кона с числами:
      showMessageDialog(null,
        // Отображаемый в окне текст:
        "Числа "+(num-1)+", "+num+" и "+(num+1),
        "Числа",          // Название окна
        INFORMATION_MESSAGE  // Тип окна
      );
    // Обработка ошибок (код выполняется,
    // если в контрольном блоке возникла ошибка):
    }catch(Exception e){
      // Отображение диалогового окна:
      showMessageDialog(null,
        "Что-то пошло не так...", // Сообщение в окне
        "Ошибка",              // Название окна
        ERROR_MESSAGE          // Тип окна
      );
    }
  }
}
```

Прежде чем приступить к анализу программного кода, кратко опишем возможные варианты при выполнении программы. Так, в начале выполнения программы появляется диалоговое окно с текстовым полем, в которое предлагается ввести целое число. На рис. 11.1 показано окно, в поле ввода которого введено значение 20.

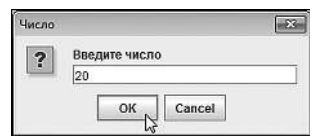


Рис. 11.1. В текстовое поле введено число 20 для считывания программой

После щелчка на кнопке **ОК** появляется новое диалоговое окно — такое, как показано на рис. 11.2.

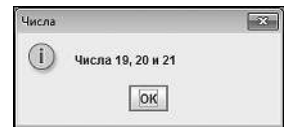


Рис. 11.2. В диалоговом окне отображается три целых числа

Окно содержит значения трех целых чисел (введенного пользователем и двух соседних). Возможна и другая ситуация. На рис. 11.3 показано диалоговое окно с полем ввода, в которое введено текстовое значение "текст".

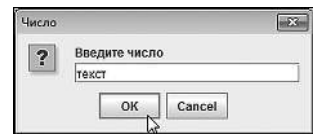


Рис. 11.3. В поле ввода вместо числа указан текст

После щелчка на кнопке **ОК** появляется окно с сообщением об ошибка, как показано на рис. 11.4.

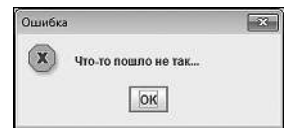


Рис. 11.4. Диалоговое окно с сообщением об ошибке

Такое же окно появляется, если закрыть окно с полем ввода щелчком на кнопке **Cancel** (или на системной пиктограмме закрытия окна в правом верхнем углу).

Далее проанализируем программный код примера.

НА ЗАМЕТКУ

Программный код начинается инструкциями статического импорта, которыми импортируются статические члены класса `JOptionPane` и класса `Integer`. Как следствие, к статическим методам (таким как

showMessageDialog() и showInputDialog()) и константам (таким как ERROR_MESSAGE, QUESTION_MESSAGE и INFORMATION_MESSAGE) класса JOptionPane и к статическому методу parseInt() класса Integer можно обращаться без указания имени соответствующего класса.

В главном методе программы в текстовую переменную input записывается результат вызова метода showInputDialog() (статический метод класса JOptionPane). Далее следует блок кода, выделенный ключевым словом try и фигурными скобками. Это блок *контролируемого кода*. Если при выполнении кода внутри try-блока возникает ошибка, то выполнение кода останавливается и будет выполняться код в catch-блоке, который расположен внизу под try-блоком. Если при выполнении команд в try-блоке ошибок не возникает, то команды в catch-блоке не выполняются.

НА ЗАМЕТКУ

Вся эта схема немного напоминает условный оператор, где в роли «условия» выступает событие, состоящее в возникновении ошибки.

Первой в теле try-блока выполняется команда num=parseInt(input), которой выполняется попытка преобразовать значение текстовой переменной input к целочисленному формату (результат записывается в целочисленную переменную num). Собственно именно здесь мы ожидаем неприятностей. Если значение переменной input не является текстовым представлением целого числа, то возникнет ошибка. В таком случае следующая команда в try-блоке выполняться уже не будет, а выполнится команда в catch-блоке, которой отображается диалоговое окно с сообщением об ошибке. Напротив, если при выполнении команды num=parseInt(input) ошибки не будет, то после нее выполнится следующая команда в try-блоке, которой отображается диалоговое окно с тремя числовыми значениями. Команды в catch-блоке, расположенном после try-блока, будут проигнорированы.

Принципы обработки исключений

В рассмотренном выше примере в круглых скобках после ключевого слова catch было указана инструкция Exception и идентификатор e. Чтобы понять их назначение, следует немного подробнее рассмотреть общую схему обработки исключительных ситуаций. Этим и займемся далее.

Итак, что происходит, если при выполнении программы возникает ошибка? Во-первых, выполнение программы приостанавливается. Во-вторых,

создается объект, который содержит информацию об ошибке, которая возникла. Поскольку объекты создаются на основе классов, то возникает естественный вопрос: на основе какого класса создается объект ошибки? Ответ состоит в том, что в Java есть иерархия классов, каждый из которых соответствует ошибке определенного типа. В вершине этой иерархии находится класс Throwable. У класса Throwable имеется два подкласса: Error и Exception. Класс Error соответствует «фатальным» ошибкам, которые обычно программными методами не обрабатываются. Нас интересует класс Exception. У него очень много подклассов, и среди этих подклассов есть класс RuntimeException. В свою очередь, класс RuntimeException является суперклассом для классов, описывающих всевозможные ошибки, возникающие в процессе выполнения программного кода (классы обсуждаются далее). Собственно это та часть иерархии классов исключений (ошибок), которая для нас представляет наибольший интерес.

Итак, если ошибка возникла, то в соответствии с ее типом создается объект. Этот объект (исключение) передается для обработки методу, который вызвал ошибку (говорят, что исключение *вбрасывается* в метод). Если в методе ошибка обрабатывается — хорошо. Если ошибка в методе не обрабатывается, то объект ошибки передается для обработки в метод, в котором вызывался данный метод (вызвавший ошибку), и так по цепочке, вплоть до главного метода программы. Если и в главном методе ошибка не обрабатывается, то в дело вступает так называемый *обработчик по умолчанию*. Неприятное последствие задействования обработчика по умолчанию в том, что программа досрочно прекращает выполнение.

Теперь рассмотрим процесс обработки исключений. Если мы «подозреваем» некоторый фрагмент кода на предмет того, что он может вызвать ошибку, то такой фрагмент кода помещается в try-блок. После try-блока следуют catch-блоки (и может быть несколько). Каждый catch-блок предназначен для обработки ошибок определенного типа. Под типом ошибки в данном случае имеется в виду класс, на основе которого, в случае возникновения ошибки, создается объект ошибки. Некоторые наиболее актуальные с практической точки зрения классы ошибок перечислены в табл. 11.1.

НА ЗАМЕТКУ

Все перечисленные классы являются подклассами класса RuntimeException. Но нужно понимать, что существует много других классов, используемых при обработке исключительных ситуаций, и они не всегда являются подклассами класса RuntimeException.

Табл. 11.1. Некоторые подклассы класса RuntimeException

Класс исключения	Описание
ArithmeticException	Ошибка, связанная с выполнением арифметических операций
ArrayStoreException	Ошибка, связанная с попыткой записать в массив объект типа, не соответствующего типу элементов массива
ClassCastException	Ошибка, связанная с попыткой некорректного приведения типов
IllegalArgumentException	Ошибка, связанная с некорректной передачей аргументов методу. У данного класса довольно много подклассов. Среди них, в частности, есть класс NumberFormatException, соответствующий ошибке, возникающей при попытке преобразовать некорректное текстовое представление для числа в число
IndexOutOfBoundsException	Ошибка, связанная с некорректно указанным индексом. У данного класса есть два подкласса: класс IndexOutOfBoundsException соответствует ошибке, возникающей при неправильно указанном индексе массива, а класс StringIndexOutOfBoundsException соответствует ошибке, связанной с некорректно указанным индексом элемента в текстовой строке
NegativeArraySizeException	Ошибка, связанная с попыткой создать массив отрицательного размера
NoSuchElementException	Ошибка, связанная с обращением к несуществующему элементу
NullPointerException	Ошибка, связанная с использованием пустой ссылки (значение null) вместо ссылки на объект
UnsupportedOperationException	Ошибка связана с тем, что запрашиваемая операция не поддерживается

В каждом catch-блоке в круглых скобках кроме класса ошибки указывается формальное название для объекта ошибки. При необходимости, данный объект может быть использован в явном виде в программном коде catch-блока.

Если при выполнении программного кода в try-блоке ошибка не возникает, то все catch-блоки игнорируются. Если в try-блоке возникла ошибка, то, как отмечалось ранее, для данной ошибки создается объект, и далее с целью обработки исключения проверяются catch-блоки. Проверка выполняется на предмет совпадения класса объекта ошибки и класса, указанного в catch-блоке. Если совпадение найдено, объект ошибки передается в соответствующий catch-блок для обработки. Важное обстоятельство состоит в том, что «совпадение» понимается в широком смысле: классы могут совпадать не буквально — достаточно, чтобы класс объекта ошибки был подклассом класса, указанного в catch-блоке. В этом смысле, если мы в catch-блоке указываем имя класса Exception, то такой catch-блок будет перехватывать фактически все ошибки (которые в принципе можно обработать), поскольку классы ошибок, с которыми мы имеем дело, прямо или опосредованно являются наследниками класса Exception.

НА ЗАМЕТКУ

В try-catch конструкции после catch-блоков может использоваться finally-блок (блок начинается ключевым словом finally и заключается в фигурные скобки). Программный код из finally-блока выполняется в любом случае: и если возникла ошибка, и если она не возникла. Обычно finally-блоки полезны при использовании вложенных try-блоков (когда внутри try-блока размещается еще одна try-catch конструкция).

Небольшой пример, в котором обрабатываются разные типы ошибок, представлен в листинге 11.2. В представленной программе отображается диалоговое окно, в котором пользователю предлагается указать размер числового массива. Если пользователь вводит корректное значение, то создается соответствующий числовой массив и заполняется символами. Последовательность символов из массива отображается в диалоговом окне. После этого пользователя просят указать индекс элемента в массиве, и после считывания введенного пользователем индекса отображается соответствующий символ. Также в программе отслеживаются возможные ошибки, связанные с тем, что:

- пользователь отменил вод числа или ввел нечисловое значение;
- ввел отрицательное значение для размера массива;
- указал индекс элемента массива, который выходит за допустимый диапазон значений индекса.

При обработке исключений используются классы NumberFormatException (ошибка, возникающая при попытке преобразовать некорректное текстовое представление для числа в число), NegativeArraySizeException (ошибка возникает при попытке создать массив отрицательного размера) и ArrayIndexOutOfBoundsException (ошибка возникает при обращении к элементу по индексу, выходящему за допустимый диапазон значений). Как именно используются данные классы, станет понятно из анализа представленного ниже программного кода.

Листинг 11.2. Программный код проекта MoreTryCatchApplication

```
// Статический импорт:
import static javax.swing.JOptionPane.*;
import static java.lang.Integer.*;
// Главный класс:
```

```
class MoreTryCatchDemo{
  public static void main(String[] args){
    // Переменная для считывания данных из поля ввода:
    String input;
    // Переменная массива:
    char[] symbs;
    // Переменные для записи размера массива
    // и индекса элемента:
    int size,index;
    // Отображение окна с полем ввода для считывания
    // размера массива:
    input=showInputDialog(null,
      "Укажите размер массива", // Текст над полем ввода
      "Символьный массив",     // Название окна
      QUESTION_MESSAGE         // Тип пиктограммы
    );
    // Контролируемый код:
    try{
      // Преобразование текста в число:
      size=parseInt(input);
      // Создание массива:
      symbs=new char[size];
      // Текстовая переменная для формирования
      // текста сообщения:
      String txt="| ";
      // Заполнение массива символами и формирование
      // текста для отображения в окне:
      for(int k=0;k<size;k++){
        // Элементу массива присваивается значение:
        symbs[k]=(char)('A'+k);
        // К тексту дописывается значение элемента:
        txt+=symbs[k]+" | ";
      }
```

```
    // Отображение сообщения с символами из массива:
    showMessageDialog(null,
      txt,              // Сообщение
      "Символы из массива", // Название окна
      INFORMATION_MESSAGE  // Тип сообщения
    );
    // Отображение окна с полем ввода для считывания
    // значения индекса элемента массива:
    input=showInputDialog(null,
      // Текст над полем ввода:
      "Укажите индекс элемента",
      // Название окна:
      "Индекс элемента массива",
      // Тип пиктограммы:
      QUESTION_MESSAGE
    );
    // Преобразование текста в число:
    index=parseInt(input);
    // Текст для отображения в диалоговом окне:
    txt="Индекс — "+index+"\nСимвол — "+symbs[index];
    // Отображение значения элемента массива:
    showMessageDialog(null,
      txt,            // Текст сообщения
      "Символ",       // Название окна
      INFORMATION_MESSAGE // Тип окна
    );
  }
  // Обработка исключительных ситуаций.
  // Пользователь отменил ввод или ввел
  // нечисловое значение:
  catch(NumberFormatException e){
    // Отображение окна с сообщением:
    showMessageDialog(null,
```

```
        // Сообщение:
        "К сожалению, возникала ошибка...",
        "Ошибка",       // Название окна
        WARNING_MESSAGE  // Тип окна
      );
    }
    // Пользователь указал неверный размер для массива:
    catch(NegativeArraySizeException e){
      // Отображение диалогового окна:
      showMessageDialog(null,
        // Сообщение:
        "Некорректный размер массива!",
        // Название окна:
        "Ошибка при создании массива",
        // Пиктограмма:
        ERROR_MESSAGE
      );
    }
    // Пользователь указал неверное значение
    // для индекса элемента массива:
    catch(ArrayIndexOutOfBoundsException e){
      // Отображение диалогового окна:
      showMessageDialog(null,
        // Сообщение:
        "Вы уверены? Такого элемента нет!",
        // Название окна:
        "Ошибка при выборе индекса",
        // Пиктограмма:
        QUESTION_MESSAGE
      );
    }
  }
}
```

Хотя код достаточно большой, он одновременно и простой, поэтому особых комментариев не требует. В процессе выполнения программы последовательно отображаются диалоговые окна: с полем ввода или с сообщением. Ошибки могут возникнуть в тех случаях, когда считанное текстовое значение преобразуется в число, когда создается массив (если размер массива отрицательный) или если для элемента указан неправильный индекс. Для каждой из этих ошибок предназначен свой catch-блок с соответствующим классом, указанным в catch-инструкции.

Возможных исходов выполнения программы несколько. В любом случае в начале выполнения программы появляется диалоговое окно с полем ввода, в котором следует указать размер массива. Окно с заполненным полем показано на рис. 11.5.

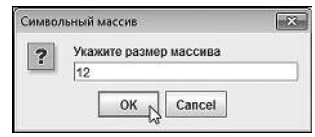


Рис. 11.5. Окно с полем ввода для определения размера массива

Если в поле ввода указано корректное значение, то следующим появляется окно со значениями элементов созданного массива. На рис. 11.6 показано окно, в котором отображается содержимое символьного массива из 12 элементов.

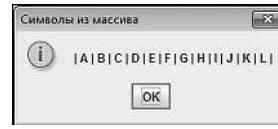


Рис. 11.6. В окне отображается содержимое символьного массива

Следующим открывается окно с полем ввода, в котором нужно указать индекс элемента массива. На рис. 11.7 в поле ввода указано значение 5. Поскольку на предыдущем этапе создавался массив из 12 элементов, то значение 5 для индекса элемента является корректным.

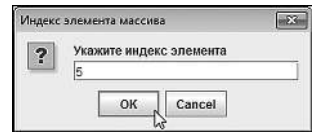


Рис. 11.7. Окно с полем для ввода значения индекса элемента массива

Последним появляется окно (такое, как на рис. 11.8), в котором отображается значение индекса элемента массива и значение элемента (символ).

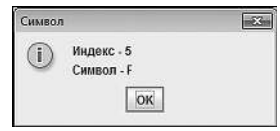


Рис. 11.8. В окне отображается индекс элемента и значение элемента символьного массива

Выше описан, так сказать, «нормальный ход событий». Но возможны и другие сценарии. Например, если при вводе значения для индекса (см. рис. 11.7) пользователь вводит целочисленное значение, но не попадающее в диапазон допустимых значений для индекса элемента массива (если массив из 12 элементов, то индекс может быть целым числом в диапазоне значений от 0 до 11), то появится окно, как на рис. 11.9.

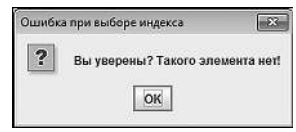


Рис. 11.9. Окно появляется при некорректном индексе для элемента массива

Это последствия «включения в игру» обработчика исключений на основе catch-блока с классом исключения ArrayIndexOutOfBoundsException. Ошибка возникает при попытке получить доступ к элементу с указанным индексом.

Если же пользователь ввел для индекса (см. рис. 11.7) или для размера массива (см. рис. 11.5) нечисловое значение или отменил ввод, то появится окно, как на рис. 11.10.

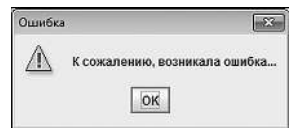


Рис. 11.10. Окно появляется при вводе нечислового значения для размера массива или индекса элемента, а также при отмене ввода пользователем

В данном случае «сработал» catch-блок с классом исключения NumberFormatException. Ошибка возникла при попытке преобразовать введенное пользователем значение к целочисленному формату.

Наконец, если пользователь при определении размера массива (см. рис. 11.5) ввел целое отрицательное число (например, значение -5), то появится окно с сообщение об ошибке. Как выглядит окно, показано на рис. 11.11.

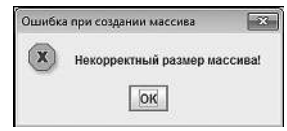


Рис. 11.11. Окно появляется при попытке создать массив отрицательного размера

За отображение окна «отвечает» catch-блок с классом исключения NegativeArraySizeException. Ошибка происходит в момент создания символьного массива.

Вложенные try-catch блоки

Блоки try-catch могут быть вложенными. В таком случае если ошибка возникает во внутреннем try-блоке, то для обработки ошибки сначала просматриваются catch-блоки внутреннего try-блока (в котором возникла ошибка). Если ошибка внутренними catch-блоками не обрабатывается, то исключение передается для обработки во внешних catch-блоках.

Далее рассматривается небольшая иллюстрация к использованию вложенных блоков обработки исключительных ситуаций. Задача, по большому счету, решается та же, что и в предыдущем примере, но способ решения другой. Во-первых, мы используем консольный ввод/вывод. Во-вторых, в программе использованы вложенные try-блоки. Рассмотрим программный код в листинге 11.3.

Листинг 11.3. Программный код проекта NestedTryCatchApplication

```
// Импорт классов:
import java.util.*;
// Главный класс:
class NestedTryCatchDemo{
  // Главный метод:
  public static void main(String[] args){
    // Объект класса Scanner (нужен для реализации
    // консольного ввода):
```

```
Scanner input=new Scanner(System.in);
// Переменная массива:
char[] symbs;
// Переменные для записи размера массива
// и индекса элемента:
int size,index;
// Контролируемый код (внешний блок):
try{
  // Отображение сообщения:
  System.out.print("Укажите размер массива: ");
  // Считывание размера массива:
  size=input.nextInt();
  // Создание массива:
  symbs=new char[size];
  System.out.print("| ");
  // Заполнение массива символами:
  for(int k=0;k<size;k++){
    // Элементу массива присваивается значение:
    symbs[k]=(char)('A'+k);
    // Отображается значение элемента:
    System.out.print(symbs[k]+" | ");
  }
  // Контролируемый код (внутренний блок):
  try{
    System.out.print("\nУкажите индекс: ");
    // Считывание значения индекса:
    index=input.nextInt();
    // Значение элемента:
    System.out.println("Символ — "+symbs[index]);
  }
  // Если пользователь ввел некорректный индекс:
  catch(ArrayIndexOutOfBoundsException e){
    System.out.println("Такого элемента нет");
```

```
      }
      // Блок выполняется всегда:
      finally{
        System.out.println("Массив создан успешно");
      }
      System.out.println("Для индекса указано числовое значение");
    }
    // Если введено не число:
    catch(InputMismatchException e){
      System.out.println("Ошибка: вы не ввели число");
    }
    // Если для массива указан отрицательный размер:
    catch(NegativeArraySizeException e){
      System.out.println("Неверный размер массива");
    }
    System.out.println("Выполнение программы завершено");
  }
}
```

Консольный ввод в программе реализуется на основе объекта input класса Scanner (аргументом конструктору при создании объекта передается ссылка System.in на объект стандартного потока ввода). Для считывания целочисленных значений, который пользователь вводит с клавиатуры, из объекта input вызывается метод nextInt(), возвращающий результатом очередное введенное пользователем число. Если пользователь вместо числа вводит иное значение (например, текст), то при попытке считать такое значение как числовое с помощью метода nextInt() генерируется исключение класса InputMismatchException (ошибка, связанная с некорректностью типа вводимых данных). И класс Scanner, и класс InputMismatchException находятся в пакете java.util, поэтому программа начинается инструкцией import java.util.*, которой импортируются классы из указанного пакета.

ДЕТАЛИ

Класс InputMismatchException является подклассом класса NoSuchElementException (ошибка, связанная с тем, что запрашиваемый элемент не существует), который, в свою очередь, является подклассом класса

RuntimeException (ошибки времени выполнения). Класс NoSuchElementException находится в пакете java.util, в то время как класс RuntimeException находится в пакете java.lang. Классы из пакета java.lang доступны автоматически даже без использования import-инструкции. Для использования классов из других пакетов, их приходится импортировать.

Программа организована так: сначала пользователя просят ввести размер массива. Массив создается и пользователю предлагается ввести индекс элемента массива. Индекс считывается и в окне вывода отображается значение элемента массива с соответствующим индексом. Практически весь код из главного метода размещен в try-блоке. Но внутри этого try-блока есть еще один внутренний try-блок, в который включены команды, связанные со считыванием значения индекса элемента и определением значения соответствующего элемента. Для внутреннего try-блока предусмотрен один catch-блок, в котором обрабатываются исключения класса ArrayIndexOutOfBoundsException. Исключение такого типа генерируется, если пользователь ввел числовое значение для индекса, но оно выходит за диапазон допустимых значений. Еще для внутреннего try-блока описан finally-блок.

НА ЗАМЕТКУ

Код в finally-блоке выполняется в любом случае: и если при выполнении try-блока возникала ошибка, и если при выполнении try-блока ошибки не было. Блок finally в конструкции try-catch можно не использовать.

Для внешнего try-блока предусмотрены два catch-блока, отлавливающие ошибки классов InputMismatchException и NegativeArraySizeException. То есть конструкция получается нетривиальная. Попробуем разобраться, как выполняется данный код.

Если при выполнении программы пользователь все значения вводит корректно, то код выполняется в том порядке, как он выписан, за исключением команд в catch-блоках — эти блоки при отсутствии ошибок (в процессе выполнения программы) игнорируются. Ниже показано, как может выглядеть результат выполнения программы в таком случае (здесь и далее жирным шрифтом выделены значения, вводимые пользователем):

Результат выполнения программы (из листинга 11.3)

```
Укажите размер массива: 12
|A|B|C|D|E|F|G|H|I|J|K|L|
```

```
Укажите индекс: 5
Символ — F
Массив создан успешно
Для индекса указано числовое значение
Выполнение программы завершено
```

Следующая ситуация: пользователь корректно указывает размер массива, но при вводе индекса элемента указывает числовое, но некорректное значение. Результат может быть таким, как показано ниже.

Результат выполнения программы (из листинга 11.3)

```
Укажите размер массива: 12
|A|B|C|D|E|F|G|H|I|J|K|L|
Укажите индекс: -5
Такого элемента нет
Массив создан успешно
Для индекса указано числовое значение
Выполнение программы завершено
```

В данном случае ошибка генерируется во внутреннем try-блоке при попытке выполнить команду System.out.println("Символ — "+symbs[index]), в которой есть инструкция symbs[index] обращения к элементу массива, и индекс элемента некорректный. Если так, то генерируется исключение класса ArrayIndexOutOfBoundsException, и это исключение обрабатывается во внутреннем catch-блоке. После этого продолжается выполнение команд, расположенных после данного catch-блока.

Ниже показан возможный результат выполнения программы в случае, когда размер массива указан корректно, а при вводе значения индекса указано нечисловое значение.

Результат выполнения программы (из листинга 11.3)

```
Укажите размер массива: 12
|A|B|C|D|E|F|G|H|I|J|K|L|
Укажите индекс: пять
Массив создан успешно
```

```
Ошибка: вы не ввели число
Выполнение программы завершено
```

Ошибка возникает во внутреннем try-блоке при попытке выполнить команду index=input.nextInt(). Сгенерированное исключение относится к классу InputMismatchException, и это исключение внутренним catch-блоком не обрабатывается (блок предназначен для обработки ошибок иного типа). Поэтому обработка передается во внешний catch-блок (предусмотренный для обработки ошибок класса InputMismatchException). Но перед этим выполняются команды из finally-блока. Другими словами, внешний catch-блок «примется за дело» только после того, как будет выполнен код в finally-блоке. После выполнения внешнего catch-блока следующей будет выполняться команда после этого блока (команда System.out.println("Выполнение программы завершено")).

Теперь рассмотрим случай, когда при вводе размера массива указано отрицательное число:

Результат выполнения программы (из листинга 11.3)

```
Укажите размер массива: -5
Неверный размер массива
Выполнение программы завершено
```

Исключение класса NegativeArraySizeException генерируется при выполнении команды symbs=new char[size] во внешнем try-блоке и обрабатывается в соответствующем внешнем catch-блоке.

НА ЗАМЕТКУ

Упоминавшийся ранее блок finally в данном случае не выполняется, поскольку он относится к внутренней try-catch конструкции.

Похожая ситуация имеет место, когда пользователь в качестве размера массива вводит нечисловое значение.

Результат выполнения программы (из листинга 11.3)

```
Укажите размер массива: пять
Ошибка: вы не ввели число
Выполнение программы завершено
```

В таком случае при выполнении команды size=input.nextInt() во внешнем try-блоке генерируется исключение класса InputMismatchException, обрабатываемое в специально предусмотренном для ошибок такого типа внешнем catch-блоке.

Использование объекта исключения

В описании catch-блоков неизменно присутствует, кроме имени класса обрабатываемого в блоке исключения, еще и формальное обозначение для объекта исключения. Эти объекты можно использовать при обработке ошибок. В частности, объект исключения можно использовать в выражениях в качестве «текстового» операнда: благодаря переопределенному методу toString() объект исключения в таких случаях автоматически приводится к текстовому формату. Полученное текстовое значение содержит краткую, но все же полезную информацию о возникшей ошибке (в принципе это название класса ошибки и обычно минимальные поясняющие фразы). Очень простой пример представлен в листинге 11.4.

Листинг 11.4. Программный код проекта UsingExceptionObjectApplication

```
class UsingExceptionObjectDemo{
  public static void main(String[] args){
    // Код с ошибкой:
    try{
      System.out.println("Отрицательный размер:");
      int[] a=new int[-1];
    }
    // Обработка ошибки:
    catch(NegativeArraySizeException e){
      System.out.println(e);
    }
    // Код с ошибкой:
    try{
      System.out.println("Неверный индекс:");
      int[] b={1};
      b[-1]=0;
```

```
    }
    // Обработка ошибки:
    catch(ArrayIndexOutOfBoundsException e){
      System.out.println(e);
    }
    // Код с ошибкой:
    try{
      System.out.println("Деление на нуль:");
      int c=10/0;
    }
    // Обработка ошибки:
    catch(ArithmeticException e){
      System.out.println(e);
    }
  }
}
```

Ниже показан результат выполнения программы.

Результат выполнения программы (из листинга 11.4)

```
Отрицательный размер:
java.lang.NegativeArraySizeException
Неверный индекс:
java.lang.ArrayIndexOutOfBoundsException: -1
Деление на нуль:
java.lang.ArithmeticException: / by zero
```

В программе (в главном методе) последовательно размещено три try-catch конструкции. В try-блоке генерируется ошибка определенного типа, а в catch-блоке эта ошибка обрабатывается. Обработка ошибки состоит в том, что объект исключения (во всех трех catch-блоках обозначен как e) передается аргументом методу println(). Последствия приведения объекта исключения к текстовому формату можно оценить по результату выполнения программы.

НА ЗАМЕТКУ

Хотя в каждом catch-блоке объект исключения обозначен одинаково (переменная e), речь идет о разных объектах исключений. В некотором смысле инструкция из имени класса ошибки и названия переменной, обозначающей объект исключения, напоминает инструкцию объявления аргумента метода. Переменная, объявленная в catch-блоке как объект исключения, доступна только в этом catch-блоке.

Генерирование исключений

Сапоги-скороходы болотные — к сожалению, промокают. Ковер-самолет — нелетающий. Шапка...
— Невидимка?
— До сих пор ищем.

Из к/ф «Чародеи»

Хотя на первый взгляд это и выглядит несколько странным, но исключения можно генерировать, так сказать, «вручную». В частности, для искусственного генерирования исключений используется оператор throw, после которого указывается объект исключения. Объект исключения создают специально (как создается любой другой объект), или использую уже сгенерированный объект, переданный для обработки в catch-блок. Пример, в котором используется искусственное генерирование исключений, представлен в листинге 11.5.

Листинг 11.5. Программный код проекта UsingThrowApplication

```
class UsingThrowDemo{
  public static void main(String[] args){
    // Создание объекта исключения:
    Exception me=new Exception("Искусственная ошибка");
    // Контролируемый код (внешний блок):
    try{
      System.out.println("Генерируется ошибка");
      // Контролируемый код (внутренний блок):
      try{
        // Генерирование исключения:
```

```
        throw me;
      }
      // Обработка исключения (внутренний блок):
      catch(Exception one){
        System.out.println(one);
        System.out.println("А теперь еще одна ошибка");
        // Повторное генерирование исключения:
        throw one;
      }
    }
    // Обработка исключения (внешний блок):
    catch(Exception two){
      System.out.println(two);
    }
    System.out.println("Выполнение программы завершено");
  }
}
```

Результат выполнения программы будет таким, как показано ниже:

Результат выполнения программы (из листинга 11.5)

```
Генерируется ошибка
java.lang.Exception: Искусственная ошибка
А теперь еще одна ошибка
java.lang.Exception: Искусственная ошибка
Выполнение программы завершено
```

В программе командой `Exception me=new Exception("Искусственная ошибка")` создается объект `me` класса `Exception`, то есть объект исключения. Текстовый аргумент, переданный конструктору класса `Exception`, служит описанием ошибки. Но создание объекта исключения не означает, что ошибка возникла. Ее необходимо сгенерировать. Для этого используем команду `throw me` во внутреннем `try`-блоке. Ошибка перехватывается и обрабатывается во внутреннем `catch`-блоке. Причем важно понимать, что объект

исключения, отождествляемый в catch-блоке с переменной one, на самом деле является объектом me, который использовался в инструкции генерирования исключения.

НА ЗАМЕТКУ

Если более точно, то и one, и me являются объектными переменными, но ссылаются они на один и тот же объект исключения.

При обработке ошибки во внутреннем catch-блоке, кроме прочего, выполняется команда System.out.println(one). В результате выполнения команды в окне вывода появляется название класса исключения и описание, которое совпадает текстом, переданным конструктору класса Exception при создании объекта me. Далее командой throw one выполняется повторное генерирование исключения. Это исключение перехватывается во внешнем catch-блоке. Причем объект исключения two из этого catch-блока на самом деле является все тем же объектом me. Результат выполнения программы служит тому подтверждением.

Контролируемые и неконтролируемые исключения

Если бы на каждую печь была отдельная кочерга, тогда бы придирались. А так — не гарантирую!

Из к/ф «Безумный день инженера Баркасова»

Все исключения в Java делятся на *контролируемые* и *неконтролируемые*. К неконтролируемым исключениям относятся исключения классов, являющихся подклассами класса RuntimeException или класса Error. Все остальные исключения относятся к контролируемым.

Разница между исключениями разных типов состоит в том, что для контролируемых исключений автоматически выполняется проверка на наличие обработки исключения. Другими словами, если метод может сгенерировать в процессе выполнения контролируемое исключение, то для такого исключения в методе должна быть предусмотрена обработка. Если метод все же не содержит код для обработки контролируемого исключения (но потенциально может такое исключение сгенерировать), то в описании метода в явном виде указывается, что он может

генерировать необрабатываемое контролируемое исключение. Делается это просто: в описании метода после его имени и списка аргументов (но перед фигурными скобками с кодом метода) указывается ключевое слово throws и через запятую перечисляются классы контролируемых исключений, которые метод может сгенерировать, но которые в теле метода не обрабатываются. Некоторые классы контролируемых исключений представлены в табл. 11.2.

Табл. 11.2. Некоторые классы контролируемых исключений

Класс исключения	Описание
ClassNotFoundException	Ошибка, связанная с невозможностью получения доступа к классу
IllegalAccessException	Ошибка при попытке доступа к ресурсу
InstantiationException	Ошибка с невозможностью создания объекта (например, на основе абстрактного класса)
InterruptedException	Ошибка, связанная с прерыванием одного потока другим потоком
NoSuchFieldException	Ошибка, связанная с отсутствием поля
NoSuchMethodException	Ошибка, связанная с отсутствием метода

НА ЗАМЕТКУ

Классы ClassNotFoundException, IllegalAccessException, InstantiationException, NoSuchFieldException и NoSuchMethodException являются подклассами класса ReflectiveOperationException, который, в свою очередь, является подклассом класса Exception. Класс InterruptedException является подклассом класса Exception. Все перечисленные классы находятся в пакете java.lang.

Для большей наглядности рассмотрим несложный программный код, представленный в листинге 11.6.

Листинг 11.6. Программный код проекта UsingCheckedExceptionsApplication

```
class UsingCheckedExceptionsDemo{
  // Метод выбрасывает контролируемое исключение:
  static void alpha(int n) throws Exception{
    // Текст для передачи аргументом конструктору
    // при создании объекта исключения:
    String txt="Метод alpha() с аргументом "+n;
```

```
    // Генерирование исключения:
    throw new Exception(txt);
  }
  // Метод выбрасывает неконтролируемое исключение:
  static void bravo(int n){
    // Текст для передачи аргументом конструктору
    // при создании объекта исключения:
    String txt="Метод bravo() с аргументом "+n;
    // Контролируемый код:
    try{
      if(n>0){
        // Генерирование контролируемого исключения:
        throw new Exception(txt);
      }
      else{
        // Генерирование неконтролируемого исключения:
        throw new RuntimeException(txt);
      }
    }
    // Обработка неконтролируемого исключения:
    catch(RuntimeException e){
      // Повторное генерирование неконтролируемого
      // исключения:
      throw e;
    }
    // Обработка контролируемого исключения:
    catch(Exception e){
      System.out.println("Обработка ошибки в bravo():");
      System.out.println(e.getMessage());
      System.out.println("**************************");
    }
  }
```

```
// Метод для вызова при обработке исключений.
// Аргументом методу передается объект исключения:
static void catchMe(Exception e){
  System.out.println("Обработка ошибки в main():");
  System.out.println(e.getMessage());
  System.out.println("-------------------------");
}
// Главный метод:
public static void main(String[] args){
  // Контролируемый код:
  try{
    // При вызове метода выбрасывается
    // контролируемое исключение класса Exception:
    alpha(123);
  }
  // Обработка исключения:
  catch(Exception e){
    // Вызов метода для обработки исключения:
    catchMe(e);
  }
  // Контролируемый код:
  try{
    // При вызове метода не выбрасывается
    // исключение:
    bravo(123);
  }
  // Обработка исключения (блок не используется,
  // поскольку исключение не генерируется):
  catch(Exception e){
    // Вызов метода для обработки исключения:
    catchMe(e);
  }
```

```
    // Контролируемый код:
    try{
      // При вызове метода выбрасывается
      // неконтролируемое исключение
      // класса RuntimeException:
      bravo(-1);
    }
    // Обработка исключения:
    catch(Exception e){
      // Вызов метода для обработки исключения:
      catchMe(e);
    }
  }
}
```

Ниже показан результат выполнения программы.

Результат выполнения программы (из листинга 11.6)

```
Обработка ошибки в main():
Метод alpha() с аргументом 123
--------------------------
Обработка ошибки в bravo():
Метод bravo() с аргументом 123
**************************
Обработка ошибки в main():
Метод bravo() с аргументом -1
--------------------------
```

Мы описываем в классе UsingCheckedExceptionsDemo несколько статических методов (и один из них метод main()). В методе alpha() командой throw new Exception(txt) генерируется исключение класса Exception (перед этим определяется значение текстовой переменной txt). Поскольку в методе alpha() исключение не обрабатывается, а исключения класса Exception относятся к контролируемым, то в описании метода указана инструкция

throws Exception. Она означает, что метод (может) сгенерировать (или *выбросить*) исключение класса Exception.

НА ЗАМЕТКУ

В команде throw new Exception(txt) после оператора throw указана инструкция new Exception(txt), которой создается анонимный объект класса Exception. Анонимный — потому что ссылка на этот объект в объектную переменную не записывается.

Метод bravo() описан немного сложнее. В теле метода формируется и записывается в текстовую переменную txt текст для передачи аргументом конструктору при создании объекта исключения. Затем происходит такое: проверяется целочисленный аргумент метода, и если он положительный, то командой throw new Exception(txt) генерируется исключение класса Exception. Как мы уже знаем, такое исключение относится к контролируемым. Но, в отличие от метода alpha(), в методе bravo() данное исключение перехватывается и обрабатывается в catch-блоке. Среди команд, выполняемых при обработке исключения класса Exception, есть команда System.out.println(e.getMessage()). В команде использована инструкция e.getMessage(). В данной инструкции из объекта исключения e вызывается метод getMessage(). Метод возвращает результатом текстовое значение с описанием ошибки — в нашем случае речь идет о тексте, который передавался конструктору при создании объекта исключения.

Таким образом, исключение класса Exception если и генерируется в методе, то методом же оно и обрабатывается. Напомним, генерирование такого исключения происходит, если методу bravo() аргументом передано положительное целочисленное значение. В противном случае командой throw new RuntimeException(txt) генерируется исключение класса RuntimeException, которое обрабатывается в catch-блоке. Однако при обработке исключения инструкцией throw e оно генерируется повторно (через e обозначен объект исключения, переданный в catch-блок для обработки).

В итоге получается следующее. Если аргументу bravo() передан положительный аргумент, то в методе генерируется исключение класса Exception, и оно обрабатывается в методе. Если аргументом методу bravo() передано неположительное число, то генерируется исключение класса RuntimeException, оно обрабатывается, но при обработке генерируется повторно. Это повторно сгенерированное исключение методом не обрабатывается. Однако поскольку речь идет об исключении класса

RuntimeException, относящемуся к неконтролируемым исключениям, то в описании метода bravo() инструкцию throws использовать не нужно.

ДЕТАЛИ

В методе bravo() в зависимости от значения аргумента генерируется исключение класса Exception или исключение класса RuntimeException. Соответственно, в методе предусмотрены catch-блоки для обработки исключений данных классов. Специфика ситуации в том, что класс RuntimeException является подклассом класса Exception. Поэтому catch-блок, описанный для исключения класса Exception, мог бы обрабатывать и исключения RuntimeException — если бы не был предусмотрен catch-блок специально для обработки исключений класса RuntimeException.

При генерировании исключения класса RuntimeException оно повторно генерируется в catch-блоке, причем с тем же объектом исключения, что и первый раз. В результате метод bravo() при неположительных значениях аргумента выбрасывает исключение класса RuntimeException. Но если мы попытаемся добиться того же эффекта, ограничившись однократным генерированием исключения класса RuntimeException, отказавшись от обработки (и повторного генерирования) этого исключения в catch-блоке, то такое исключение будет обработано в catch-блоке, предназначенном для обработки исключений класса Exception. И все потому, что класс RuntimeException является подклассом класса Exception.

Вспомогательный статический метод catchMe() предназначен для вызова при обработке исключений в главном методе main(). Этот метод мы используем, чтобы не повторять многократно один и тот же код. Аргументом методу передается объект класса Exception. Но теоретически аргументом методу может быть передан и объект класса RuntimeException. Причина очевидна: класс RuntimeException является подклассом класса Exception, а переменная суперкласса может ссылаться на объект подкласса. При выполнении метода catchMe() отображается информация (сообщение с описанием ошибки) об исключении, объект которого передан аргументом методу.

В главном методе программы один раз вызывается метод alpha() и дважды вызывается метод bravo() (с разными аргументами). Каждая инструкция вызова метода заключается в try-блок, а для обработки возможной ошибки при выполнении метода предусмотрен catch-блок, в котором формально обрабатываются исключения класса Exception, но на самом деле такой блок может обрабатывать практически любую ошибку. Что касается результатов выполнения программы, то они достаточно очевидны и, думается, не требуют особых пояснений.

Создание пользовательских классов исключений

— Кот не заговорил?
— Он делает успехи и старается.
— Я слышу это второй год.

Из к/ф «Чародеи»

Существует возможность создавать пользовательские классы для исключений. Рецепт простой: на основе одного из классов исключений путем наследования создается класс. Объект, созданный на основе такого класса, может использоваться при генерировании исключительных ситуаций. Например, в качестве суперкласса можем использовать класс Exception, или, скажем, класс RuntimeException. В первом случае получим пользовательский класс исключения контролируемого типа, а во втором — неконтролируемого типа.

Небольшая иллюстрация к созданию и использованию пользовательских классов для исключений представлена в листинге 11.7.

Листинг 11.7. Программный код проекта UsingMyExceptionApplication

```
// Класс контролируемого исключения создается
// наследованием класса Exception:
class MyException extends Exception{
  // Закрытое числовое поле:
  private int code;
  // Конструктор:
  MyException(int n){
    super();
    code=n;
  }
  // Переопределение метода toString():
  public String toString(){
    String txt="Исключение класса MyException\n";
    txt+="Код ошибки: "+code+"\n";
    txt+="------------------------------";
    return txt;
```

```
  }
}
// Класс неконтролируемого исключения создается
// наследованием класса RuntimeException:
class MyMistake extends RuntimeException{
  // Закрытое числовое поле:
  private int code;
  // Конструктор:
  MyMistake(int n){
    super();
    code=n;
  }
  // Переопределение метода toString():
  public String toString(){
    String txt="Исключение класса MyMistake\n";
    txt+="Код ошибки: "+code+"\n";
    txt+="******************************";
    return txt;
  }
}
// Главный класс:
class UsingMyExceptionDemo{
  // Статический метод выбрасывает контролируемое
  // исключение класса MyException:
  static void alpha(int n) throws MyException{
    throw new MyException(n);
  }
  // Статический метод выбрасывает неконтролируемое
  // исключение класса MyMistake:
  static void bravo(int n){
    throw new MyMistake(n);
  }
  // Главный метод:
```

```
  public static void main(String[] args){
    // Контролируемый код (внешний блок):
    try{
      // Контролируемый код (внутренний блок):
      try{
        // Метод выбрасывает исключение MyException:
        alpha(123);
      }
      // Обработка исключения класса MyException:
      catch(MyException e){
        System.out.println(e);
        bravo(321);
      }
    }
    // Обработка исключения класса MyMistake:
    catch(MyMistake e){
      System.out.println(e);
    }
  }
}
```

В результате выполнения программы получаем такое:

Результат выполнения программы (из листинга 11.7)

```
Исключение класса MyException
Код ошибки: 123
------------------------------
Исключение класса MyMistake
Код ошибки: 321
******************************
```

В программе описываются однотипные классы MyException и MyMistake. Принципиальная разница между ними в том, что класс MyException создается наследованием класса Exception, а класс MyMistake создается наследованием

класса RuntimeException. У каждого из классов есть закрытое целочисленное поле code, значение которого определяется при вызове конструктора класса. Также в каждом из классов переопределяется метод toString(), благодаря чему объекты класса можно передавать, например, аргументами методу println() (что в программе и делается).

В классе UsingMyExceptionDemo описаны статические методы alpha() и bravo(). Метод alpha() выбрасывает исключение класса MyException, а метод bravo() выбрасывает исключение класса MyMistake. Поскольку пользовательский класс MyException соответствует исключению контролируемого типа, то в описании метода alpha() присутствует инструкция throws MyException.

В главном методе программы последовательно вызываются методы alpha() и bravo(), а генерируемые при этом исключения обрабатываются с помощью вложенных инструкций try-catch.

Резюме

— Прием окончен. Обеденный перерыв.
— Царь трапезничать желает!
Из к/ф «Иван Васильевич меняет профессию»

- Для обработки ошибок (исключений), возникающих в процессе выполнения программы, используется конструкция try-catch. Контролируемый код помещается в try-блок, а для обработки возможных ошибок используются catch-блоки.
- Каждому типу ошибки соответствует определенный класс. Классы исключений образуют иерархию наследования. Наиболее важные классы: Exception, Error, RuntimeException, и ряд других.
- Если при выполнении кода в try-блоке возникает ошибка, то выполнение кода прекращается, автоматически создается объект исключения, и он передается для обработки в catch-блок, соответствующий ошибке данного типа.
- В каждом catch-блоке указывается названия класса для обрабатываемого исключения и название для объекта исключения. В catch-блоке обрабатываются не только сообщения указанного в инструкции класса, но и исключения подклассов данного класса. После выполнения команд в catch-блоке начинает выполняться следующая после конструкции try-catch команда. Если при выполнении try-блока

ошибки не возникли, то catch-блоки игнорируются. Также в try-catch инструкции может использоваться блок finally. Команды в finally-блоке выполняются в любом случае — и при возникновении ошибки, и при отсутствии ошибок.

- Для искусственного генерирования исключения используют оператор throw, после которого указывается объект исключения.
- Все исключения делятся на контролируемые и неконтролируемые. Неконтролируемым исключениям соответствуют классы исключений, являющиеся наследниками классов RuntimeException и Error. Если метод может выбрасывать контролируемое исключение, которое не обрабатывается в методе, то в названии метода указывается инструкция throws, после которой указывается класс выбрасываемого исключения.
- Можно создавать пользовательские классы исключений. Пользовательский класс исключения создается путем наследования одного из классов исключений.

Глава 12
МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ

— Вы уже думаете об этом деле?
— У меня оно не выходит из головы.

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

В Java есть встроенная поддержка для *многопоточного программирования*. При многопоточном программировании несколько блоков программного кода выполняются одновременно. Такие одновременной выполняемые блоки программного кода называются *потоками*. Далее мы познакомимся с тем, как в Java можно потоки создавать, и как затем ими управлять.

Знакомство с потоками

— Крупное дело?
— Давно такого не было!

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Как минимум с одним потоком мы неявно дело уже имели — это так называемый *главный поток* программы, который фактически и есть сама программа. Теперь нам предстоит научиться запускать из главного потока другие потоки, которые мы будем называть *дочерними*.

Чтобы понять механизм разрешения стоящих перед нами задач, имеет смысл абстрагироваться от второстепенных деталей и постараться понять, в чем же все-таки состоит наша конечная цель. В частности, нам необходимо:

- определить программный код, который должен выполняться в рамках запускаемого (дочернего) потока;
- запустить программный код в режиме потока.

Режим запуска программного кода на выполнение важен, поскольку просто запустить код на выполнение — мало. В таком случае речь будет идти о последовательном выполнении команд. А нам необходимо, чтобы команды в дочернем потоке выполнялись одновременно с выполнением главного потока.

Для решения первой задачи, связанной с определением программного кода для выполнения в дочернем потоке, необходимо создать объект класса, реализующего интерфейс Runnable. В интерфейсе Runnable объявлен метод run(). При создании объекта на основе класса, реализующего интерфейс Runnable, определяется код метода run(). Именно код метода run() выполняется при выполнении потока. Другими словами, программный код, который планируется выполнять в рамках дочернего потока, является кодом метода run() объекта, созданного на основе класса, реализующего интерфейс Runnable. С первым пунктом на теоретическом уровне вопрос решен. Теперь остановимся на вопросе запуска дочернего потока.

Для запуска дочернего потока необходимо создать объект класса Thread и вызвать из этого объекта метод start(). Здесь все просто. Но есть один момент: необходимо связать объект, в методе run() которого реализован программный код потока, с объектом класса Thread, из которого вызывается метод start() для запуска потока. Такое «связывание» происходит при создании объекта класса Thread: аргументом конструктору класса Thread передается объект с методом run(). Это общая схема. Дальше начинаются нюансы.

ДЕТАЛИ

Ситуация еще более «изощренная», чем может показаться на первый взгляд. Дело в том, что класс Thread реализует интерфейс Runnable, но только с пустым телом для метода run(). Проще говоря, в классе Thread метод run() определен так, что в нем нет команд. С другой стороны, если создать подкласс путем наследования класса Thread, то в этом подклассе можно переопределить метод run() (и соответственно, класс реализует интерфейс Runnable), у этого класса будет метод start(), который можно использовать для запуска дочернего потока. Поэтому один из путей создания дочернего потока базируется на создании подкласса класса Thread. Такой подход мы также рассмотрим.

И поскольку практика является лучшим критерием истины и подтверждением теории, далее рассмотрим некоторые характерные примеры создания дочерних потоков.

Способы создания дочерних потоков

— Какие ваши соображения?
— Запутанная история!
— Как это верно, Ватсон!

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Далее мы рассмотрим некоторые, наиболее характерные способы создания в главном потоке дочернего потока.

Явная реализация интерфейса Runnable

В рассматриваемом далее примере отдельно описывается класс, реализующий интерфейс Runnable, и созданный на его основе объект передается аргументом конструктору класса Thread при создании объекта этого класса. Рассмотрим программный код, представленный в листинге 12.1.

Листинг 12.1. Программный код проекта CreatingThreadApplication

```
// Класс реализует интерфейс Runnable:
class MyThread implements Runnable{
  // Описание метода run() (программный код для потока):
  public void run(){
    for(int i=1;i<=5;i++){
      // Отображение сообщения:
      System.out.println("Дочерний поток:\t"+i);
      try{
        // Задержка в выполнении потока:
        Thread.sleep(1200);
      } // Обработка исключения:
      catch(InterruptedException e){
        System.out.println("Прерывание дочернего потока");
      }
    }
  }
}
```

```
// Главный класс:
class CreatingThreadDemo{
  // Главный поток:
  public static void main(String[] args){
    System.out.println("Начинается выполнение программы");
    // Создание объекта класса Thread для запуска
    // дочернего потока на выполнение:
    Thread t=new Thread(new MyThread());
    System.out.println("Запускается дочерний поток");
    // Запуск дочернего потока:
    t.start();
    for(int k=0;k<=5;k++){
      // Отображение сообщения:
      System.out.println("Главный поток:\t"+(char)('A'+k));
      try{
        // Задержка в выполнении главного потока:
        Thread.sleep(2000);
      } // Обработка исключения:
      catch(InterruptedException e){
        System.out.println("Прерывание главного потока");
      }
    }
    System.out.println("Выполнение программы завершено");
  }
}
```

Результат выполнения программы, скорее всего, будет таким:

Результат выполнения программы (из листинга 12.1)

```
Начинается выполнение программы
Запускается дочерний поток
Главный поток:   A
Дочерний поток:  1
```

```
Дочерний поток: 2
Главный поток: B
Дочерний поток: 3
Дочерний поток: 4
Главный поток: C
Дочерний поток: 5
Главный поток: D
Главный поток: E
Главный поток: F
Выполнение программы завершено
```

Проанализируем программный код. В первую очередь достоин внимания класс MyThread, в котором реализуется интерфейс Runnable. Фактически класс состоит из описания метода run(). Метод не имеет аргументов и не возвращает результат. В нашей программе код метода run() выполняется в рамках дочернего потока (почему и как это происходит — увидим далее).

В методе run() запускается оператор цикла. За каждый цикл выводится сообщение о выполнении дочернего потока и целочисленное значение из натурального ряда чисел. Между выводом сообщений выполняется временная пауза интервалом в 1200 миллисекунд. Для выполнения задержки используется статический метод sleep() из класса Thread. Методом выполняется задержка в выполнении потока на время (в миллисекундах), переданное аргументом методу. Поскольку метод sleep() может выбрасывать контролируемое сообщение класса InterruptedException (исключение, связанное с прерыванием выполнения потока), то вызов метода sleep() выполняется внутри try-блока, после которого размещен catch-блок с кодом, выполняемым при возникновении исключения.

ДЕТАЛИ

До выполнения catch-блока дело обычно не доходит. Также следует учесть, что в принципе обработку исключения в методе main() можно было не выполнять, но тогда в описании метода main() пришлось бы указывать инструкцию throws с указанием класса InterruptedException исключения, которое может быть выброшено методом. Что касается метода run(), то в интерфейсе Runnable он объявлен как такой, что не выбрасывает исключение класса InterruptedException, поэтому и описан он должен быть соответствующим образом.

В главном методе программы (в классе CreatingThreadDemo), прежде всего, выводится сообщение о начале выполнения программы. Затем командой Thread t=new Thread(new MyThread()) создается объект t класса Thread. Нам этот объект нужен для запуска на выполнение дочернего потока. Аргументом конструктору класса Thread передается выражение new MyThread(), которым создается анонимный объект класса MyThread. Объект t связан с этим анонимным объектом и при выполнении команды t.start() начинает выполняться (в режиме дочернего потока) программный код метода run() из упомянутого анонимного объекта.

Сразу после запуска на выполнение дочернего потока в главном потоке (в методе main()) запускается оператор цикла, в котором отображается сообщение о выполнении главного потока и, кроме этого, очередная латинская буква, начиная с литеры 'A'. Между выводом сообщений выполняется пауза в 2000 миллисекунд. Для выполнения паузы используем команду Thread.sleep(2000). Причины использования конструкции try-catch такие же, как и в случае с методом run(). В конце выполнения программы отображается сообщение о завершении ее работы.

ДЕТАЛИ

Выражение (char)('A'+k) вычисляется так: к коду символа 'A' прибавляется целочисленное значение переменной k, а полученное целочисленное значение преобразуется к символьному виду. Правило преобразования целого числа к символьному виду такое: число интерпретируется как код символа в кодовой таблице.

Что касается метода sleep(), то он приостанавливает выполнение потока, из которого вызывается. Поэтому когда метод sleep() вызывается в методе run(), то речь идет о приостановке выполнения дочернего потока, а когда метод sleep() вызывается в методе main(), то приостанавливается выполнение главного потока.

Хочется подчеркнуть важный момент: после выполнения команды t.start() дочерний (метод run()) и главный (метод main()) потоки выполняются параллельно. Поэтому сообщения от дочернего и главного потока в окне вывода появляются «вперемешку». Числовые параметры для выполнения задержки дочернего и главного потоков подобраны так, что дочерний поток завершает выполнение раньше главного потока. Но вообще при выполнении нескольких потоков нет гарантии, что дочерние потоки будут завершены на момент завершения главного потока. Проще говоря, теоретически может оказаться, что в момент завершения главного

потока дочерние потоки еще будут выполняться. Проверить, выполняется ли поток, можно с помощью метода isAlive(), который вызывается из объекта потока. Метод возвращает результатом значение true если поток выполняется, и false в противном случае. Если необходимо дождаться завершения выполнения некоторого потока, то из объекта этого потока вызывают метод join().

НА ЗАМЕТКУ

Метод join() может выбрасывать неконтролируемое исключение класса InterruptedException.

Создание потока с использованием анонимного класса

Теперь рассмотрим практически тот же пример, но вместо явного описания класса, реализующего интерфейс Runnable, прибегнем к помощи анонимного объекта анонимного класса, реализующего данный интерфейс. Рассмотрим программный код, представленный в листинге 12.2.

Листинг 12.2. Программный код проекта NewThreadApplication

```
class NewThreadDemo{
  public static void main(String[] args) throws InterruptedException{
    System.out.println("Начинается выполнение программы");
    // Аргументом конструктору класса Thread передается
    // анонимный объект анонимного класса:
    Thread t=new Thread(new Runnable(){
      public void run(){
        for(int i=1;i<=5;i++){
          System.out.println("Дочерний поток:\t"+i);
          try{
            Thread.sleep(4500);
          }
          catch(InterruptedException e){
            System.out.println("Прерывание дочернего потока");
          }
        }
```

```
      }
    });
    System.out.println("Запускается дочерний поток");
    t.start();
    for(int k=0;k<=5;k++){
      System.out.println("Главный поток:\t"+(char)('A'+k));
      Thread.sleep(2000);
    }
    // Проверка дочернего потока:
    if(t.isAlive()){
      // Ожидание завершения дочернего потока:
      t.join();
    }
    System.out.println("Выполнение программы завершено");
  }
}
```

Как выглядит результат выполнения программы, показано ниже.

Результат выполнения программы (из листинга 12.2)

```
Начинается выполнение программы
Запускается дочерний поток
Главный поток:  A
Дочерний поток: 1
Главный поток:  B
Главный поток:  C
Дочерний поток: 2
Главный поток:  D
Главный поток:  E
Дочерний поток: 3
Главный поток:  F
Дочерний поток: 4
Дочерний поток: 5
Выполнение программы завершено
```

По сравнению с предыдущим примером мы внесли минимальные, но показательные изменения. В первую очередь, мы не описываем класс, реализующий интерфейс Runnable, а вместо этого при создании объекта t класса Thread в главном методе программы аргументом конструктору передается анонимный объект анонимного класса. Анонимный класс создается реализацией интерфейса Runnable.

Во-вторых, в главном методе не обрабатывается ошибка класса InterruptedException, которая потенциально может возникнуть при выполнении команды Thread.sleep(2000). Поэтому в описании метода main() указана инструкция throws InterruptedException.

В-третьих, в описании метода run() в коде создания анонимного объекта анонимного класса (аргумент конструктора класса Thread) при вызове метода sleep() указано значение 4500. Поэтому при выполнении дочернего потока сообщения отображаются с интервалом в 4,5 секунды. В главном методе программы (главный поток) сообщения отображаются с интервалом в 2 секунды. Хотя в главном потоке (в операторе цикла) отображается на одно сообщение больше, чем в дочернем, легко понять, что дочерний поток должен выполняться дольше, чем главный поток.

Однако перед отображением последнего сообщения в главном потоке в условном операторе проверяется значение выражения t.isAlive(). Если значение выражения равно true, то дочерний поток еще выполняется, и тогда благодаря команде t.join() главный поток приостанавливает свое выполнение до тех пор, пока не завершится выполнение дочернего потока.

Поэтому последнее сообщение из главного потока (метода main()) выполняется после завершения дочернего потока.

ДЕТАЛИ

Поскольку метод join() может выбрасывать контролируемое исключение класса InterruptedException, то инструкцию вызова метода join() следует размещать либо в try-блоке с предусмотренной возможностью перехвата и обработки исключения, либо метод, в котором вызывается метод join(), в описании должен содержать throws-инструкцию с названием класса InterruptedException. Используя такую инструкцию в описании метода main() мы решаем сразу две проблемы: можно не использовать обработку ошибки класса InterruptedException ни при вызове метода sleep(), ни при вызове метода join().

Создание потока с использованием лямбда-выражения

Использование лямбда-выражений нередко позволяет сделать код проще. Для большей наглядности мы рассмотрим все тот же пример, но на этот раз вместо анонимного объекта анонимного класса используем лямбда-выражение. Интересующий нас код представлен в листинге 12.3.

Листинг 12.3. Программный код проекта LambdaInThreadApplication

```
class LambdaInThreadDemo{
  public static void main(String[] args) throws InterruptedException{
    System.out.println("Начинается выполнение программы");
    // Интерфейсной переменной значением
    // присваивается лямбда-выражение:
    Runnable r=()->{
      for(int i=1;i<=5;i++){
        System.out.println("Дочерний поток:\t"+i);
        try{
          Thread.sleep(4500);
        }
        catch(InterruptedException e){
          System.out.println("Прерывание дочернего потока");
        }
      }
    };
    // Интерфейсная переменная передается аргументом
    // конструктору класса Thread:
    Thread t=new Thread(r);
    System.out.println("Запускается дочерний поток");
    t.start();
    for(int k=0;k<=5;k++){
      System.out.println("Главный поток:\t"+(char)('A'+k));
      Thread.sleep(2000);
    }
    if(t.isAlive()){
```

```
      t.join();
    }
    System.out.println("Выполнение программы завершено");
  }
}
```

Результат выполнения программы ожидается аналогичным к тому, как это было в предыдущем случае:

Результат выполнения программы (из листинга 12.3)

```
Начинается выполнение программы
Запускается дочерний поток
Главный поток: A
Дочерний поток: 1
Главный поток: B
Главный поток: C
Дочерний поток: 2
Главный поток: D
Главный поток: E
Дочерний поток: 3
Главный поток: F
Дочерний поток: 4
Дочерний поток: 5
Выполнение программы завершено
```

В данной программе в главном методе объявляется переменная r интерфейсного типа Runnable. Значением переменной присваивается лямбда-выражение, определяющее, фактически, код метода run(), выполняемого в рамках дочернего потока. В результате этой операции автоматически создается объект анонимного класса, реализующего интерфейс Runnable, а ссылка на этот объект записывается в переменную r. Метод run() в объекте реализуется с кодом, определяемым лямбда-выражением.

НА ЗАМЕТКУ

Мы воспользовались тем обстоятельством, что интерфейс Runnable является функциональным. В интерфейсе Runnable имеется

единственный абстрактный метод run(). Поэтому интерфейсной переменной типа Runnable можно присвоить значением лямбда-выражение, определяющее код метода, не имеющего аргументов и не возвращающего результат.

Интерфейсная переменная r передается аргументом конструктору класса Thread при создании объекта потока. Во всем остальном код должен быть понятен читателю.

Наследование класса Thread

Как отмечалось ранее, в классе Thread реализуется интерфейс Runnable, но метод run() там содержит пустое тело — то есть при выполнении такого метода ничего не происходит. Но при наследовании класса Thread в подклассе метод run() можно переопределить. Таким образом, объект подкласса, наследующего класс Thread, будет иметь как метод start() для запуска потока, так и метод run() с кодом, выполняемым в рамках данного потока. Программа, аналогичная рассмотренным ранее, в которой используется наследование класса Thread, представлена в листинге 12.4.

Листинг 12.4. Программный код проекта ExtendingThreadApplication

```
// Подкласс класса Thread:
class MyThread extends Thread{
  // Переопределение метода run():
  public void run(){
    for(int i=1;i<=5;i++){
      System.out.println("Дочерний поток:\t"+i);
      try{
        Thread.sleep(4500);
      }
      catch(InterruptedException e){
        System.out.println("Прерывание дочернего потока");
      }
    }
  }
}
// Главный класс:
```

```
class ExtendingThreadDemo{
  public static void main(String[] args) throws InterruptedException{
    System.out.println("Начинается выполнение программы");
    // Создание объекта класса MyThread:
    MyThread t=new MyThread();
    System.out.println("Запускается дочерний поток");
    // Запуск дочернего потока:
    t.start();
    for(int k=0;k<=5;k++){
      System.out.println("Главный поток:\t"+(char)('A'+k));
      Thread.sleep(2000);
    }
    if(t.isAlive()){
      t.join();
    }
    System.out.println("Выполнение программы завершено");
  }
}
```

Результат выполнения программы фактически такой же, как и в предыдущем примере (см. листинг 12.3 и результаты его выполнения).

Мы использовали в данной программе простой и интуитивно понятный код. Подкласс MyThread создается наследованием класса Thread. В теле класса переопределяется метод run(). В главном методе программы создается объект t класса MyThread.

При создании объекта t конструктору аргументы не передаются. Это означает, что при вызове конструктора суперкласса Thread() аргументы ему также не передаются. В таком случае при вызове из данного объекта метода start() поиск метода run() выполняется в том же объекте. Поэтому кода из объекта t вызывается метод start(), то в режиме дочернего потока начинает выполняться метод run().

НА ЗАМЕТКУ

Если из объекта t просто вызвать метод run() командой t.run(), то поток запущен не будет. В таком случае программный код метода run() будет выполняться в главном потоке.

Существуют и иные способы реализации дочерних потоков. Некоторые еще будут рассмотрены далее.

Работа с потоками

После истории с часами я готов верить всему, что вы скажете. Но, черт возьми, как?!

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Далее мы уделим некоторое внимание собственно работе с потоками. Начнем с потока, который запускается первым — речь о главном потоке.

Главный поток

Хотя мы теперь уже знаем, что выполнение программы начинается с выполнения главного потока, но открытым остается вопрос о том, как получить доступ к этому потоку. Полезным будет статический метод currentThread() класса Thread. Результатом метод возвращает ссылку на объект того потока, в процессе выполнения которого вызывается метод. Поэтому если метод currentThread() вызывается в методе main(), то результатом метода будет ссылка на объект главного потока. Небольшой пример с получением доступа к объекту главного потока представлен в листинге 12.5.

Листинг 12.5. Программный код проекта MainThreadApplication

```
class MainThreadDemo{
  public static void main(String[] args){
    // Объектная переменная для записи ссылки на поток:
    Thread t;
    // Получение ссылки на объект главного потока:
    t=Thread.currentThread();
    // Отображение информации о потоке:
    System.out.println(t);
    // Изменение имени потока:
    t.setName("Главный поток");
```

```
    // Задается приоритетность потока:
    t.setPriority(7);
    // Отображение информации о потоке:
    System.out.println(t);
  }
}
```

В результате выполнения программы в окне вывода появляются следующие сообщения:

Результат выполнения программы (из листинга 12.5)

```
Thread[main,5,main]
Thread[Главный поток,7,main]
```

В программе объявляется переменная t класса Thread, значением которой присваивается результат вызова статического метода currentThread() из класса Thread. В итоге переменная t значением получает ссылку на объект потока, в котором вызывался метод currentThread() — в данном случае это объект главного потока. Поскольку в классе Thread переопределен метод toString(), то объекты класса можно передавать аргументом в метод println(). В результате отображается некоторая полезная информация о потоке: после ключевого слова Thread в квадратных скобках отображается *название потока*, *приоритет потока* и *группа*, к которой относится поток.

ДЕТАЛИ

Название потока представляет собой текстовое значение, идентифицирующее поток. У главного потока по умолчанию название main. У каждого потока есть приоритет — целочисленное значение в диапазоне от 1 до 10. Собственно само значение не столь принципиально, но при «конкуренции за ресурсы» разных потоков имеет значение, у какого потока приоритет больше. Для главного потока по умолчанию приоритет равен 5.

Также потоки разбиваются на группы потоков (некоторые операции могут выполняться сразу для всей группы потоков). Главный поток относится к группе main.

Командой t.setName("Главный поток") главному потоку присваивается новое имя, а с помощью команды t.setPriority(7) задается новое значение для

приоритета главного потока. С помощью команды System.out.println(t) отображаются новые сведения о потоке.

Методы для работы с потоками

Есть определенное количество методов, предназначенных для работы с потоками. Часть из методов мы уже использовали. Небольшой список таких методов (включая и уже использовавшиеся нами методы) представлены и кратко описаны в табл. 12.1.

Табл. 12.1. Методы для работы с потоками

Метод	Описание
activeCount()	Методом возвращается количество активных потоков в группе
checkAccess()	Метод позволяет выполнить проверку на предмет того, может ли указанный поток быть изменен выполняемым потоком
currentThread()	Метод возвращает результатом ссылку на объект текущего потока (потока, в котором вызывается метод)
enumerate()	Метод предназначен для копирования в массив ссылок на все активные потоки в данной группе потоков (и ее подгруппах)
getId()	Методом в качестве результата возвращается идентификатор потока — целое число, идентифицирующее поток
getName()	Метод в качестве результата возвращает имя потока
getPriority()	Метод возвращает результатом приоритет потока (целое число в диапазоне значений от 1 до 10)
getState()	Метод в качестве результата возвращается объект, определяющий статус потока
getThreadGroup()	Результатом метода является объект, определяющий группу, к которой принадлежит поток
holdsLock()	Метод в качестве результата возвращается логическое значение, определяющее, удерживает ли метод *монитор* (специальный объект, используемый для блокировки некоторого ресурса с целью недопущения обращения к нему сразу нескольких методов)
interrupt()	Метод предназначен для выполнения прерывания потока
interrupted()	Метод используется для проверки того, был ли поток прерван. При этом меняется статус потока (в отношении прерывания)
isAlive()	Метод для определения того, активен ли поток
isDaemon()	Проверка потока на предмет того, является ли он *демон-потоком* (такие потоки обычно выполняются в фоновом режиме и автоматически завершаются при завершении главного потока)
isInterrupted()	Метод используется для проверки того, был ли поток прерван. При этом статус потока (в отношении прерывания) не меняется
join()	Вызов метода приводит к тому, что поток, в котором вызывается метод, ожидает завершения потока, из объекта которого вызывается метод

Метод	Описание
notify()	При вызове метода один из потоков, находящихся в режиме ожидания (для получения доступа к ресурсу), переводится в режим выполнения
notifyAll()	При вызове метода все потоки, находящиеся в режиме ожидания (для получения доступа к ресурсу), переводятся в режим выполнения
run()	Метод с кодом, выполняемым в рамках потока. Определяет точку входа в поток
setDaemon()	Метод позволяет определить поток как *демон-поток* (особый вид потоков, которые автоматически завершаются при завершении выполнения главного потока)
setName()	Метод позволяет задать название для потока
setPriority()	Метод предназначен для присваивания значения приоритету потока
sleep()	Метод позволяет приостановить выполнение потока. Аргументом методу передается целое число, определяющее время (в миллисекундах) приостановки в выполнении потока
start()	Метод вызывается при запуске потока на выполнение
wait()	Метод предназначен для перевода потока в режим ожидания (при получении доступа к ресурсу)
yield()	Методом посылается сигнал диспетчеру потоков, что текущий поток готов уступить использование процессора в пользу других потоков

Некоторые из приведенных методов имеют разные версии, некоторые выбрасывают контролируемые исключения. Понятно, что при использовании того или иного метода такие моменты следует принимать во внимание.

Приведенный список (хотя он и неполный) методов для работы с потоками позволяет составить некоторое представление об операциях, которые выполняются с потоками.

Создание нескольких потоков

Далее рассмотрим небольшой пример, в котором создается несколько дочерних потоков. Для создания потоков описывается класс MyThread, являющийся подклассом класса Thread. Код класса MyThread описан так, что создание объекта класса MyThread приводит к запуску дочернего потока.

Каждый поток выводит в консольное окно сообщения. Между выводом сообщений выполняется приостановка выполнения потока. Время приостановки выполнения потока является случайным (в диапазоне от 1 до 3 секунд). А теперь проанализируем программный код в листинге 12.6.

Листинг 12.6. Программный код проекта MultiThreadApplication

```
// Импорт класса Random:
import java.util.Random;
// Подкласс MyThread создается наследованием
// суперкласса Thread:
class MyThread extends Thread{
  // Количество сообщений:
  private int count;
  // Конструктор:
  MyThread(String name,int count){
    // Вызов конструктора суперкласса:
    super(name);
    // Значение целочисленного поля:
    this.count=count;
    // Запуск потока на выполнение:
    start();
  }
  // Переопределение метода run():
  public void run(){
    // Отображение сообщения о запуске потока:
    System.out.println("Выполняется поток "+getName());
    // Создание объекта класса Random
    // для генерирования случайных чисел:
    Random rnd=new Random();
    // Оператор цикла, в котором выводятся сообщения:
    for(int k=1;k<=count;k++){
      System.out.println("Сообщение от потока "+getName()+":\t"+getName().charAt(0)+k);
      try{
        // Задержка в выполнении потока:
        Thread.sleep(1000+rnd.nextInt(2001));
      }
      catch(InterruptedException e){
        System.out.println("Прерывание потока "+getName());
```

```
      }
    }
    // Сообщение о завершении выполнения потока:
    System.out.println("Поток "+getName()+" завершен");
  }
}
// Главный класс:
class MultiThreadDemo{
  public static void main(String[] args) throws InterruptedException{
    System.out.println("Начинает выполняться главный поток");
    // Создание объектов — запуск потоков:
    MyThread A=new MyThread("ALPHA",5);
    MyThread B=new MyThread("BRAVO",3);
    MyThread C=new MyThread("CHARLIE",7);
    // Сообщения главного потока:
    for(int k=1;k<=4;k++){
      System.out.println("Сообщение от главного потока:\t"+k);
      // Задержка в выполнении потока:
      Thread.sleep(2000);
    }
    // Ожидание завершения дочерних потоков:
    if(A.isAlive()){
      A.join();
    }
    if(B.isAlive()){
      B.join();
    }
    if(C.isAlive()){
      C.join();
    }
    // Сообщение о завершении главного потока:
    System.out.println("Главный поток завершен");
  }
}
```

Результат выполнения программы (с учетом использования генератора случайных чисел) может быть таким:

Результат выполнения программы (из листинга 12.6)

```
Начинает выполняться главный поток
Выполняется поток ALPHA
Сообщение от главного потока: 1
Выполняется поток BRAVO
Выполняется поток CHARLIE
Сообщение от потока BRAVO:    B1
Сообщение от потока CHARLIE:  C1
Сообщение от потока ALPHA:    A1
Сообщение от потока BRAVO:    B2
Сообщение от потока ALPHA:    A2
Сообщение от главного потока: 2
Сообщение от потока ALPHA:    A3
Сообщение от потока CHARLIE:  C2
Сообщение от главного потока: 3
Сообщение от потока BRAVO:    B3
Поток BRAVO завершен
Сообщение от потока CHARLIE:  C3
Сообщение от потока ALPHA:    A4
Сообщение от главного потока: 4
Сообщение от потока ALPHA:    A5
Сообщение от потока CHARLIE:  C4
Поток ALPHA завершен
Сообщение от потока CHARLIE:  C5
Сообщение от потока CHARLIE:  C6
Сообщение от потока CHARLIE:  C7
Поток CHARLIE завершен
Главный поток завершен
```

В классе `MyThread` в закрытое целочисленное поле `count` записывается значение, определяющее количество сообщений (не считая первого

и последнего), которые выводятся в консольное окно при выполнении потока. Значение поля определяется при создании объекта с помощью второго аргумента, который передается конструктору. Первым аргументом конструктору передается текстовое значение, которое затем передается конструктору суперкласса (которым, напомним, является класс Thread).

ДЕТАЛИ

Конструктор класса Thread имеет несколько версий. В частности, можно вызывать конструктор класса Thread:

- без аргументов — создается объект потока с параметрами по умолчанию;
- с текстовым аргументом — такой аргумент задает название потока;
- с аргументом, являющимся объектом класса, реализующим интерфейс Runnable — такой аргумент содержит метод run(), код которого выполняется в потоке;
- можно передать первым аргументом объект с методом run(), а вторым — текстовое значение, определяющее название потока.

Существуют и другие версии конструктора класса Thread.

Таким образом, при создании объекта класса MyThread первым аргументом конструктору передается название потока, а вторым аргументом указывается количество «основных» сообщений, отображаемых в консольном окне при выполнении потока. Более того, поскольку в конструкторе класса MyThread выполняется команда start(), то создание объекта класса MyThread означает автоматический запуск дочернего потока.

При выполнении дочернего потока выполняется код метода run(), описанного в классе MyThread (это на самом деле переопределение метода run() из класса Thread). В теле метода run() командой System.out.println("Выполняется поток "+getName()) отображается сообщение о запуске потока. Результатом инструкции getName() является имя потока (то, что было указано при создании объекта класса MyThread). Командой Random rnd=new Random() создается объект rnd класса Random.

НА ЗАМЕТКУ

Для использования класса Random в начале программы инструкцией import java.util.Random выполняется импортирование данного класса.

С помощью объекта rnd мы генерируем случайные числа: в команде Thread.sleep(1000+rnd.nextInt(2001)) использовано выражение rnd.nextInt(2001),

значением которого является случайное целое число в диапазоне значений от 0 до 2000. Таким образом, при выполнении потока между сообщениями выполняется временная задержка продолжительностью от 1 до 3 секунд (результатом выражения 1000+rnd.nextInt(2001) является целое число в диапазоне значений от 1000 до 3000). Сами сообщения отображаются в операторе цикла командой System.out.println("Сообщение от потока "+getName()+":\ t"+getName().charAt(0)+k), в которой выражение getName().charAt(0) значением возвращает первый символ в названии потока.

ДЕТАЛИ

Методом getName() возвращается текстовое значение с названием потока. Из этого текста вызывается метод charAt() с аргументом 0. Метод charAt() результатом возвращает символ из текстовой строки, который имеет индекс, переданный аргументом методу. Поэтому инструкцией getName().charAt(0) возвращается первый (с нулевым индексом) символ в названии потока. Этот символ дописывается к предыдущей текстовой строке. Затем к этой текстовой строке дописывается значение индексной переменной k. В итоге сообщение дочернего потока содержит название потока, а также комбинацию из первой буквы в названии потока и порядковый номер сообщения.

По завершении выполнения потока командой System.out.println("Поток "+getName()+" завершен") отображается соответствующее сообщение.

В главном методе программы создаются объекты A, B и C класса MyThread. При создании объектов автоматически запускаются дочерние потоки. Главный поток с интервалом в 2 секунды также выводит серию сообщений. Завершение главного потока происходит после завершения всех дочерних потоков.

Создание демон-потока

Есть специальный тип потоков, которые называются *демон-потоками*. Обычно такие потоки выполняются в фоновом режиме и играют некоторую «вспомогательную» роль по «обслуживанию» других потоков. Главная особенность демон-потока состоит в том, что при завершении главного потока демон-поток завершается автоматически. В некоторых случаях такие потоки бывают очень полезными. Далее мы рассмотрим небольшой пример, связанный с использованием демон-потока.

Мы рассмотрим задачу о вычислении суммы натуральных чисел. Но программу организуем с привлечением дочернего демон-потока. А именно, при запуске программы запускается демон-поток, в котором, собственно, происходит вычисление суммы натуральных чисел. Каждое новое значение для суммы отображается в консольном окне. Одновременно из главного потока отображается диалоговое окно, в котором пользователь должен нажать одну из двух кнопок (**Yes** или **No**). Если пользователь нажимает кнопку **No**, то окно закрывается, но процесс вычисления суммы продолжается, и окно с двумя кнопками появится снова. Все это продолжается до тех пор, пока пользователь не нажмет в окне кнопку **Yes**. Чтобы понять, как такая схема реализуется в программном коде, рассмотрим программу из листинга 12.7.

Листинг 12.7. Программный код проекта DaemonThreadApplication

```
// Статический импорт:
import static javax.swing.JOptionPane.*;
class DaemonThreadDemo{
  public static void main(String[] args) throws InterruptedException{
    // Создание объекта для дочернего потока. Первый
    // аргумент конструктора является лямбда-выражением,
    // второй аргумент — название потока:
    Thread t=new Thread(()->{
      // Индексная переменная и
      // переменная для записи суммы чисел:
      int k=1,s=0;
      // Бесконечный цикл для вычисления суммы:
      while(true){
        // Отображение сообщения:
        System.out.println(Thread.currentThread().getName()+": "+s);
        try{
          // Задержка в выполнении потока:
          Thread.sleep(1000);
        } // Обработка исключения:
        catch(InterruptedException e){}
        // Прибавление слагаемого к сумме:
```

```
        s+=k;
        // Увеличение значения индексной переменной:
        k++;
      }
    },"Вычисление суммы");
    // Статус демон-потока:
    t.setDaemon(true);
    // Переменная для записи результат
    // выбора пользователя (нажатая кнопка):
    int res;
    // Запуск потока на выполнение:
    t.start();
    // Отображение диалогового окна:
    do{
      // Задержка в выполнении потока:
      Thread.sleep(3000);
      // Отображение окна и запоминание
      // выбора пользователя:
      res=showConfirmDialog(null,
        // Текст в окне:
        "Закончить вычисление суммы?",
        // Название окна:
        "Сумма чисел",
        // Кнопки в окне:
        YES_NO_OPTION);
    }while(res!=YES_OPTION);
  }
}
```

При запуске программы на выполнение появляется диалоговое окно, представленное на рис. 12.1.

Если нажать кнопку **No** или закрыть окно щелчком на системной пиктограмме в правом верхнем углу окна, окно закрывается, но через 3 секунды

появляется снова. Все это время с интервалом в 1 секунду в консольном окне появляется сообщение с очередным значением для суммы, как показано ниже:

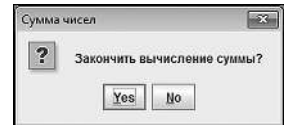


Рис. 12.1. В процессе выполнения программы периодически отображается диалоговое окно с двумя кнопками

Результат выполнения программы (из листинга 12.7)

```
Вычисление суммы: 0
Вычисление суммы: 1
Вычисление суммы: 3
Вычисление суммы: 6
Вычисление суммы: 10
Вычисление суммы: 15
Вычисление суммы: 21
Вычисление суммы: 28
```

Так продолжается до тех пор, пока пользователь в диалоговом окне не щелкнет кнопку **Yes**. После щелчка на кнопке **Yes** окно закрывается, а также завершается выполнение дочернего демон-потока (и поэтому отображение сообщений в консольном окне прекращается).

Теперь проанализируем программный код примера. В нем есть несколько специфических моментов. Так, для создания дочернего потока в главном методе программы создается объект t класса Thread. Первым аргументом конструктору передается лямбда-выражение, определяющее код, выполняемый в рамках потока. В этом коде запускается бесконечный оператор цикла (в while-операторе условием указано значение true), поэтому формально сумма вычисляется до бесконечности. За каждый цикл отображается название потока и текущее значение для суммы. Для определения названия потока использована инструкция Thread.currentThread().getName(). В ней метод getName(), возвращающий результатом название потока, вызывается из ссылки на исполняемый поток. Ссылку на исполняемый поток получаем посредством выражения Thread.currentThread(). Отображение сообщений (равно как и вычисление нового

значения для суммы) происходит с интервалом в 1 секунду, а для обработки исключения класса InterruptedException использован пустой catch-блок (в нем нет команд).

В главном методе программы командой t.setDaemon(true) мы определяем поток, реализуемый через объект t, как демон-поток. Запускаем его на выполнение с помощью команды t.start().

После этого начинается вычисление суммы чисел в рамках дочернего демон-потока. Одновременно в главном потоке запускается оператор цикла do-while, в котором сначала делается задержка в 3 секунды, а затем вызывается метод showConfirmDialog().

Это статический метод из класса JOptionPane. Но поскольку мы воспользовались в программе процедурой статического импорта статических членов класса JOptionPane, то при вызове метода и обращении к статическим константам метода имя класса можно не указывать.

При вызове метода showConfirmDialog() отображается диалоговое окно с несколькими кнопками, стандартной пиктограммой (вопросительный знак) и текстом в области окна. Аргументы методу могут передаваться по-разному, но в данном случае первым аргументом передается пустая ссылка null на родительское окно (такого окна в данном случае нет), второй текстовый аргумент "Закончить вычисление суммы?" определяет текст в окне, третий аргумент "Сумма чисел" определяет название окна, а статическая константа YES_NO_OPTION определяет кнопки, отображаемые в окне (в данном случае будут кнопки **Yes** и **No**).

НА ЗАМЕТКУ

Могут использоваться такие константы для определения кнопок, отображаемых в диалоговом окне: DEFAULT_OPTION (по умолчанию отображается кнопка **OK**), YES_NO_OPTION (отображаются кнопки **Yes** и **No**), YES_NO_CANCEL_OPTION (отображаются кнопки **Yes**, **No** и **Cancel**) и OK_CANCEL_OPTION (отображаются кнопки **OK** и **Cancel**).

Результатом метод showConfirmDialog() возвращает целочисленное значение, определяющее кнопку, которую нажал пользователь в диалоговом окне. Если была нажата кнопка **Yes**, то возвращаемое методом showConfirmDialog() значение равно константе YES_OPTION. Поскольку результат вызова метода записывается в целочисленную переменную res, то условие выполнения оператора цикла do-while имеет вид res!=YES_OPTION.

НА ЗАМЕТКУ

Константы, определяющие результат метода showConfirmDialog(), такие: YES_OPTION (нажата кнопка **Yes**), NO_OPTION (нажата кнопка **No**), CANCEL_OPTION (нажата кнопка **Cancel**), OK_OPTION (нажата кнопка **OK**) и CLOSED_OPTION (окно закрыто щелчком на системной пиктограмме).

Синхронизация потоков

— Ну что, не передумали?
— Мне выбирать не приходится.

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Нередко случается так, что несколько потоков в процессе выполнения обращаются к общему ресурсу. Здесь может возникнуть проблема, связанная с тем, что несколько потоков пытаются одновременно выполнить некоторые операции с данным общим ресурсом. Решение проблемы может быть связано с *синхронизацией потоков* при обращении к ресурсу. Идея синхронизации простая — общий ресурс при обращении к нему одного потока блокируется для всех других потоков. Технически все реализуется просто. Например, если речь идет о синхронизации потоков при обращении к некоторому объекту, то используется такой шаблон:

```
synchronized(объект){
  // код синхронизации
}
```

Если нужно синхронизировать метод (такой метод будет доступен в каждый момент только для одного потока), то в сигнатуре метода указывается ключевое слово synchronized. Небольшая программа, в которой используется синхронизация потоков, представлена в листинге 12.8.

Листинг 12.8. Программный код проекта SynchronizedThreadsApplication

```
// Класс с целочисленным полем:
class MyNumber{
  int number;
}
// Класс для создания потоков:
```

```
class MyThread extends Thread{
  // Ссылка на объект с целочисленным полем:
  private MyNumber obj;
  // Интервал приостановки в выполнении потока:
  private int time;
  // Количество циклов:
  private int count;
  // Поле логического типа:
  private boolean state;
  // Конструктор:
  MyThread(String name,MyNumber obj,int time,int count,boolean state){
    // Вызов конструктора суперкласса:
    super(name);
    // Ссылка на объект:
    this.obj=obj;
    // Интервал задержки:
    this.time=time;
    // Количество циклов:
    this.count=count;
    // Значение поля логического типа:
    this.state=state;
    // Запуск потока:
    start();
  }
  // Метод для отображения "линии" из символов:
  private void line(){
    // Локальная символьная переменная:
    char s;
    // Значение переменной:
    if(state) s='-';
    else s='*';
    // Отображение последовательности символов:
    for(int k=1;k<=35;k++){
```

```
      System.out.print(s);
    }
    System.out.println("");
  }
  // Переопределение метода run():
  public void run(){
    // Оператор цикла:
    for(int k=1;k<=count;k++){
      // Блок синхронизации:
      synchronized(obj){
        // Отображается "линия":
        line();
        // Отображение сообщения:
        System.out.println("Поток "+getName()+": исходное значение "+obj.number);
        // Контролируемый код:
        try{
          // Задержка в выполнении потока:
          Thread.sleep(time);
        } // Обработка исключения:
        catch(InterruptedException e){
          System.out.println(e);
        }
        // Изменение значения поля объекта:
        if(state) obj.number++;
        else obj.number--;
        // Отображение сообщения:
        System.out.println("Поток "+getName()+": новое значение "+obj.number);
        // Отображение "линии":
        line();
      } // Завершение блока синхронизации
    }
  }
}
```

```
// Главный класс:
class SynchronizedThreadsDemo{
  // Главный метод:
  public static void main(String[] args){
    // Целочисленные переменные:
    int n=100,count=5,time=1000,dt=200;
    // Создание объекта с целочисленным полем:
    MyNumber obj=new MyNumber();
    // Присваивание полю объекта значения:
    obj.number=n;
    // Создание первого потока:
    MyThread Alpha=new MyThread("Alpha",obj,time+dt,count,true);
    // Создание второго потока:
    MyThread Bravo=new MyThread("Bravo",obj,time-dt,count,false);
    // Контролируемый код:
    try{
      // Ожидание завершения потоков:
      if(Alpha.isAlive()) Alpha.join();
      if(Bravo.isAlive()) Bravo.join();
    } // Обработка исключения:
    catch(InterruptedException e){
      System.out.println(e);
    }
  }
}
```

Результат выполнения программы, скорее всего, будет таким:

Результат выполнения программы (из листинга 12.8)

```
-----------------------------------
Поток Alpha: исходное значение 100
Поток Alpha: новое значение 101
-----------------------------------
```

```
**********************************
Поток Bravo: исходное значение 101
Поток Bravo: новое значение 100
**********************************
**********************************
Поток Bravo: исходное значение 100
Поток Bravo: новое значение 99
**********************************
**********************************
Поток Bravo: исходное значение 99
Поток Bravo: новое значение 98
**********************************
**********************************
Поток Bravo: исходное значение 98
Поток Bravo: новое значение 97
**********************************
-----------------------------------
Поток Alpha: исходное значение 97
Поток Alpha: новое значение 98
-----------------------------------
-----------------------------------
Поток Alpha: исходное значение 98
Поток Alpha: новое значение 99
-----------------------------------
-----------------------------------
Поток Alpha: исходное значение 99
Поток Alpha: новое значение 100
-----------------------------------
-----------------------------------
Поток Alpha: исходное значение 100
Поток Alpha: новое значение 101
-----------------------------------
```

```
**********************************
Поток Bravo: исходное значение 101
Поток Bravo: новое значение 100
**********************************
```

Программа достаточно простая. В процессе ее выполнения идет «борьба» между двумя дочерними потоками за «ресурс», в роли которого выступает объект с целочисленным полем. Объект создается на основе класса MyNumber. Класс очень простой: в нем описано всего одно открытое целочисленное поле number.

Потоки создаются на основе класса MyThread, наследующего класс Thread. При создании объекта класса MyThread конструктору передается несколько аргументов, в том числе и ссылка на объект класса MyNumber. Именно с этим объектом будут выполняться «операции»: в зависимости от значения поля state логического типа при выполнении потока поле number объекта класса MyNumber увеличивается на единицу или уменьшается на единицу. Изменение значения поля number происходит при выполнении оператора цикла в методе run(). Количество циклов задается полем number. Весь процесс состоит из двух этапов. Сначала считывается текущее значение поля number, выводится сообщение о считанном значении, затем делается пауза в выполнении потока (продолжительность паузы определяется значением поля time), и в зависимости от значения поля state значение поля number увеличивается (при значении state равном true) или уменьшается (при значении state равном false) на единицу. Также в начале каждого цикла и в конце каждого цикла в окне вывода отображается импровизированная «линия», состоящая из символов '-' если поле state равно true, и состоящая из символов '*' при значении поля state равном false. Отображение «линии» реализовано через закрытый метод line().

> **НА ЗАМЕТКУ**
>
> При отображении сообщений указывается название потока. Название потока получаем при вызове метода getName(). Задается название потока с помощью первого аргумента конструктора класса MyThread. Этот аргумент передается аргументом при вызове конструктора суперкласса.

Поскольку в конструкторе класса MyThread есть вызов метода start(), то создание объекта класса MyThread приводит к автоматическому

запуску дочернего потока. В главном методе программы (в классе SynchronizedThreadsDemo) запускаются два потока (объекты Alpha и Bravo класса MyThread), которые имеют одновременный доступ к объекту obj класса MyNumber. При выполнении потока Alpha за каждый цикл значение поля number объекта obj увеличивается на единицу, а при выполнении потока Bravo значение поля number уменьшается на единицу. Поскольку количество циклов в каждом потоке выполняется одно и то же, то в итоге значение поля number объекта obj останется неизменным. Вместе с тем, в каждом потоке между отображением сообщения о текущем значении поля и отображением сообщения о новом значении поля проходит какое-то время (для каждого потока свое). Поэтому, если не выполнить синхронизацию потоков, сообщения разных потоков будут отображаться «вперемешку». Чтобы этого не было, в классе MyThread в описании метода run(), команды в теле оператора цикла помещены в synchronized-блок. В круглых скобках после ключевого слова synchronized указано название объекта, доступ к которому блокируется если с ним уже «работает» поток. Поэтому сообщения от потоков появляются «блоками»: если поток начал считывание значения поля number объекта obj, то другой поток в этот процесс уже не вклинится, пока объект не получит новое значение для поля и оно не будет отображено в окне вывода.

НА ЗАМЕТКУ

Однако один поток может «вклиниться» в работу другого потока между выполнением разных циклов. Желающие могут поэкспериментировать с программным кодом, разместив, например, в блок синхронизации не тело цикла, а весь оператор цикла. Также полезно посмотреть, как изменится результат выполнения программы, если блок синхронизации не использовать вовсе.

Резюме

Сначала меня это забавляло. С одной стороны.
Но с другой — я понимал, что пора объясниться.

Из к/ф «Приключения Шерлока Холмса
и доктора Ватсона»

- Потоки представляют собой блоки программного кода, выполняемые одновременной. Для реализации потоков используют класс Thread и интерфейс Runnable.

- Общая схема создания потока состоит в том, что создается объект класса, реализующего интерфейс Runnable. В классе описывается метод run() из данного интерфейса. Метод run() определяет код потока. Для запуска потока создается объект класса Thread. Аргументом конструктору класса Thread передается объект с методом run(). Поток запускается при вызове метода start() из объекта класса Thread.
- Интерфейс Runnable является функциональным, поэтому при создании потоков можно использовать лямбда-выражения.
- В классе Thread реализуется интерфейс Runnable с пустым кодом для метода run(). Поэтому создание потока может базироваться на наследовании класса Thread с переопределением в подклассе метода run().
- Существует группа методов, предназначенных для работы с потоками. В некоторых случаях приходится выполнять синхронизацию потоков для регулирования доступа потоков к общему ресурсу.

Глава 13
ПРИЛОЖЕНИЯ С ГРАФИЧЕСКИМ ИНТЕРФЕЙСОМ

Ох, красота-то какая! Лепота!

Из к/ф «Иван Васильевич меняет профессию»

В этой главе мы приступаем к рассмотрению способов создания приложений с графическим интерфейсом. Мы уже использовали элементы графического интерфейса ранее. Но там речь шла об использовании стандартных окон. В данном случае мы будем создавать графические компоненты практически «с нуля». Правда, сразу следует отметить, что благодаря наличию в Java специальных библиотек для разработки графического интерфейса, создавать приложения с графическим интерфейсом на самом деле не очень сложно. Вместе с тем, чтобы успешно использовать широкие возможности Java, создавая приложения с графическим интерфейсом, важно понять и освоить некоторые базовые приемы и подходы. Собственно, о них речь и пойдет далее.

Принципы создания приложений с интерфейсом

Ведь это же настоящая тайна! Ты потом никогда себе не простишь!

Из к/ф «Гостья из будущего»

Задачу по созданию приложения с графическим интерфейсом условно можно разделить на два неравноценных этапа:

- создание собственно компонентов графического интерфейса;
- организация обработки событий, связанных с графическими компонентами.

Первая задача решается относительно просто. Вторая является более сложной. Мы перед рассмотрением примеров сначала проанализируем общие подходы.

Итак, создание того или иного графического компонента подразумевает создание некоторого объекта. Объект создается на основе класса, соответствующего графическому компоненту. Проще говоря, для каждого типа графического компонента существует класс (и обычно не один), на основе которого следует создать объект. В Java есть две основные библиотеки классов, предназначенных для создания графических компонентов: это библиотека AWT и библиотека Swing. Графические компоненты можно создавать с использованием классов или одной, или другой библиотеки. Например, для создания такого графического компонента как кнопка, можно использовать класс `JButton` из библиотеки Swing или класс `Button` из библиотеки AWT. Далее, чтобы создать окно используют класс `JFrame` из библиотеки Swing или класс `Frame` из библиотеки AWT. Объект для текстовой метки может создаваться на основе класса `JLabel` из библиотеки Swing или на основе класса `Label` из библиотеки AWT. И так практически для всех компонентов. Мы для создания компонентов графического интерфейса будем использовать классы из библиотеки Swing. Главная причина состоит в том, что для компонентов, созданных на основе Swing-классов, спектр «возможностей» несколько шире по сравнению с аналогичными компонентами, созданными на основе классов библиотеки AWT.

НА ЗАМЕТКУ

Сказанное вовсе не означает, что мы не будем использовать библиотеку AWT. Наоборот, без этой библиотеки крайне проблематично организовать обработку событий. Просто при создании компонентов интерфейса будут использоваться классы из библиотеки Swing.

Принцип «конструирования» сложных компонентов графического интерфейса сводится к тому, что создаются его отдельные компоненты (путем создания объектов соответствующих классов), выполняются настройки их параметров, а затем эти компоненты добавляются и размещаются в контейнерах (специальных компонентах, вроде окна или панели, которые могут содержать в себе другие компоненты интерфейса). Для выполнения всех перечисленных операций имеются специальные методы, так что процесс оформления «дизайна» сложностей в большинстве случаев не вызывает.

Немного сложнее проходит процесс «обучения» активных компонентов интерфейса. Ведь, кроме того, чтобы поместить компонент в «правильном месте», важно определить его реакцию на всевозможные события.

Примером такого события может быть щелчок на кнопке, ввод символа в поле ввода или выбор пункта в меню. Во всех таких случаях используется *обработка событий*. Как она выполняется, мы рассмотрим немного позже, когда приступим к обсуждению программ с функциональным графическим интерфейсом.

Создание окна

Начинаю действовать без шума и пыли по вновь утвержденному плану.

Из к/ф «Бриллиантовая рука»

В первом, рассматриваемом далее, примере обработка событий не используется, поскольку там нет компонентов, которые должны реагировать на события (речь пойдет о создании пустого окна). Затем примеры будут более сложные. На них мы проиллюстрируем основные подходы, используемые при создании графических компонентов. Причем в некоторых случаях для решения одной и той же задачи мы будем использовать разные подходы. Очень часто такой метод бывает полезным. Начнем с того, что выясним, как стандартными средствами Java можно создать пустое окно (окно, не содержащее никаких других компонентов).

Пустое окно

Как отмечалось выше, за создание окна в библиотеке Swing «отвечает» класс JFrame. Для создания окна необходимо создать объект класса JFrame (объект окна) и выполнять ряд настроек. Настройки выполняются путем вызова специальных методов из объекта класса окна. Как именно все это реализуется в виде программного кода, показано в листинге 13.1.

Листинг 13.1. Программный код проекта JustAWindowApplication

```
// Импорт класса JFrame:
import javax.swing.JFrame;
// Главный класс:
class JustAWindowDemo{
  // Главный метод:
  public static void main(String[] args){
    // Создание объекта окна:
```

```
    JFrame wnd=new JFrame("Обычное окно");
    // Размеры окна:
    wnd.setSize(300,200);
    // Положение окна на экране:
    wnd.setLocation(250,250);
    // Окно постоянных размеров:
    wnd.setResizable(false);
    // Реакция на щелчок системной пиктограммы:
    wnd.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    // Отображение окна на экране:
    wnd.setVisible(true);
  }
}
```

При запуске данной программы на выполнение появляется окно, представленное на рис. 13.1.

Рис. 13.1. При запуске программы отображается пустое окно

Это обычное серое окно с названием **Обычное окно** (в строке названия). Если в правом верхнем углу окна щелкнуть системную пиктограмму с крестиком, окно закроется, а программа прекратит выполнение. Это, в общем-то, единственная «функциональность», заложенная в окно посредством программного кода из листинга 13.1 (конечно, если не считать возможности перетаскивать окно и сворачивать/разворачивать его).

НА ЗАМЕТКУ

Созданное окно является окном постоянных размеров: окно с помощью системных пиктограмм можно свернуть и развернуть, но нельзя изменить его размер.

Теперь проанализируем программный код, благодаря которому отображается данное окно. Начальной инструкцией import javax.swing.JFrame импортируется класс JFrame. Все остальное происходит в главном методе main() в классе JustAWindowDemo. Там командой JFrame wnd=new JFrame("Обычное окно") создается объект wnd класса JFrame. Это объект окна. Конструктору класса JFrame передается текст "Обычное окно", определяющий название окна.

НА ЗАМЕТКУ

Сразу заметим, что создание объекта окна не означает, что окно отображается. Для отображения окна используется метод setVisible(), который вызывается с аргументом true из объекта окна. Обычно данный метод вызывают после того, как настроены все параметры окна.

Размеры окна задаются с помощью команды wnd.setSize(300,200). Первый аргумент 300 метода setSize() определяет ширину окна в пикселах, а второй аргумент 200 задает высоту окна в пикселах. Положение окна на экране определяется командой wnd.setLocation(250,250). Два аргумента, которые передаются методу setLocation(), определяют положение левого верхнего угла окна по отношению к левому верхнему углу экрана.

НА ЗАМЕТКУ

Вместо использования методов setSize() и setLocation(), можно было воспользоваться методом setBounds(). Метод setBounds() позволяет задать положение и размеры компонента. В частности, вместо команд wnd.setSize(300,200) и wnd.setLocation(250,250) мы могли использовать всего одну команду wnd.setBounds(250,250,300,200).

ДЕТАЛИ

Координаты компонентов определяются в пикселах в системе координат контейнера. Начало системы координат контейнера находится в его левом верхнем углу. Горизонтальная координатная ось направлена слева направо. Вертикальная координатная ось направлена сверху вниз. Координаты компонента в контейнере — это координаты точки в его левом верхнем углу. Координаты окна определяются по отношению к точке в левом верхнем углу экрана.

Команда wnd.setResizable(false) означает, что окно имеет постоянные размеры и поэтому после отображения нельзя будет с помощью мышки «растянуть» размеры окна.

НА ЗАМЕТКУ

Несложно догадаться, что вызов из объекта окна метода setResizable() с аргументом true приводит к тому, что размеры окна можно будет изменять. Такой режим для окна используется по умолчанию.

Стандартное создаваемое окно содержит пиктограмму с крестиком. Однако нам необходимо определить реакцию окна (и программы) на щелчок пользователя на системной пиктограмме. Для этого из объекта окна вызывается метод setDefaultCloseOperation(). Аргументом методу передается статическая константа EXIT_ON_CLOSE класса JFrame, означающая, что при закрытии окна щелчком на системной пиктограмме программа должна завершить выполнение.

Наконец, командой wnd.setVisible(true) окно отображается на экране.

НА ЗАМЕТКУ

Если из объекта окна вызвать метод setVisisble() с аргументом false, то окно будет убрано с экрана. Не свернуто, а именно убрано. Впоследствии вызвав из объекта этого же окна метод setVisisble() с аргументом true можно будет снова отобразить окно на экране.

Альтернативный способ создания окна

Выше все команды по созданию и настройке параметров окна выполнялись в главном методе программы. Такой подход на самом деле очень неудобный и на практике не используется. Более разумный подход состоит в том, чтобы создать путем наследования на основе класса JFrame пользовательский подкласс. Именно такой подход реализован в программе в листинге 13.2.

Листинг 13.2. Программный код проекта SimpleWindowApplication

```
// Импорт класса JFrame:
import javax.swing.JFrame;
// Подкласс создан наследованием суперкласса JFrame:
class MyFrame extends JFrame{
  // Конструктор:
  MyFrame(String name){
```

```
    // Вызов конструктора суперкласса:
    super(name);
    // Положение и размеры окна:
    setBounds(250,250,300,200);
    // Окно постоянных размеров:
    setResizable(false);
    // Реакция на щелчок системной пиктограммы:
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // Отображение окна на экране:
    setVisible(true);
  }
}
// Главный класс:
class SimpleWindowDemo{
  // Главный метод:
  public static void main(String[] args){
    // Создание анонимного объекта подкласса:
    new MyFrame("Обычное окно");
  }
}
```

Результат выполнения программы точно такой же, как и в предыдущем случае. Но код немного иной. В данном случае мы описываем класс MyFrame, который наследует класс JFrame. Весь код класса MyFrame по большому счету состоит из описания конструктора. Именно в конструктор перенесены все команды по созданию и настройке параметров окна. Первой командой вызывается конструктор суперкласса — то есть конструктор класса JFrame. Аргументом конструктору суперкласса передается текстовое значение, которое в свою очередь, является конструктором класса MyFrame(). Это текстовое значение определяет название окна. Положение на экране и размеры окна определяются командой setBounds(250,250,300,200). Командой setResizable(false) делаем размеры окна постоянными, реакция на щелчок системной пиктограммы в окне определяется командой setDefaultCloseOperation(EXIT_ON_CLOSE), а с помощью команды setVisible(true) окно отображается на экране. Таким образом, создание объекта класса MyFrame

означает не только создание объекта окна, но и приводит к отображению окна на экране.

> **НА ЗАМЕТКУ**
>
> При вызове методов, унаследованных из класса JFrame в конструкторе класса MyFrame объект, из которого вызывается метод, не указывается. В таком случае речь идет о вызове из создаваемого объекта — то есть из объекта окна. Также указывая статическую константу EXIT_ON_CLOSE класса JFrame имя класса JFrame можно не указывать, поскольку константа через механизм наследования доступна в классе MyFrame «напрямую».

В главном методе программы командой new MyFrame("Обычное окно") создается анонимный объект класса MyFrame (переданный конструктору текст определяет название окна). То, что объект создается анонимный — не принципиально. Просто мы не планируем выполнять дальнейшие операции с окном, поэтому нет смысла запоминать ссылку на объект окна.

Окно с кнопкой

Искусство по-прежнему в большом долгу.

Из к/ф «Покровские ворота»

В следующем примере создается окно с текстом в главной области и кнопкой, щелчок на которой приводит к закрытию окна и завершению выполнения программы. Здесь, по сравнению с предыдущим примером, добавляется еще два графических компонента: метка с текстом и кнопка. С меткой ситуация простая: это элемент статический (в том смысле, что такая метка не будет реагировать на события), поэтому метку нужно просто создать и добавить в окно. Кнопку тоже нужно создать и добавить в окно, но кроме этого необходимо еще «научить» кнопку реагировать на щелчок, то есть предусмотреть элементарную обработку событий.

Явное использование объекта обработчика

Далее мы сталкиваемся с *обработкой событий*. Поступим мы так: сначала рассмотрим пример, а затем будут общие пояснения относительно принципов реализации обработки событий в Java.

Но, прежде чем приступить к анализу программного кода, сделаем несколько замечаний. Как мы помним, для создания графического компонента необходимо создать объект класса, соответствующего этому компоненту. Мы уже знаем, что для создания окна создается объект класса JFrame. За создание кнопок «отвечает» класс JButton, а текст мы будем добавлять с использованием текстовой метки (объект класса JLabel). Все перечисленные классы относятся к библиотеке Swing. Поэтому программный код начинается инструкцией import javax.swing.*, которой подключаются все классы из пакета javax.swing. Также нам понадобятся некоторые утилиты, связанные с обработкой событий, из пакета java.awt.event, поэтому в программе есть еще и инструкция import java.awt.enent.*.

А теперь рассмотрим программный код, представленный в листинге 13.3.

Листинг 13.3. Программный код проекта WindowWithButtonApplication

```
// Импорт классов:
import javax.swing.*;
import java.awt.event.*;
// Класс для создания окна:
class MyFrame extends JFrame{
  // Конструктор:
  MyFrame(){
    // Вызов конструктора суперкласса:
    super("Окно с кнопкой");
    // Положение и размеры окна:
    setBounds(250,250,300,200);
    // Окно постоянных размеров:
    setResizable(false);
    // Реакция на щелчок системной пиктограммы:
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // Отключение менеджера компоновки:
    setLayout(null);
    // Создание объекта метки:
    JLabel lbl=new JLabel("Щелчок на кнопке приводит к закрытию окна");
    // Положение и размеры метки:
```

```
    lbl.setBounds(10,30,280,50);
    // Добавление метки в окно:
    add(lbl);
    // Создание объекта кнопки:
    JButton btn=new JButton("Закрыть окно");
    // Положение и размеры кнопки:
    btn.setBounds(50,120,200,30);
    // Создание объекта для обработчика события,
    // происходящего при щелчке кнопки:
    MyHandler hnd=new MyHandler();
    // Регистрация обработчика в кнопке:
    btn.addActionListener(hnd);
    // Добавление кнопки в окно:
    add(btn);
    // Отображение окна на экране:
    setVisible(true);
  }
}
// Класс обработчика:
class MyHandler implements ActionListener{
  // Определение метода из интерфейса:
  public void actionPerformed(ActionEvent e){
    // Завершение выполнения программы:
    System.exit(0);
  }
}
// Главный класс:
class WindowWithButtonDemo{
  // Главный метод:
  public static void main(String[] args){
    // Создание объекта окна:
    new MyFrame();
  }
}
```

При запуске программы на выполнение появляется диалоговое окно, представленное на рис. 13.2.

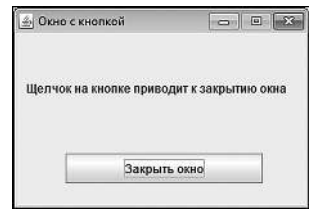


Рис. 13.2. Окно с текстом и кнопкой, щелчок на которой приводит к закрытию окна

При щелчке на кнопке **Закрыть окно** (или на системной пиктограмме с крестиком в правом верхнем углу окна) окно закрывается.

Что касается программного кода, то здесь есть несколько новых для нас моментов. Основу программы составляет класс MyFrame, являющийся подклассом суперкласса JFrame. Большинство команд нам знакомы, но есть и новые. Так, командой setLayout(null) для объекта окна отключается *менеджер компоновки*. Менеджер компоновки отвечает за размещение компонентов в контейнере (в данном случае в окне). Если его не отключить, то параметры, задающие размер и положение метки и кнопки в окне, будут проигнорированы, и компоненты в таком случае размещаются в соответствии со встроенными алгоритмами менеджера компоновки (по умолчанию компоненты добавляются в окно в один ряд слева направо в порядке добавления компонентов).

НА ЗАМЕТКУ

Менеджеров компоновки существует несколько, и каждый из них имеет свои алгоритмы размещения компонентов в контейнере. Метод setLayout() предназначен для определения менеджера компоновки, поэтому в общем случае аргументом менеджеру передается объект менеджера компоновки. Но если аргументом методу передать пустую ссылку null, то менеджер компоновки в контейнере, из которого вызывается метод setLayout(), использоваться не будет.

Объект lbl для метки создается командой JLabel lbl=new JLabel("Щелчок на кнопке приводит к закрытию окна"). Метка — это прямоугольная область с текстом (хотя может содержать и изображение). В данном случае аргументом конструктору класса JLabel передан текст, который отображается в метке.

После создания объекта метки командой lbl.setBounds(10,30,280,50) задается положение и размер метки в контейнере, которым в данном случае выступает окно. В соответствии с приведенной командой левый верхний угол метки будет находиться на расстоянии в 10 пикселов вправо и 30 пикселов вниз от левого верхнего угла контейнера, в который будет помещаться метка. Ширина метки составляет 280 пикселов, а высота равна 50 пикселам. Командой add(lbl) метка добавляется в окно.

ДЕТАЛИ

Метод add() позволяет добавлять компоненты в контейнер. Вызывается метод из объекта контейнера, а аргументом методу передается объект компонента, который добавляется в контейнер. Координаты, определяющие положение компонента, задаются по отношению к контейнеру. Но какой именно это контейнер, становится понятно, только когда компонент добавляется в контейнер.

Что касается прочих методов, вроде метода setBounds(), то они задают параметры и характеристики того объекта, из которого вызываются. Поэтому, например, командой setBounds(250,250,300,200), в которой метод вызывается из объекта окна (команда размещена в конструкторе), задается положение и размеры окна, а командой lbl.setBounds(10,30,280,50) задаются положение и размеры метки, поскольку в последнем случае метод вызывается из объекта метки. Аналогичная команда есть и для объекта кнопки.

Объект кнопки btn создается командой JButton btn=new JButton("Закрыть окно"). Текстовый аргумент класса JButton определяет название, отображаемое на кнопке. Команда btn.setBounds(50,120,200,30) использована для определения положения кнопки в контейнере (50 пикселов вправо и 120 пикселов вниз от левого верхнего угла контейнера) и определения размеров кнопки (200 пикселов в ширину и 30 пикселов в высоту).

Далее выполняются действия, связанные с обработкой события, состоящего в щелчке на кнопке. В частности, командой MyHandler hnd=new MyHandler() создается объект класса MyHandler (класс описывается далее). Данный объект hnd будем называть *объектом обработчика* или просто *обработчиком*. Командой btn.addActionListener(hnd) объект обработчика регистрируется в кнопке. Наконец, командой add(btn) кнопка добавляется в окно. Команда setVisible(true) которой окно отображается на экране, размещается в конструкторе класса MyFrame самой последней и выполняется после того, как все компоненты созданы и добавлены в окно, а для окна выполнены все нужные настройки.

В главном методе программы создается анонимный объект класса MyFrame, что приводит к отображению окна на экране. Нам осталось выяснить, почему щелчок на кнопке **Закрыть окно** (см. рис. 13.2) приводит к завершению выполнения программы. Для этого обратимся к описанию класса MyHandler. Класс наследует интерфейс ActionListener. В этом интерфейсе есть один абстрактный метод actionPerformed(), который и описывается в классе MyHandler. В теле метода выполняется команда System.exit(0), которой завершается выполнение программы. При щелчке на кнопке **Закрыть окно** выполняется метод actionPerformed() объекта обработчика hnd, который регистрировался в кнопке btn командой btn.addActionListener(hnd) (см. код конструктора класса MyFrame в листинге 13.2).

НА ЗАМЕТКУ

При описании метода actionPerformed() в классе MyHandler аргументом метода указан объект е класса ActionEvent. Это объект события. Он в теле метода actionPerformed() не используется (такая реализация метода, но в принципе может использоваться). Но поскольку речь идет об определении абстрактного метода из интерфейса ActionListener, то метод должен быть описан с такой же сигнатурой, с которой он объявлен в интерфейсе — то есть с аргументом типа ActionEvent.

Таким образом, мы создали класс MyHandler, реализующий интерфейс ActionListener. То, что класс наследует интерфейс ActionListener, означает, что у объектов класса MyHandler есть метод actionPerformed(). При создании окна в конструкторе класса MyFrame создается объект класса MyHandler, и этот объект «связывается» с кнопкой путем регистрации методом addActionListener(). При щелчке на кнопке из этого «зарегистрированного» объекта вызывается метод actionPerformed(). Это, так сказать, формальная сторона вопроса. Чтобы понять глубинный смысл происходящего, целесообразно кратко остановиться на принципах обработки событий.

Принципы обработки событий

Допустим, имеется приложение с графическим интерфейсом. Там есть какие-то графические компоненты, и с этими компонентами может что-то происходить. Назовем то, что в принципе может произойти с компонентом, *событием*. Например, щелчок на кнопке, наведение курсора мыши на компонент, ввод символа в поле ввода, выбор опции и тому подобное. Каждый раз, когда происходит такое событие, автоматические создается объект, который содержит определенную полезную

информацию, касающуюся события. Будем называть этот объект *объектом события*, а иногда (если это не приводит к недоразумениям) просто будем отождествлять его с событием. Объект события создается на основе класса. В Java существует иерархия классов, соответствующих всем возможным событиям, которые могут происходить с графическими компонентами. Такие классы называются *классами событий*. Классы событий содержат в своем названии слово *Event* (что означает *событие*). Для каждого компонента есть набор событий, на которые компонент может в принципе реагировать. В частности, кнопки могут реагировать на события класса ActionEvent. Но то, что компонент может реагировать на событие, еще не означает, что он на него станет реагировать. Чтобы компонент реагировал на событие, необходимо в данном компоненте зарегистрировать обработчик данного события (для разных событий — разные обработчики). Обработчик события — это объект. Но не любой, а тот, что создан на основе класса, реализующего определенный интерфейс. Почему это важно? Просто если класс (на основе которого создается обработчик) реализует определенный интерфейс, то у него точно есть определенные методы (которые объявлены в интерфейсе и описаны в классе обработчика). Таким образом, если в компоненте зарегистрирован обработчик, то у этого обработчика есть определенный набор методов, которые и используются при обработке события.

Если мы знаем, как называется класс события, то по имени класса можно определить имя интерфейса, который должен реализовать класс обработчика, а также название метода, которым обработчик регистрируется в компоненте. Правила такие.

- Чтобы по имени класса события определить имя интерфейса (для реализации в классе обработчика), то в названии класса события следует слово *Event* заменить на слово *Listener*. Например, если речь идет о событии класса ActionEvent, то соответствующий интерфейс называется ActionListener.
- Чтобы узнать имя метода, которым в компоненте регистрируется обработчик, в названии интерфейса (который реализуется в классе обработчике) в начале нужно добавить слово *add* (что означает *добавить*). Например, обработчики событий класса ActionEvent создаются на основе классов, реализующих интерфейс ActionListener, регистрируются в компоненте с помощью метода addActionListener().

Теперь допустим, что в компоненте зарегистрирован обработчик для определенного события. При возникновении события, как отмечалось выше, автоматически создается объект события. Объект события

передается обработчику (зарегистрированному в компоненте) для обработки. Обработчик «знает», какой метод должен быть вызван для обработки события. Этот метод вызывается из объекта обработчика, а аргументом методу передается объект события. Поэтому у методов, которые наследуются в классах обработчиков из интерфейсов, один аргумент, и этот аргумент — объект события, на которое реагирует компонент. Также следует иметь в виду, что такие методы не возвращают результат, и поскольку они объявлены в интерфейсе, то в классе обработчика должны описываться с ключевым словом public.

НА ЗАМЕТКУ

К сожалению, нет простого правила, которое позволило бы определить название методов (а их может быть несколько) из интерфейсов, реализуемых в классе обработчика. В таких случаях придется обращаться к справке по тому или иному интерфейсу. Вместе с тем, при работе со средой NetBeans при описании класса, реализующего интерфейс, автоматически появляется подсказка с информацией о том, какой метод из интерфейса должен быть описан в классе (чтобы класс не был абстрактным).

Теперь, если вернуться к рассмотренному выше примеру, то в интерфейсе ActionListener объявлен всего один абстрактный метод actionPerformed(). Метод не возвращает результат. У метода один аргумент — ссылка на объект класса ActionEvent. Это объект события. Соответственно, если в компоненте регистрируется обработчик для обработки событий класса ActionEvent, то обработчиком должен быть объект класса, реализующего интерфейс ActionListener, регистрируется обработчик методом addActionListener(), а при возникновении с компонентом события данного типа из объекта-обработчика вызывается метод actionPerformed(), и аргументом методу передается ссылка на объект события. Именно такую схему мы реализовали в предыдущем примере. Правда, способ реализации не самый оптимальный. Но нам важно было, прежде всего, понять базовый подход, и для этого мы все этапы показали «в явном виде». Однако существую и иные возможности. Их обсудим далее.

НА ЗАМЕТКУ

Мы будем в основном обсуждать разные способы реализации обработки событий. Но чтобы примеры не были слишком уж однотипными, будут иметь место и небольшие «декоративные» новшества.

Обработчик на основе анонимного класса

Неудобство описанного выше подхода состоит в том, что нам пришлось описывать отдельный класс для обработчика. Обычно такие классы используются единожды. Это намек на то, что имеет смысл создавать объект обработчика на основе анонимного класса. Именно такой подход использован в программе, представленной в листинге 13.4.

Листинг 13.4. Программный код проекта AnonymousHandlerApplication

```
// Импорт классов:
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
// Класс для создания окна:
class MyFrame extends JFrame{
  // Конструктор:
  MyFrame(String name){
    // Вызов конструктора суперкласса:
    super(name);
    // Положение и размеры окна:
    setBounds(250,250,300,200);
    // Окно постоянных размеров:
    setResizable(false);
    // Реакция на щелчок системной пиктограммы:
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // Отключение менеджера компоновки:
    setLayout(null);
    // Создание объекта метки:
    JLabel lbl=new JLabel("Текст синего цвета",JLabel.CENTER);
    // Положение и размеры метки:
    lbl.setBounds(10,30,280,50);
    // Синий цвет для текста метки:
    lbl.setForeground(Color.BLUE);
    // Рамка вокруг метки:
```

```
    lbl.setBorder(BorderFactory.createEtchedBorder());
    // Добавление метки в окно:
    add(lbl);
    // Создание объекта кнопки:
    JButton btn=new JButton("Закрыть окно");
    // Положение и размеры кнопки:
    btn.setBounds(50,120,200,30);
    // Отменяется режим отображения рамки фокуса:
    btn.setFocusPainted(false);
    // Регистрация анонимного обработчика в кнопке:
    btn.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent e){
        System.exit(0);
      }
    });
    // Добавление кнопки в окно:
    add(btn);
    // Отображение окна на экране:
    setVisible(true);
  }
}
// Главный класс:
class AnonymousHandlerDemo{
  // Главный метод:
  public static void main(String[] args){
    // Создание объекта окна:
    new MyFrame("Анонимный обработчик");
  }
}
```

Если запустить программу на выполнение, то появится окно, представленное на рис. 13.3.

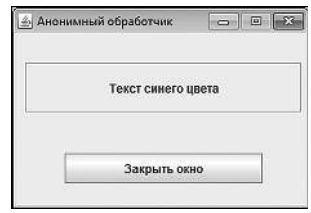


Рис. 13.3. Окно с текстом синего цвета и рамкой вокруг метки

Окно, которое появляется на экране, как и ранее, содержит метку и кнопку. Но теперь текст в метке синего цвета, а вокруг метки отображается «вдавленная» рамка. Также на кнопке не отображается рамка фокуса (штрихованная рамка, которая по умолчанию отображается вокруг текстовой надписи в области кнопки, если последней передан фокус).

Основные изменения в программном коде такие.

- Конструктор класса MyFrame имеет текстовый аргумент, определяющий название окна.
- При создании объекта метки в конструкторе класса MyFrame вторым аргументом конструктору класса JLabel передается статическая константа CENTER класса JLabel. Благодаря этому текст в области метки выравнивается по центру (по умолчанию текст выравнивается по левому краю).
- С помощью команды lbl.setForeground(Color.BLUE) для текста метки устанавливается синий цвет. Для применения цвета из объекта метки вызывается метод setForeground(). Аргументом методу передается статическая константа BLUE класса Color. Для использования класса Color в шапке программы использована инструкция import java.awt.* (класс Color относится к библиотеке AWT).
- Вокруг метки отображается рамка. Рамка отображается командой lbl.setBorder(BorderFactory.createEtchedBorder()). Рамка задается с помощью метода setBorder(). Аргументом методу следует передать объект класса, реализующего интерфейс Border. Такой объект определяет тип рамки. Мы для получения ссылки на объект класса Border воспользовались статическим методом createEtchedBorder() класса BorderFactory. Метод результатом возвращает объект, который соответствует «вдавленной» границе: эффект такой, как если бы границу провели вдавливанием линий.
- Для отмены режима отображения рамки фокуса для кнопки (а такой режим используется по умолчанию), из объекта кнопки btn вызывается метод btn.setFocusPainted() с аргументом false.

Но главное изменение — это, конечно, использование для кнопки анонимного обработчика анонимного класса, реализующего интерфейс ActionListener. Речь идет вот об этом блоке кода в теле конструктора класса MyFrame:

```
btn.addActionListener(new ActionListener(){
  public void actionPerformed(ActionEvent e){
    System.exit(0);
  }
});
```

Благодаря такому подходу отпадает необходимость в описании отдельного класса для обработчика события щелчка на кнопке.

Обработчик на основе лямбда-выражения

Несложно заметить, что интерфейс ActionListener является функциональным (в интерфейсе объявлен только один абстрактный метод). Поэтому процесс создания и регистрации обработчика в кнопке можно еще упростить. В листинге 13.5 представлена версия программы, в которой при реализации обработки событий используется лямбда-выражение.

Листинг 13.5. Программный код проекта LambdaHandlerApplication

```
// Импорт классов:
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
// Класс для создания окна:
class MyFrame extends JFrame{
  // Конструктор:
  MyFrame(String name){
    // Вызов конструктора суперкласса:
    super(name);
    // Положение и размеры окна:
    setBounds(250,250,300,200);
    // Окно постоянных размеров:
```

```
setResizable(false);
// Реакция на щелчок системной пиктограммы:
setDefaultCloseOperation(EXIT_ON_CLOSE);
// Отключение менеджера компоновки:
setLayout(null);
// Объект изображения (для отображения в метке):
ImageIcon img=new ImageIcon("d:/books/pictures/giraffe.png");
// Текстовое значение для отображения в метке:
String txt="<html>Это жираф.<br>Он большой.<br>Он все видит.</html>";
// Создание объекта метки:
JLabel lbl=new JLabel(txt,img,JLabel.LEFT);
// Создание объекта шрифта:
Font F=new Font(Font.MONOSPACED,Font.BOLD,16);
// Применение шрифта к метке:
lbl.setFont(F);
// Положение и размеры метки:
lbl.setBounds(10,30,280,80);
// Переход в режим непрозрачности метки:
lbl.setOpaque(true);
// Светло-серый цвет для фона метки:
lbl.setBackground(Color.LIGHT_GRAY);
// Рамка вокруг метки:
lbl.setBorder(BorderFactory.createEtchedBorder());
// Добавление метки в окно:
add(lbl);
// Создание объекта кнопки:
JButton btn=new JButton("Закрыть окно");
// Положение и размеры кнопки:
btn.setBounds(50,120,200,30);
// Отменяется режим отображения рамки фокуса:
btn.setFocusPainted(false);
// Регистрация в кнопке обработчика
// на основе лямбда-выражения:
```

```
    btn.addActionListener(e->{System.exit(0);});
    // Добавление кнопки в окно:
    add(btn);
    // Отображение окна на экране:
    setVisible(true);
  }
}
// Главный класс:
class LambdaHandlerDemo{
  // Главный метод:
  public static void main(String[] args){
    // Создание объекта окна:
    new MyFrame("Лямбда-выражение");
  }
}
```

Окно, которое отображается при выполнении программы, показано на рис. 13.4.

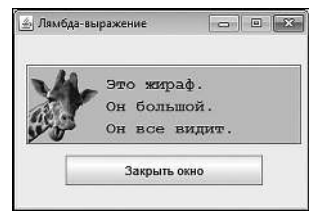


Рис. 13.4. Окно с текстом и изображением, с выделенным фоном для метки

Особенность окна в том, что кроме текста в метке отображается пиктограмма, фон метки выделен серым цветом, а собственно текст в метке отображается в несколько строк, причем шрифт текста отличается от используемого по умолчанию.

Главное изменение в программном коде на этот раз состоит в том, что обработчик для кнопки регистрируется посредством передачи аргументом методу addActionListener() лямбда-выражения e->{System.exit(0);}. Данное выражение определяет код метода actionPerformed() в объекте обработчика. Стоит заметить, что типа аргумента e в лямбда-выражении явно не

указан: он определяется автоматически исходя из сигнатуры метода actionPerformed().

Все прочие новшества имеют отношение к метке. Во-первых, метка, созданная на основе объекта класса JLabel, может содержать не только текст, но и изображение (одновременно с текстом или только изображение). В данном случае в текстовой метке отображается и текст, и изображение. Для изображения командой ImageIcon img=new ImageIcon("d:/books/pictures/giraffe.png"). Данной командой на основе графического файла giraffe.png создается объект класса ImageIcon, который затем передается вторым аргументом конструктору класса JLabel. Третьим аргументом конструктору передается статическая константа LEFT класса JLabel, определяющая способ выравнивания содержимого метки по левому краю. Первым аргументом конструктору класса JLabel передается текстовая переменная txt, значение которой содержит гипертекстовые дескрипторы.

ДЕТАЛИ

Язык гипертекстовой разметки HTML (сокращение от *HyperText Markup Language*) используется при создании веб-страниц. Соответствующий текст содержит, кроме собственно текста, специальные инструкции, которые называют дескрипторами. Дескрипторы используются браузером для определения способа отображения блоков документа. Дескрипторы заключаются в угловые скобки. Признаком начала HTML-кода является дескриптор <html>. Дескриптор </html> означает завершение HTML-кода. Дескриптор
 является инструкцией перехода к новой строке.

Если текст, предназначенный для отображения в текстовой метке, содержит HTML-дескрипторы, то такой текст должен начинаться с дескриптора <html> и заканчиваться дескриптором </html>. В данном случае мы в текстовой переменной txt, кроме начального и конечного дескрипторов, используем инструкции
. При отображении текста в текстовой метке там, где находится инструкция
, выполняется переход к новой строке.

Еще один новый момент — определение шрифта для метки. Шрифт реализуется через объект F класса Font из библиотеки AWT. Конструктору класса Font передаются три аргумента. Первым аргументом передана статическая текстовая константа MONOSPACED класса Font, определяющая тип шрифта, второй аргумент задает стиль шрифта (указанная вторым аргументом константа BOLD класса Font соответствует жирному стилю

шрифта), а третий целочисленный аргумент задает размер шрифта. После того, как создан объект F шрифта, командой lbl.setFont(F) шрифт применяется к текстовой метке.

Наконец, для текстовой метки указан фоновый цвет. Делается это вызовом из объекта метки метода setBackground() с аргументом Color.LIGHT_GRAY, означающим применение светло-серого цвета для фона метки. Но этого на самом деле мало. Дело в том, что по умолчанию для метки (как и для многих других компонентов) используется режим прозрачности, при котором некоторые области внутри компонента не рисуются и являются как бы прозрачными. Поэтому командой lbl.setOpaque(true) делаем метку непрозрачной.

Обработчик на основе объекта окна

Еще один полезный подход состоит в том, что интерфейс, который следует реализовать в классе обработчика, реализуется в классе окна. Таким образом, объект окна может использоваться в качестве обработчика компонента в этом окне. Как указанный подход реализуется на практике, показано в листинге 13.6 на примере создания окна с кнопкой, щелчок на которой приводит к закрытию окна и завершению выполнения программы. В этом примере мы познакомимся с новым компонентом — *панелью*. Панель представляет собой прямоугольную область и является контейнером, в который могут добавляться другие компоненты. Реализуется панель через объект класса JPanel. А теперь рассмотрим представленный ниже код.

Листинг 13.6. Программный код проекта WindowAsHandlerApplication

```
// Импорт классов:
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
// Класс для создания окна
// реализует интерфейс ActionListener:
class MyFrame extends JFrame implements ActionListener{
  // Описание метода из интерфейса ActionListener:
  public void actionPerformed(ActionEvent e){
    System.exit(0);
```

```
}
// Конструктор:
MyFrame(String name){
  // Вызов конструктора суперкласса:
  super(name);
  // Положение и размеры окна:
  setBounds(250,250,300,200);
  // Окно постоянных размеров:
  setResizable(false);
  // Реакция на щелчок системной пиктограммы:
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  // Отключение менеджера компоновки для окна:
  setLayout(null);
  // Создание объекта панели:
  JPanel pnl=new JPanel();
  // Положение и размеры панели:
  pnl.setBounds(5,5,285,110);
  // Рамка вокруг панели:
  pnl.setBorder(BorderFactory.createEtchedBorder());
  // Отключение менеджера компоновки для панели:
  pnl.setLayout(null);
  // Объект изображения (для отображения в метке):
  ImageIcon img=new ImageIcon("d:/books/pictures/giraffe.png");
  // Текстовое значение для отображения в метке:
  String txt="<html>Это жираф.<br>Он большой.<br>Он все видит.</html>";
  // Создание объекта метки с изображением:
  JLabel imgLbl=new JLabel(img);
  // Положение и размеры метки:
  imgLbl.setBounds(10,10,90,90);
  // Рамка вокруг метки:
  imgLbl.setBorder(BorderFactory.createEtchedBorder());
  // Создание объекта для метки с текстом:
  JLabel txtLbl=new JLabel(txt,JLabel.CENTER);
```

```
    // Положение и размеры метки:
    txtLbl.setBounds(110,10,165,90);
    // Рамка вокруг метки:
    txtLbl.setBorder(BorderFactory.createEtchedBorder());
    // Создание объекта кнопки:
    JButton btn=new JButton("Закрыть окно");
    // Положение и размеры кнопки:
    btn.setBounds(50,120,200,30);
    // Отменяется режим отображения рамки фокуса:
    btn.setFocusPainted(false);
    // Регистрация в кнопке обработчиком
    // объекта окна:
    btn.addActionListener(this);
    // Создание объекта шрифта:
    Font F=new Font(
      // Название шрифта как у кнопки:
      btn.getFont().getName(),
      // Стиль — жирный курсив:
      Font.BOLD|Font.ITALIC,
      // Размер шрифта на 2 больше чем у кнопки:
      btn.getFont().getSize()+2);
    // Применение шрифта к метке с текстом:
    txtLbl.setFont(F);
    // Добавление меток на панель:
    pnl.add(imgLbl);
    pnl.add(txtLbl);
    // Добавление панели в окно:
    add(pnl);
    // Добавление кнопки в окно:
    add(btn);
    // Отображение окна на экране:
    setVisible(true);
  }
```

```
}
// Главный класс:
class WindowAsHandlerDemo{
  // Главный метод:
  public static void main(String[] args){
    // Создание объекта окна:
    new MyFrame("Обработчик — объект окна");
  }
}
```

Если запустить программу на выполнение, то появится окно, представленное на рис. 13.5.

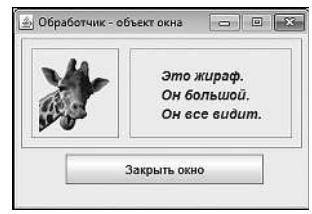


Рис. 13.5. Окно с кнопкой и панелью, на которой размещены две метки (с изображением и текстом)

Схема, использованная в данном случае, такая: в конструкторе класса MyFrame создается панель, на которую добавляются две метки: одна с изображением, а другая с текстом. Затем панель добавляется в окно. Также в окно добавляется кнопка.

Для удобства восприятия и панель, и две метки выделяются с помощью вдавленной рамки, так что достаточно легко представить, как именно размещаются компоненты. Теперь обратимся к анализу программного кода примера.

В первую очередь стоит заметить, что класс MyFrame, помимо наследования класса JFrame, еще и реализует интерфейс ActionListener. Поэтому теперь в классе MyFrame кроме конструктора описан еще и метод actionPerformed(). В конструкторе класса командой btn.addActionListener(this) обработчиком в кнопке регистрируется объект окна. Это означает, что при щелчке на кнопке будет вызываться метод actionPerformed() из объекта окна, то есть тот метод, что описан в классе MyFrame.

Достойны внимания и некоторые иные блоки кода в конструкторе класса MyFrame. Так, объект для панели pnl создается командой JPanel pnl=new JPanel(). Главное назначение панели — быть контейнером для других компонентов. Но некоторые параметры панели все же следует определить. Так, командой pnl.setBounds(5,5,285,110) задается положение и размеры панели. Поскольку панель добавляется в окно, то положение панели задается по отношению к окну. В данном случае мы создаем панель шириной 285 пикселов и высотой 110 пикселов. Левый верхний угол панели будет смещен на 5 пикселов влево и на 5 пикселов вниз по отношению к левому верхнему углу рабочей области окна.

НА ЗАМЕТКУ

Мы оставляем без внимания выбор конкретных числовых значений для размеров и положения компонентов, хотя это и важный вопрос. Хочется верить, что читатель в случае необходимости сможет восстановить соответствующие несложные арифметические расчеты. Но вообще хочется заметить, что более надежный способ подразумевает использование относительных, а не абсолютных характеристик для линейных размеров: например, разумно задать размеры окна через целочисленные переменные и затем вычислять размеры и положение прочих компонентов на основе этих параметров. Но это увеличивает объем программного кода, что нежелательно, учитывая учебный характер рассматриваемых примеров.

Также стоит заметить, что используемое в программе изображение имеет размеры 75 пикселов в ширину и 75 пикселов в высоту.

Рамка вокруг панели устанавливается командой pnl.setBorder(BorderFactory.createEtchedBorder()). Также с помощью команды pnl.setLayout(null) отключается менеджер компоновки для панели.

НА ЗАМЕТКУ

Следует учесть, что менеджеры компоновки есть у всех контейнеров, в том числе и у панели. Их того, что мы отключили менеджер компоновки для окна, не следует, что он отключился для панели. У панели свой менеджер компоновки и его отключать следует отдельно.

Командой JLabel imgLbl=new JLabel(img) создается метка с изображением, а командой JLabel txtLbl=new JLabel(txt,JLabel.CENTER) создается метка с текстом (текст выравнивается по центру). Для каждой метки устанавливается вдавленная рамка, и задаются размеры и положение в контейнере.

НА ЗАМЕТКУ

Важно понимать, что положение меток, поскольку они добавляются на панель, определяются по отношению к левому верхнему углу панели, а не окна, как это было ранее. А вот кнопка добавляется непосредственно в окно, поэтому ее положение задается относительно левого верхнего угла рабочей области окна.

Еще один «прием» связан с определением шрифта для текстовой метки (метки с текстом). Шрифт определяется на основе шрифта, используемого по умолчанию для кнопки (и поэтому команды определения шрифта размещаются в программном коде после того, как создана кнопка). Объект F шрифта создается на основе класса Font. Аргументы конструктору класса Font передаются такие.

- Первым аргументом передана инструкция btn.getFont().getName(). Это логическое название шрифта, используемое для отображения названия кнопки. Следует учесть, что в результате вызова метод getFont() из объекта кнопки btn возвращается ссылка на объект класса Font, описывающего шрифт, применяемый к кнопке. Из этого объекта вызывается метод getName(), который возвращает текстовое значение с логическим названием шрифта. И это значение передается первым аргументом конструктору класса Font при создании нового объекта шрифта.
- Вторым аргументом конструктора указано выражение Font.BOLD|Font.ITALIC означающее, что одновременно используется и жирный, и курсивный стили (в результате получается жирный курсивный стиль).
- Третьим аргументом конструктору при создании объекта шрифта передано выражение btn.getFont().getSize()+2, в котором к выражению btn.getFont().getSize() прибавляется значение 2. Результат выражения btn.getFont().getSize() представляет собой результат вызова метода getSize() из объекта шрифта, используемого для кнопки. Ссылка на последний дается выражением btn.getFont(). Метод getSize(), вызванный из объекта шрифта, возвращает результатом размер шрифта. Поэтому размер шрифта для метки на 2 больше размера шрифта для кнопки.

После определения объекта шрифта F данный шрифт с помощью команды txtLbl.setFont(F) применяется к метке с текстом. Далее командами pnl.add(imgLbl) и pnl.add(txtLbl) метки добавляются на панель, после чего командой add(pnl) панель добавляется в окно. Кнопка также добавляется в окно с помощью команды add(btn).

Создание класса для кнопки

Кнопка, которую мы стабильно добавляем в окно, имеет одну особенность: при щелчке на ней завершается выполнение программы. Важно здесь то, что при обработке события щечка на кнопке никакие иные компоненты, кроме собственно кнопки, не задействованы. В этом смысле кнопка достаточно автономна и универсальна. В рассматриваемой далее программе мы вместо того, чтобы создавать и настраивать кнопку в конструкторе окна, опишем отдельный класс для кнопки, которая предназначена для завершения работы программы. Также мы опишем отдельный класс для панели. Рассмотрим программу в листинге 13.7.

Листинг 13.7. Программный код проекта ButtonAndHandlerApplication

```
// Импорт классов:
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
// Класс кнопки:
class MyButton extends JButton implements ActionListener{
  // Описание метода из интерфейса ActionPerformed:
  public void actionPerformed(ActionEvent e){
    System.exit(0);
  }
  // Конструктор:
  MyButton(int x,int y,int w,int h){
    // Вызов конструктора суперкласса:
    super("Закрыть окно");
    // Положение и размеры кнопки:
    setBounds(x,y,w,h);
    // Отмена отображения рамки фокуса:
    setFocusPainted(false);
    // Регистрация обработчика в кнопке:
    addActionListener(this);
  }
}
// Класс панели:
```

```
class MyPanel extends JPanel{
  // Конструктор:
  MyPanel(String txt,ImageIcon img){
    // Вызов конструктора суперкласса:
    super();
    // Положение и размеры панели:
    setBounds(5,5,285,110);
    // Рамка вокруг панели:
    setBorder(BorderFactory.createEtchedBorder());
    // Отключение менеджера компоновки:
    setLayout(null);
    // Создание объекта для метки с изображением:
    JLabel imgLbl=new JLabel(img);
    // Положение и размеры метки:
    imgLbl.setBounds(10,10,90,90);
    // Рамка вокруг метки:
    imgLbl.setBorder(BorderFactory.createEtchedBorder());
    // Создание объекта для метки с текстом:
    JLabel txtLbl=new JLabel(txt,JLabel.CENTER);
    // Положение и размеры метки:
    txtLbl.setBounds(110,10,165,90);
    // Рамка вокруг метки:
    txtLbl.setBorder(BorderFactory.createEtchedBorder());
    // Добавление меток на панель:
    add(txtLbl);
    add(imgLbl);
  }
}
// Класс для окна:
class MyFrame extends JFrame{
  // Конструктор:
  MyFrame(String name,String txt,ImageIcon img){
    // Вызов конструктора суперкласса:
```

```
    super(name);
    // Положение и размеры окна:
    setBounds(250,250,300,200);
    // Окно постоянных размеров:
    setResizable(false);
    // Реакция на щелчок системной пиктограммы:
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // Отключение менеджера компоновки для окна:
    setLayout(null);
    // Создание объекта панели:
    MyPanel pnl=new MyPanel(txt,img);
    // Создание объекта кнопки:
    MyButton btn=new MyButton(50,120,200,30);
    // Добавление панели в окно:
    add(pnl);
    // Добавление кнопки в окно:
    add(btn);
    // Отображение окна на экране:
    setVisible(true);
  }
}
// Главный класс:
class ButtonAndHandlerDemo{
  // Главный метод:
  public static void main(String[] args){
    // Объект для изображения:
    ImageIcon img=new ImageIcon("d:/books/pictures/giraffe.png");
    // Текстовое значение:
    String txt="<html>Это жираф.<br>Он большой.<br>Он все видит.</html>";
    // Создание объекта окна:
    new MyFrame("Обработчик — объект кнопки",txt,img);
  }
}
```

При выполнении программы отображается практически такое же окно, как и в предыдущем случае (см. рис. 13.5) — за исключением разве что шрифта, использованного при отображении текста в области текстовой метки. Окно показано на рис. 13.6.

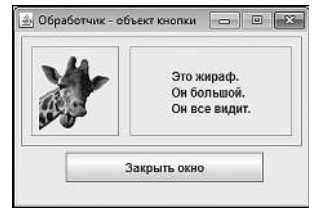


Рис. 13.6. При выполнении программы отображается окно с изображением и текстом

Но код, выполнение которого приводит к такому результату, немного «реорганизован». В частности, в программе описан класс MyButton, который наследует класс JButton и реализует интерфейс ActionListener. В конструкторе класса MyButton командой addActionListener(this) обработчиком для кнопки регистрируется сам объект кнопки. Поэтому при щелчке на кнопке, куда бы она ни была добавлена, выполняется метод actionPerformed() объекта кнопки и, как следствие, программа прекращает выполнение (в силу определения метода).

При создании объекта класса MyButton конструктору передаются четыре целочисленных аргумента, определяющие положение и размеры кнопки.

Также в программе описан класс MyPanel, наследующий класс JPanel. Объект класса MyPanel соответствует панели с двумя метками: с текстом и изображением.

Ссылки на текст и изображение передаются аргументами конструктору класса MyPanel. В теле конструктора класса создаются метки, выполняется настройка их параметров (и параметров панели), метки добавляются на панель.

В конструкторе класса MyFrame, предназначенного для создания объекта окна, выполняются настройки параметров окна, создается объект класса MyPanel (панель с текстом и изображением) и объект класса MyButton (кнопка), после чего панель и копка добавляются в окно. У конструктора класса MyFrame, кроме первого текстового аргумента, определяющего название окна, теперь есть еще два аргумента: текст для текстовой метки и ссылка

на изображение для метки с изображением. Эти аргументы передаются в конструктор класса MyPanel при создании в теле конструктора класса MyFrame объекта панели.

Как следствие, в главном методе программы создается объект для изображения и определяется текстовое значение для отображения в области окна, а при создании объекта класса MyFrame конструктору передается три аргумента.

НА ЗАМЕТКУ

Хочется подчеркнуть два обстоятельства. Во-первых, кнопка, реализованная через объект класса MyButton, уже имеет зарегистрированный обработчик для события, состоящего в щелчке на кнопке. Во-вторых, мы описываем классы для создания и окна, и панели фиксированного размера с фиксированным размещением компонентов. Другими словами, степени свободы при использовании описанных в программе классов ограничиваются использованием разного текста и разных изображений в области панели. Размеры компонентов фиксированные, и если изменить размер одного из них, то тогда придется вручную пересчитывать размеры прочих компонентов. Это не очень хорошо. Но зато такой подход позволяет сократить объем программного кода, что в данном случае намного важнее его универсальности.

Резюме

— Независимые умы никогда не боялись банальностей.
— Ты-то тут при чем?!
— Грубо, но правда — я ни при чем!

Из к/ф «Покровские ворота»

- Создание приложений с графическим интерфейсом подразумевает создание собственно графических компонентов и реализацию обработки событий.
- Для создания графических компонентов может использоваться библиотека Swing. Для создания того или иного компонента на основе класса библиотеки создается объект компонента. С помощью методов объекта и ряда других утилит выполняются настройки параметров компонента.

- Каждый компонент может реагировать на определенные события. При возникновении события для него создается объект. Объекты событий создаются на основе специальных классов событий. Для определения реакции компонента на событие определенного типа в компоненте регистрируется обработчик события. Это объект, созданный на основе класса, реализующего определенный интерфейс (связанный с событием). Если в компоненте зарегистрирован обработчик события, то при возникновении этого события объект события передается в обработчик для обработки.

Глава 14
ОБРАБОТКА СОБЫТИЙ

Ну и что вы скажете обо всем этом, Ватсон?

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

В этой главе мы продолжим знакомство с методами создания приложений с графическим интерфейсом. Предметом нашего рассмотрения станут основные классы событий, которые наиболее часто используются при реализации графического интерфейса. Также мы продолжим знакомство с классами, через которые реализуются компоненты графического интерфейса. Если точнее, то далее состоится «общее» знакомство — мы просто узнаем, какие в принципе есть компоненты. Относительно детальное знакомство произойдет с одним компонентом (текстовым полем), и то в контексте обработки событий. Прочие компоненты интерфейса обсуждаются в следующей главе книги.

Классы компонентов и событий

Вот и выходит, что все мироздание суть игра моего ума. А если вы со мной согласитесь — то и вашего тоже!

Из к/ф «Формула любви»

Прежде чем продолжить рассмотрение методов создания приложений с графическим интерфейсом, сделаем небольшой обзор основных классов, которые используются при создании компонентов и реализации обработки событий.

Классы графических компонентов

С некоторыми классами библиотеки Swing, используемыми для создания графических компонентов, мы уже познакомились. Среди них есть

класс JFrame (класс для создания объекта окна), JButton (класс для создания объекта кнопки), JLabel (класс для создания метки) и JPanel (класс для создания панели). Понятно, что существуют и иные классы, предназначенные для создания графических компонентов. В табл. 14.1 представлены некоторые классы (в том числе и рассмотренные ранее) из библиотеки Swing, которые используются для реализации графических компонентов.

Табл. 14.1. Классы для создания графических компонентов

Класс компонента	Краткое описание класса
JButton	Класс предназначен для создания объектов стандартных кнопок
JCheckBox	С помощью класса создается объект для такого элемента, как опция (элемент, для которого можно установить или отменить флажок)
JCheckBoxMenuItem	На основе класса создается объект для опционной команды меню (команда меню, для которого можно установить или отменить флажок)
JComboBox	Класс предназначен для создания объекта раскрывающегося списка (отображается один элемент, а при щелчке на специальной пиктограмме раскрывается список, в котором можно выбрать новое значение)
JEditorPane	Класс предназначен для реализации текстовой области, данные в которой могут отображаться с использованием различных шрифтов
JFormattedTextField	Подкласс класса JTextField. Данный класс предназначен для создания текстового поля
JFrame	С помощью класса создается объект окна
JLabel	Класс предназначен для создания меток
JList	С помощью данного класса создаются списки выбора: набор явно отображаемых элементов, которые можно выбирать
JMenu	Класс предназначен для создания пунктов (или подпунктов) в главном меню
JMenuBar	С помощью объекта данного класса создается панель меню
JMenuItem	На основе класса создаются объекты для элементов меню (команды меню)
JPanel	Класс предназначен для создания панелей (контейнер, который может содержать другие компоненты)
JPasswordField	Класс позволяет создать объект для поля, в которое вводится пароль (поле ввода пароля). Класс является подклассом класса JFormattedTextField
JPopupMenu	Класс используется при создании контекстного меню
JProgressBar	Класс предназначен для создания индикатора процесса (компонент, показывающий степень завершения некоторого процесса)

Класс компонента	Краткое описание класса
JRadioButton	С помощью объектов класса создаются переключателя. Переключатели обычно объединяются в группы. Принципиальное отличие переключателя от опции состоит в том, что в группе может быть установлен (выбран) один и только один переключатель. При работе с переключателями используют класс ButtonGroup, на основе которого создается объект группы переключателей
JRadioButtonMenuItem	С помощью объектов данного класса реализуются переключатели, являющиеся элементами (командами) меню
JScrollBar	С помощью объекта данного класса создаются полосы прокрутки
JScrollPane	Класс предназначен для реализации панели с полосами прокрутки
JSeparator	С помощью объекта класса в меню создается разделитель (линия) — декоративный элемент, разделяющий команды в пункте меню
JSlider	Класс предназначен для создания линейного регулятора (слайдера)
JSpinner	Посредством объекта класса реализуется спиннер. Спиннер представляет собой поле со значением и двумя пиктограммами, щелчок на которых приводит к изменению значения в поле спиннера
JSplitPane	Класс предназначен для реализации панели, разделенной на две части
JTabbedPane	С помощью класса создается панель с вкладками
JTextArea	Класс позволяет создать текстовую область
JTextField	Класс используется при создании поля для ввода текста
JTextPane	Класс предназначен для реализации текстовой области и имеет расширенные возможности в отношении отображения данных. Является подклассом класса JEditorPane
JToggleButton	Класс предназначен для создания опционной кнопки (кнопка, которая может находиться в двух состояниях — нажатом и не нажатом)
JToolBar	Через объект класса реализуется панель инструментов
JToolTip	Класс полезен при создании контекстной подсказки для компонентов графического интерфейса

Каждый класс из приведенной табл. 14.1 размещен в пакете javax.swing. Большинство из перечисленных классов (собственно все, за исключением класса JFrame) прямо или опосредованно являются наследниками класса JComponent.

НА ЗАМЕТКУ

Компоненты, классы которых (из библиотеки Swing) наследуют класс JComponent, называются *легкими* — в отличие от компонентов, реализуемых через классы библиотеки AWT, которые называются *тяжелыми*.

Также стоит заметить, что часто компоненты графического интерфейса называют по имени класса, на основе которого создается объект для компонента.

Более детально методы работы с классами компонентов мы рассмотрим немного позже. А сейчас сделаем небольшой обзор классов, описывающих события.

Классы событий

В Java достаточно много классов событий, большинство из них собрано в библиотеке AWT в пакете java.awt.event. Также есть немного классов событий в библиотеке Swing в пакете javax.swing.event. В основном это классы событий, связанных с компонентами, специфичными именно для библиотеки (таким, например, как спиннер). Наиболее актуальные классы событий из пакета java.awt.event представлены и кратко описаны в табл. 14.2.

Табл. 14.2. Некоторые классы событий из пакета java.awt.event

Класс события	Описание события
ActionEvent	Класс для события, являющегося характерными для данного компонента. Например, характерным событием для кнопки является щелчок на ней
AdjustmentEvent	Класс для события, которое связано с изменением состояния компонента со средствами прокрутки — такого, например, как полоса прокрутки
ComponentEvent	Класс события, связанного с изменением положения, размеров или иных параметров компонента
ContainerEvent	Класс события, связанного с изменением содержимого контейнера
FocusEvent	Класс события, состоящего в том, что компонент получил или потерял фокус
InputEvent	Базовый класс для событий, связанных с вводом данных
ItemEvent	Класс для события, связанного с изменением состояния компонента (в связи с его выбором или отменой выбора)
KeyEvent	Класс для события, связанного с нажатием клавиш на клавиатуре (при активном компоненте)
MouseEvent	Класс для события, связанного с действиями мышью в области компонента
MouseWheelEvent	Класс события, связанного с вращением колеса мыши в области компонента
TextEvent	Класс для события, связанного с изменением текста в компоненте
WindowEvent	Класс события, связанного с изменением статуса окна (таким, например, как открытие, закрытие, сворачивание или разворачивание)

Некоторые классы событий представлены, как отмечалось выше, в библиотеке Swing. В табл. 14.3 представлены и кратко описаны несколько классов событий из пакета javax.swing.event.

Табл. 14.3. Некоторые классы событий из пакета javax.swing.event

Класс события	Описание события
ChangeEvent	Класс для события, связанного с изменением состояния компонента
MenuEvent	Класс для события, связанного операциями с меню
PopupMenuEvent	Класс для события, связанного операциями с контекстным меню

Классы событий из библиотеки Swing являются наследниками (прямыми или опосредованными) класса EventObject из пакета java.util (при этом сами классы событий находятся в пакете javax.swing.event). Классы событий из библиотеки AWT наследуют (прямо или через посредников) класс AWTEvent из пакета java.awt (а сами классы событий находятся в пакете java.awt.event). Класс AWTEvent является, в свою очередь, подклассом класса EventObject. Класс EventObject является прямым наследником (подклассом) суперкласса Object. Эти обстоятельства определяют некоторые общие свойства классов событий.

Более детально способы методы работы с классами для графических компонентов интерфейса и классами событий мы рассмотрим далее на конкретных (в основном, несложных) примерах.

Использование текстового поля

Странный способ украшать дом монограммой королевы.

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Одним из относительно простых и исключительно полезных графических компонентов является текстовое поле. Компонент реализуется через объект класса JTextField. Рассмотрим несколько примеров, в которых, кроме прочего, используется текстовое поле.

Считывание значения поля

В представленном в листинге 14.1 примере реализуется очень простая схема. Создается окно, у которого есть текстовая метка, поле ввода

и две кнопки: кнопка **Применить** и кнопка **Закрыть**. При щелчке на кнопке **Применить** текст, который пользователь ввел в текстовое поле, отображается в метке. Если щелкнуть на кнопке **Закрыть**, окно закрывается. Теперь перейдем к рассмотрению программного кода примера.

Листинг 14.1. Программный код проекта TextFieldApplication

```
// Импорт классов:
import javax.swing.*;
import javax.swing.border.*;
import java.awt.event.*;
// Класс для создания окна:
class MyFrame extends JFrame{
  // Метка:
  private JLabel L;
  // Текстовое поле:
  private JTextField T;
  // Конструктор:
  MyFrame(){
    // Вызов конструктора суперкласса:
    super("Окно с текстовым полем");
    // Значения для размеров окна:
    int w=300,h=160;
    // Положение и размеры окна:
    setBounds(250,250,w,h);
    // Реакция на щелчок системной пиктограммы:
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // Окно постоянных размеров:
    setResizable(false);
    // Отключение менеджера компоновки:
    setLayout(null);
    // Создание метки:
    L=new JLabel();
    // Положение и размеры метки:
    L.setBounds(10,10,w-25,30);
```

```
    // Выделение метки с помощью эффекта "вдавливания":
    L.setBorder(BorderFactory.createBevelBorder(BevelBorder.LOWERED));
    // Создание текстового поля:
    T=new JTextField();
    // Положение и размеры поля:
    T.setBounds(L.getX(),50,L.getWidth(),30);
    // Ширина кнопок:
    int bw=(T.getWidth()-20)/2;
    // Создание первой кнопки:
    JButton appB=new JButton("Применить");
    // Положение и размеры первой кнопки:
    appB.setBounds(T.getX(),90,bw,30);
    // Отмена режима отображения фокуса
    // для первой кнопки:
    appB.setFocusPainted(false);
    // Обработчик события для первой кнопки:
    appB.addActionListener(e->L.setText(T.getText()));
    // Создание второй кнопки:
    JButton extB=new JButton("Закрыть");
    // Положение и размеры второй кнопки:
    extB.setBounds(appB.getX()+appB.getWidth()+20,appB.getY(),appB.getWidth(),appB.
getHeight());
    // Отмена режима отображения фокуса
    // для второй кнопки:
    extB.setFocusPainted(false);
    // Обработчик события для второй кнопки:
    extB.addActionListener(e->System.exit(0));
    // Добавление в окно метки:
    add(L);
    // Добавление поля в окно:
    add(T);
    // Добавление первой кнопки в окно:
    add(appB);
    // Добавление второй кнопки в окно:
```

```
    add(extB);
    // Отображение окна:
    setVisible(true);
  }
}
// Главный класс:
class TextFieldDemo{
  // Главный метод:
  public static void main(String[] args){
    // Создание окна:
    new MyFrame();
  }
}
```

При запуске программы на выполнение отображается окно, представленное на рис. 14.1.

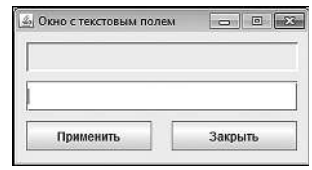


Рис. 14.1. При выполнении программы отображается окно с меткой, текстовым полем и двумя кнопками

Допустим, в текстовое поле вводится текст, как показано на рис. 14.2.

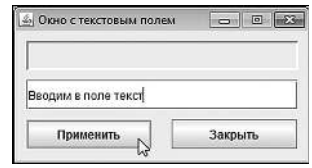


Рис. 14.2. После ввода в поле текста щелкаем кнопку **Применить**

После щелчка на кнопке **Применить** данный текст отображается в текстовой метке. Результат такой операции проиллюстрирован на рис. 14.3.

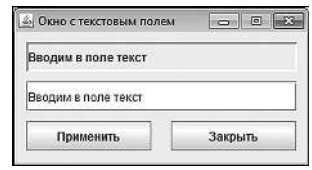


Рис. 14.3. После щелчка на кнопке **Применить** текст из поля отображается в метке

Операцию можно повторить: на рис. 14.4 показано, как в поле вводится новый текст, но при этом в текстовой метке содержится значение, которое было ранее.

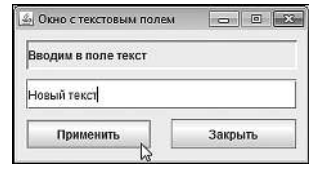


Рис. 14.4. Ввод нового текста в текстовом поле

После щелчка на кнопке **Применить** уже новое значение в текстовом поле отображается и в текстовой метке, как показано на рис. 14.5.

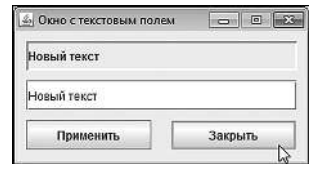


Рис. 14.5. В меткое отображается новое значение из текстового поля

После щелчка на кнопке **Закрыть** окно закрывается, а программа прекращает выполнение.

Проанализируем основные моменты в программном коде примера. Основу программы составляет класс `MyFrame`, наследующий класс `JFrame`. В классе объявлены два закрытых поля: ссылка `L` на объект класса `JLabel` (метка) и ссылка `T` на объект класса `JTextField` (текстовое поле). Все настройки и важные операции выполняются в конструкторе класса `MyFrame`.

Особенности кода конструктора класса `MyFrame` можно разделить на две группы: «тактические» и «стратегические». К «стратегическим»

нововведениям можно отнести использование нового для нас графического компонента — текстового поля, которое реализуется через объект класса JTextField. К «тактическим» новшествам относится использование целочисленных переменных w и h для значений ширины и высоты окна. Данные переменные используются в некоторых командах для определения размеров прочих компонентов (так код выглядит как минимум более информативным). Также при определении координат и размеров компонентов окна использованы методы getX() (горизонтальная координата компонента), getY() (вертикальная координата компонента), getWidth() и getHeight() (соответственно ширина и высота компонента). Методы вызываются из объекта компонента, у которого считывается параметр.

Объект для метки создается командой L=new JLabel(). Аргумент конструктору класса JLabel не передается, поэтому метка получится пустой (но для нее можно будет задать текст потом). Положение и размеры метки определяются командой L.setBounds(10,10,w-25,30). Также мы применяем к метке эффект «вдавливания», для чего командой L.setBorder(BorderFactory.createBevelBorder(BevelBorder.LOWERED)) формально определяем рамку для метки.

ДЕТАЛИ

Для определения рамки из объекта метки вызывается метод setBorder(). Аргументом методу передается ссылка на объект, определяющий выделение метки с помощью «вдавленной» границы. Для получения ссылки на данный объект из класса BorderFactory вызывается статический метод createBevelBorder(). Аргументом методу передается статическая константа LOWERED класса BevelBorder. Именно константа LOWERED определяет эффект «вдавливания». Если вместо статической константы LOWERED аргументом методу createBevelBorder() передать статическую константу RAISED из класса BevelBorder, то метка будет отображаться так, как если бы она выступала (поднималась) над областью окна. Для использования класса BevelBorder в начале программы добавлена инструкция import javax.swing.border.*.

Объект для текстового поля создаем командой T=new JTextField(). Положение и размеры поля задаются командой T.setBounds(L.getX(),50,L.getWidth(),30). Здесь мы горизонтальную координату поля задаем равной горизонтальной координате метки (инструкция L.getX()). Ширина поля устанавливается равной ширине метки (инструкция L.getWidth()). Для вычисления ширины кнопок мы объявляем целочисленную переменную bw и присваиваем ей значением величину (T.getWidth()-20)/2 (ширина кнопки — это уменьшенная на 20 пикселов и деленная на два ширина текстового поля). Первая

кнопка создается командой JButton appB=new JButton("Применить"). Ее положение и размеры задаем командой appB.setBounds(T.getX(),90,bw,30). Для кнопки командой appB.setFocusPainted(false) отменяем режима отображения фокуса. Наконец, с помощью инструкции appB.addActionListener(e->L.setText(T.getText())) регистрируем в кнопке обработчик щелчка на кнопке. В данном случае аргументом методу addActionListener() передается лямбда-выражение e->L.setText(T.getText()), определяющее метод actionPerformed() в объекте-обработчике. В соответствии с кодом лямбда-выражения при щелчке на кнопке метке присваивается новый текст. Для этого из объекта метки L вызывается метод setText(). В общем случае аргументом методу передается текст для отображения в метке. В данной ситуации аргументом методу setText() передается выражение T.getText(), результатом которого является текст, считанный из текстового поля.

Вторая кнопка создается аналогичным образом: собственно объект кнопки создаем командой JButton extB=new JButton("Закрыть"). При вычислении положения и размеров второй кнопки горизонтальная координата вычисляется инструкцией appB.getX()+appB.getWidth()+20: к горизонтальной координате первой appB.getX() кнопки прибавляется ширина кнопки appB.getWidth() и еще 20 пикселов. Вертикальная координата второй кнопки такая же, как и у первой кнопки (инструкция appB.getY()). Размеры второй кнопки определяются через размеры первой кнопки (ширина вычисляется как appB.getWidth() и высота вычисляется выражением appB.getHeight()).

Для регистрации обработчика во второй кнопке использована команда extB.addActionListener(e->System.exit(0)). Переданное аргументом методу addActionListener() лямбда-выражение e->System.exit(0) означает, что при щелчке на кнопке программа завершает выполнение.

После добавления метки, поля и кнопок в окно (команды add(L), add(T), add(appB) и add(extB) соответственно), с помощью команды setVisible(true) окно отображается на экране. Поэтому при создании в главном методе программы командой new MyFrame() объекта окна, окно автоматически появляется на экране.

Использование общего обработчика

В рассмотренном выше примере мы использовали два обработчика — по одному для каждой из кнопок. Вместе с тем, у компонента может быть больше чем один обработчик (если компонент реагирует на несколько событий), а также один обработчик может использоваться сразу

в нескольких компонентах. Именно такой случай и рассмотрим: для обеих кнопок используем один обработчик. Но поскольку для каждой из кнопок обработка должна выполняться по-разному, то при возникновении события необходимо «идентифицировать» компонент, на котором произошло событие. Такую «идентификацию» можно провести на основе объекта события, который передается методу actionPerformed(). Так что в рассматриваемом далее примере мы, кроме прочего, еще и увидим, какова может быть польза от объекта события.

Мы рассмотрим, как и в предыдущем примере, окно с полем ввода, текстовой меткой и двумя кнопками. Но теперь метка тоже будет реагировать на события: при наведении курсора мыши на область метки она будет отображаться с применением эффекта «поднятия» (область метки отображается так, как будто она находится над областью окна), а текст в метке (пока над областью метки находится курсор) выравнивается по правому краю. Если курсор мыши убрать из области метки, то метка отображается с использованием эффекта «вдавливания», текст выравнивается по левому краю. Здесь мы имеем дело с событием класса MouseEvent, связанным с действиями с мышью.

При реализации программы мы используем следующий подход. В классе MyFrame реализуем интерфейсы ActionListener и MouseListener. Из интерфейса ActionListener в классе описывается метод actionPerformed(). Данный метод вызывается при обработке щелчков на кнопках. В интерфейсе MouseListener в общей сложности пять абстрактных методов: mouseEntered(), mouseExited(), mousePressed(), mouseReleased() и mouseClicked(). Каждый из методов имеет свое «назначение»:

- метод mouseEntered() вызывается при наведении курсора мыши на область компонента, в котором зарегистрирован обработчик;
- метод mouseExited() вызывается, когда курсор мыши покидает область компонента, в котором зарегистрирован обработчик;
- метод mousePressed() определяет реакцию на нажатие кнопки мыши;
- метод mouseReleased() вызывается как реакция на отпускание кнопки мыши;
- метод mouseClicked() вызывается при щелчке на кнопке мыши (нажатие и отпускание кнопки мыши).

Нас в данном случае интересует только два метода: mouseEntered() и mouseExited(). В теле метода mouseEntered() мы опишем реакцию метки на наведение курсора на область метки, а в методе mouseExited() опишем

реакцию метки на перемещение курсора за область метки. Но поскольку при реализации интерфейса в неабстрактном классе необходимо описать все его методы, то нам кроме перечисленных двух методов придется определить и три оставшихся (mousePressed(), mouseReleased() и mouseClicked()). К решению этой формальной задачи мы подойдем тоже формально: «ненужные» методы будут описаны с пустым телом.

НА ЗАМЕТКУ

Ситуация, когда из нескольких абстрактных методов интерфейса на самом деле нужны только некоторые, встречается достаточно часто. Выходом может быть описание остальных методов с пустым телом. Также для многих интерфейсов имеются специальные классы-адаптеры, которые реализуют соответствующий интерфейс с методами, описанными с пустым телом (интерфейс MouseListener реализуется в классе-адаптере MouseAdapter). В таком случае вместо реализации интерфейса можно использовать наследование класса-адаптера с переопределением кода только тех методов, которые собственно нужны. Проблема, однако, в том, что в Java в подклассе может наследоваться только один класс.

Теперь рассмотрим программный код в листинге 14.2.

Листинг 14.2. Программный код проекта MoreTextFieldApplication

```
// Импорт классов:
import javax.swing.*;
import javax.swing.border.*;
import java.awt.event.*;
// Класс для создания окна реализует
// интерфейсы ActionListener и MouseListener:
class MyFrame extends JFrame implements ActionListener,MouseListener{
  // Метка:
  private JLabel L;
  // Текстовое поле:
  private JTextField T;
  // Название кнопок:
  private String apply="Применить";
  private String exit="Закрыть";
  // Метод для обработки щелчка на кнопке:
```

```
public void actionPerformed(ActionEvent e){
  // Определение названия кнопки, на которой
  // произошло событие:
  String txt=e.getActionCommand();
  // Если первая кнопка:
  if(txt.equals(apply)){
    // Присваивание текста из поля метке:
    L.setText(T.getText());
  } // Если вторая кнопка:
  else{
    // Завершение выполнения программы:
    System.exit(0);
  }
}
// Метод выполняется, когда курсор мыши покидает
// область компонента:
public void mouseExited(MouseEvent e){
  // Применение эффекта "вдавливания" к метке:
  L.setBorder(BorderFactory.createBevelBorder(BevelBorder.LOWERED));
  // Применение выравнивания по левому краю
  // для текста в метке:
  L.setHorizontalAlignment(JLabel.LEFT);
}
// Метод вызывается, когда курсор оказывается над
// областью компонента:
public void mouseEntered(MouseEvent e){
  // Для метки применяется эффект "поднятия":
  L.setBorder(BorderFactory.createBevelBorder(BevelBorder.RAISED));
  // Применение выравнивания по правому краю
  // для текста в метке:
  L.setHorizontalAlignment(JLabel.RIGHT);
}
// Методы из интерфейса MouseListener
// с пустой реализацией:
```

```
public void mouseReleased(MouseEvent e){}
public void mousePressed(MouseEvent e){}
public void mouseClicked(MouseEvent e){}
// Конструктор:
MyFrame(){
  // Вызов конструктора суперкласса:
  super("Окно с текстовым полем");
  // Значения для размеров окна:
  int w=300,h=160;
  // Положение и размеры окна:
  setBounds(250,250,w,h);
  // Реакция на щелчок системной пиктограммы:
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  // Окно постоянных размеров:
  setResizable(false);
  // Отключение менеджера компоновки:
  setLayout(null);
  // Создание метки:
  L=new JLabel();
  // Положение и размеры метки:
  L.setBounds(10,10,w-25,30);
  // Выделение метки с помощью эффекта "вдавливания":
  L.setBorder(BorderFactory.createBevelBorder(BevelBorder.LOWERED));
  // Регистрация обработчика в метке:
  L.addMouseListener(this);
  // Создание текстового поля:
  T=new JTextField();
  // Положение и размеры поля:
  T.setBounds(L.getX(),50,L.getWidth(),30);
  // Ширина кнопок:
  int bw=(T.getWidth()-20)/2;
  // Создание первой кнопки:
  JButton appB=new JButton(apply);
  // Положение и размеры первой кнопки:
```

```
    appB.setBounds(T.getX(),90,bw,30);
    // Отмена режима отображения фокуса
    // для первой кнопки:
    appB.setFocusPainted(false);
    // Регистрация объекта окна
    // обработчиком события для первой кнопки:
    appB.addActionListener(this);
    // Создание второй кнопки:
    JButton extB=new JButton(exit);
    // Положение и размеры второй кнопки:
    extB.setBounds(appB.getX()+appB.getWidth()+20,appB.getY(),appB.getWidth(),appB.
getHeight());
    // Отмена режима отображения фокуса
    // для второй кнопки:
    extB.setFocusPainted(false);
    // Регистрация объекта окна
    // обработчиком события для второй кнопки:
    extB.addActionListener(this);
    // Добавление в окно метки:
    add(L);
    // Добавление поля в окно:
    add(T);
    // Добавление первой кнопки в окно:
    add(appB);
    // Добавление второй кнопки в окно:
    add(extB);
    // Отображение окна:
    setVisible(true);
  }
}
// Главный класс:
class MoreTextFieldDemo{
  // Главный метод:
  public static void main(String[] args){
```

```
    // Создание окна:
    new MyFrame();
  }
}
```

При запуске программы на выполнение отображается окно, которое внешне нам уже знакомо. Но теперь функциональность окна немого иная. На рис. 14.6 показано окно, в поле которого введен текст. Окно показано перед нажатием кнопки **Применить**, поэтому текст в метке еще не отображается.

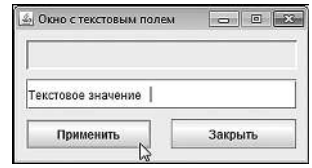


Рис. 14.6. Окно перед нажатием кнопки **Применить**

После нажатия кнопки **Применить** в метке появляется текст — такой же, как в окне. Ситуация проиллюстрирована на рис. 14.7.

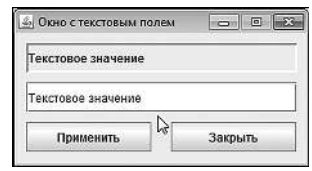


Рис. 14.7. Окно после нажатия кнопки **Применить**

Если теперь навести курсор мыши на метку, то текст метки начнет выравниваться по правому краю, а сама метка будет «приподнята» над областью окна, как показано на рис. 14.8.

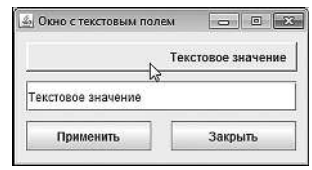


Рис. 14.8. Окно при наведении курсора мыши на область метки

НА ЗАМЕТКУ

При вводе текста в текстовое поле в конце было намеренно добавлено несколько пробелов, чтобы при выравнивании текста по правому краю между текстом и границей метки был «зазор».

Если убрать курсор из области метки, то все вернется в исходное состояние: метка будет «вдавленной», а текст в метке выравнивается по левому краю. Если щелкнуть кнопку **Закрыть**, окно будет закрыто.

Сделаем еще несколько замечаний по поводу программного кода. В конструкторе класса MyFrame для обеих кнопок в качестве аргумента методу addActionListener() передается ссылка this на объект окна. Поэтому при щелчке на любой из кнопок вызывается метод actionPerformed(), описанный в классе MyFrame. В теле метода объявляется текстовая переменная txt, значением которой присваивается результат выражения e.getActionCommand(). Через e обозначен аргумент метода — ссылка на объект события класса ActionEvent. Из этого объекта вызывается метод getActionCommand(). Результатом метод возвращает текстовое значение, представляющее собой *командную строку*, ассоциируемую с компонентом, на котором (с которым) произошло событие. Для кнопок командной строкой (по умолчанию) является название, отображаемое на кнопке. Мы названия для кнопок предусмотрительно «оформили» в виде закрытых текстовых полей apply (название для первой кнопки) и exit (название для второй кнопки). Поэтому в теле метода actionPerformed() в условном операторе проверяется условие txt.equals(apply). Если оно истинно, то в переменную txt записано такое же текстовое значение, что и в поле apply. Принимая во внимание, что текст из поля apply определяет название первой кнопки, вывод становится очевидным: истинность условия означает, что нажата первая кнопка. В таком случае командой L.setText(T.getText()) текст из поля применяется к метке. В противном случае (если условие ложно) методом исключения приходим к выводу, что щелчок выполнен на второй кнопке. Если так, то командой System.exit(0) завершается выполнение программы.

Также в конструкторе с помощью метода addMouseListener() в метке регистрируется обработчик для события класса MouseEvent. Интерес, как отмечалось, представляют два метода. В теле метода mouseEntered() командой L.setBorder(BorderFactory.createBevelBorder(BevelBorder.RAISED)) для метки задается режим отображения с эффектом «поднятия». Командой L.setHorizontalAlignment(JLabel.RIGHT) для текста метки задается режим выравнивания по правому краю. Эти операции выполняются, когда курсор

мыши наводится на область компонента, в котором зарегистрирован обработчик. Таким компонентом является метка.

В теле метода mouseExited() командой L.setBorder(BorderFactory.createBevelBorder(BevelBorder.LOWERED)) к метке применяется режим отображения с эффектом «вдавливания», а командой L.setHorizontalAlignment(JLabel.LEFT) задается режим выравнивания текста по левому краю. Указанные действия имею место, когда курсор мыши «уходит» из области компонента (в данном случае метки), в котором зарегистрирован обработчик.

ДЕТАЛИ

Методы mouseEntered() и mouseExited() описаны так, что операции выполняются с текстовой меткой. Обработчик, который «привлекает» данные методы, также зарегистрирован для метки. Но чисто теоретически обработчик можно зарегистрировать и для иных компонентов. Например, желающие могут провести следующий эксперимент: для одной (или обеих) кнопок с помощью метода addMouseListener() зарегистрировать еще один обработчик — на этот раз для события MouseEvent. Для этого достаточно вызвать из объекта кнопки метод addMouseListener() с аргументом this. В результате при наведении курсора на кнопку метка будет вести себя точно таким же образом, как и при наведении курсора на метку.

Обработчик для поля

В рассмотренных выше примерах текстовое поле было «статическим»: из поля только считывалось значение, но обработка событий для собственно текстового поля не выполнялась. Поэтому для применения текста из текстового поля к метке нам приходилось щелкать кнопку. Однако все может быть реализовано несколько проще: предусмотрев обработчик для поля, мы можем исключить из схемы «посредника» в виде кнопки и добиться ситуации, когда при вводе текста в текстовое поле он автоматически отображается в текстовой метке.

Далее мы рассматриваем модификацию предыдущего примера. Но теперь при вводе текста в поле текст автоматически отображается в метке. Кнопка теперь одна, и щелчок на ней приводит к закрытию окна и завершению выполнения программы. Но название кнопки отображается синим цветом, а при наведении курсора мыши на кнопку название кнопки отображается жирным шрифтом увеличенного размера, текст красного цвета и используется эффект подчеркивания. Если курсор мыши убрать

из области кнопки, то кнопка возвращается в исходное «состояние». Также при наведении курсора мыши на метку она отображается с эффектом «поднятия» и текст в метке выравнивается по правому краю.

В рассматриваемой далее программе мы имеем дело с обработкой событий классов ActionEvent (щелчок на кнопке), KeyEvent (ввод текста в поле) и MouseEvent (наведение курсора на кнопку и наведение курсора на метку). Соответственно, в классе MyFrame, кроме наследования окна, реализуются три интерфейса: ActionListener, KeyListener и MouseListener. Обработчиком всех событий для всех компонентов регистрируется объект окна. Для поля методом addKeyListener() объект окна регистрируется как обработчик для события класса KeyEvent. Для кнопки регистрируется два обработчика: методом addActionListener() объект окна регистрируется как обработчик для события класса ActionEvent, а методом addMouseListener() выполняется регистрация объекта окна в качестве обработчика события класса MouseEvent. Для метки регистрируется только обработчик события класса MouseEvent (используется метод addMouseListener()).

НА ЗАМЕТКУ

Один и тот же объект (объект окна) используется обработчиком события класса MouseEvent и для метки, и для кнопки. С похожей ситуацией, когда один обработчик использовался для нескольких компонентов, мы уже сталкивались. Но там речь шла о двух кнопках. В данном случае один компонент является кнопкой, а другой является меткой. Как ранее, при обработке события в теле соответствующих методов приходится выполнять идентификацию компонента, на котором это событие произошло.

Теперь рассмотрим программный код, представленный в листинге 14.3.

Листинг 14.3. Программный код проекта HandlingTextFieldApplication

```
// Импорт классов:
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import java.awt.event.*;
// Класс для создания окна реализует интерфейсы
// ActionListener, MouseListener и KeyListener:
```

```
class MyFrame extends JFrame implements ActionListener,MouseListener,KeyListener{
  // Метка:
  private JLabel L;
  // Текстовое поле:
  private JTextField T;
  // Кнопка:
  private JButton B;
  // Тип шрифта для кнопки:
  private String name="Arial";
  // Размер шрифта для кнопки:
  private int size=15;
  // Название кнопки:
  private String exit="Закрыть";
  // Метод для обработки щелчка на кнопке:
  public void actionPerformed(ActionEvent e){
    // Завершение выполнения программы:
    System.exit(0);
  }
  // Метод выполняется, когда курсор мыши покидает
  // область компонента:
  public void mouseExited(MouseEvent e){
    // Если событие произошло на метке:
    if(e.getSource()==L){
      // Применение эффекта "вдавливания" к метке:
      L.setBorder(BorderFactory.createBevelBorder(BevelBorder.LOWERED));
      // Применение выравнивания по левому краю
      // для текста в метке:
      L.setHorizontalAlignment(JLabel.LEFT);
    } // Если событие произошло на кнопке:
    else{
      // Текст (обычный) для кнопки:
      B.setText(exit);
```

```
      // Синий цвет для текста кнопки:
      B.setForeground(Color.BLUE);
      // Шрифт (обычный) для кнопки:
      B.setFont(new Font(name,Font.PLAIN,size));
    }
  }
  // Метод вызывается, когда курсор оказывается над
  // областью компонента:
  public void mouseEntered(MouseEvent e){
    // Если событие произошло на метке:
    if(e.getSource()==L){
      // Для метки применяется эффект "поднятия":
      L.setBorder(BorderFactory.createBevelBorder(BevelBorder.RAISED));
      // Применение выравнивания по правому краю
      // для текста в метке:
      L.setHorizontalAlignment(JLabel.RIGHT);
    } // Если событие произошло на кнопке:
    else{
      // Текст (подчеркнутый) для кнопки:
      B.setText("<html><u>"+exit+"</u></html>");
      // Красный цвет для текста кнопки:
      B.setForeground(Color.RED);
      // Шрифт (жирный) для текста кнопки:
      B.setFont(new Font(name,Font.BOLD,size+2));
    }
  }
  // Методы из интерфейса MouseListener
  // с пустой реализацией:
  public void mouseReleased(MouseEvent e){}
  public void mousePressed(MouseEvent e){}
  public void mouseClicked(MouseEvent e){}
  // Метод вызывается при отпускании клавиши
```

```
  // на клавиатуре:
  public void keyReleased(KeyEvent e){
    // К метке применяется текст из текстового поля:
    L.setText(T.getText());
  }
  // Методы из интерфейса KeyListener
  // с пустой реализацией:
  public void keyPressed(KeyEvent e){}
  public void keyTyped(KeyEvent e){}
  // Конструктор:
  MyFrame(){
    // Вызов конструктора суперкласса:
    super("Окно с текстовым полем");
    // Положение и размеры окна:
    setBounds(250,250,300,160);
    // Реакция на щелчок системной пиктограммы:
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // Окно постоянных размеров:
    setResizable(false);
    // Отключение менеджера компоновки:
    setLayout(null);
    // Создание метки:
    L=new JLabel();
    // Положение и размеры метки:
    L.setBounds(10,10,275,30);
    // Выделение метки с помощью эффекта "вдавливания":
    L.setBorder(BorderFactory.createBevelBorder(BevelBorder.LOWERED));
    // Регистрация в метке обработчика для
    // события класса MouseEvent:
    L.addMouseListener(this);
    // Создание текстового поля:
    T=new JTextField();
```

```
    // Положение и размеры поля:
    T.setBounds(10,50,275,30);
    // Регистрация в поле обработчика для
    // события класса KeyEvent:
    T.addKeyListener(this);
    // Создание кнопки:
    B=new JButton(exit);
    // Положение и размеры кнопки:
    B.setBounds(60,90,175,30);
    // Отмена режима отображения фокуса
    // для кнопки:
    B.setFocusPainted(false);
    // Шрифт для текста кнопки:
    B.setFont(new Font(name,Font.PLAIN,size));
    // Синий цвет для текста кнопки:
    B.setForeground(Color.BLUE);
    // Регистрация в кнопке обработчика для
    // события класса ActionEvent:
    B.addActionListener(this);
    // Регистрация в кнопке обработчика для
    // события класса MouseEvent:
    B.addMouseListener(this);
    // Добавление в окно метки:
    add(L);
    // Добавление поля в окно:
    add(T);
    // Добавление кнопки в окно:
    add(B);
    // Отображение окна:
    setVisible(true);
  }
}
```

```
// Главный класс:
class HandlingTextFieldDemo{
  // Главный метод:
  public static void main(String[] args){
    // Создание окна:
    new MyFrame();
  }
}
```

В начале выполнения программы на экране появляется окно, вид которого показан на рис. 14.9.

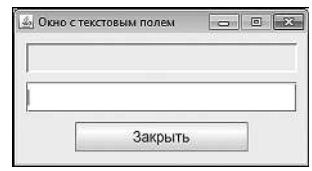


Рис. 14.9. Отображаемое окно содержит метку, поле и одну кнопку, в которой текст отображается синим цветом

У окна одна кнопка, название которой отображается синим цветом. Окно имеет поле ввода и метку (в начальный момент они пустые). Если начать вводить текст в поле ввода, то текст автоматически дублируется в текстовой метке. Процесс ввода текста в поле ввода и его дублирования в метке показан на рис. 14.10.

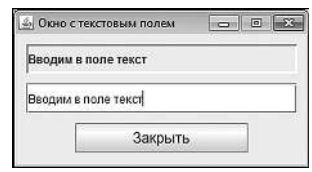


Рис. 14.10. При вводе текста в текстовом поле она автоматически дублируется в метке

Если навести курсор на область метки, то вместо эффекта «вдавливания» метка отображается с эффектом «поднятия», а текст выравнивается по правому краю. Как может выглядеть окно в таком случае показано на рис. 14.11.

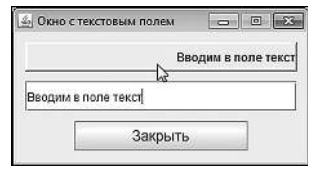


Рис. 14.11. При наведении курсора мыши на метку она отображается с использованием эффекта «поднятия», а текст выравнивается по правому краю

После «ухода» курсора из области метки метка возвращается в исходное состояние (эффект «вдавливания» и выравнивание текста по левому краю).

Если навести курсор мыши на кнопку (но не щелкать ее), то вместо синего цвета используется красный цвет для отображения названия кнопки, текст отображается с подчеркиванием, шрифт применяется жирный и его размер немного больше (на величину 2) шрифта, которым отображается название кнопки, если не нее не наведен курсор.

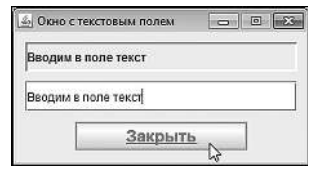


Рис. 14.12. При наведении курсора мыши на кнопку надпись в области кнопки отображается с подчеркиванием, красным цветом, стиль шрифта жирный, а размер шрифта увеличивается

Если убрать курсор из области кнопки, она примет прежний вид (как до наведения на нее курсора мыши). Щелчок на кнопке приводит к тому, что программа завершает выполнение.

Сделаем несколько замечаний относительно особенностей программного кода. Стоит заметить, что в данном случае в классе `MyFrame` описываются закрытые поля, являющиеся ссылками на метку, текстовое поле и кнопку. Также через закрытые поля определяется название для шрифта (текстовое поле `name` со значением `"Arial"`), размер шрифта (поле `size` со значение `15`) и название кнопки (текстовое поле `exit` со значением `"Закрыть"`).

В интерфейсе `MouseListener`, как мы уже знаем, пять абстрактных методов. Мы три описываем с пустым телом, и два (методы `mouseExited()`

и mouseEntered()) содержат полезный код. В обоих методах коды идентичные по своей «идеологии». В теле этих методов использован условный оператор, в котором проверяется условие вида e.getSource()==L. Здесь из аргумента метода — объекта e события класса MouseEvent, — вызывается метод getSource(). Результатом метод возвращает ссылку на объект компонента, на котором произошло событие, реализованное через объект события e.

НА ЗАМЕТКУ

Результат метода getSource() — ссылка типа Object. Поскольку класс Object находится в вершине иерархии наследования классов Java, то возвращаемая методом getSource() ссылка в принципе может ссылаться на объект любого класса. Этот момент важный, поскольку событие может происходить на компонентах разного типа, поэтому тип компонента, на который возвращается ссылка, заранее не известен. Благодаря тому, что результат метода обозначен как типа Object, есть возможность возвращать результатом метода ссылку на любой объект. У этой «медали» есть «обратная сторона»: после получения ссылки на объект компонента часто приходится явно выполнять приведение типа (перед ссылкой в круглых скобках указывается имя класса, к которому относится ссылка). В данном случае приведение типов не выполняется, поскольку речь идет о простом сравнении ссылок.

Условие e.getSource()==L истинно, если ссылка, возвращаемая методом getSource(), совпадает со ссылкой на метку L. Это означает, что событие произошло на метке. В таком случае выполняются команды по настройке параметров метки.

При ложном условии e.getSource()==L методом исключения приходим к выводу, что событие произошло на кнопке. В таком случае для кнопки задается текст, определяется цвет, которым отображается название кнопки, и задается шрифт для кнопки. Например, командой B.setText(exit) названием кнопки задается текст из поля exit. В то же время командой B.setText("<html><u>"+exit+"</u></html>") текст для названия кнопки «упаковывается» в гипертекстовую разметку:

- дескриптор <html> обозначает начало блока гипертекста;
- дескриптор <u> обозначает начало блока текста, выделяемого подчеркиванием;
- далее следует текст из поля exit;

- дескриптор </u> означает завершение блока выделяемого подчеркиванием;
- дескриптор </html> обозначает завершение текста с гипертекстовой разметкой.

В итоге после выполнения указанной команды название кнопки отображается с применением эффекта подчеркивания.

Выбор цвета для отображения названия кнопки выполняется командами B.setForeground(Color.BLUE) (синий цвет) и B.setForeground(Color.RED) (красный цвет). Наконец, шрифт для кнопки задаем, вызывая из объекта кнопки B метод setFont(). Аргументом методу передается анонимный объект класса Font. При создании этого объекта аргументами конструктора класса указываются:

- поле name, определяющее название шрифта;
- статическая константа PLAIN или BOLD класса Font, определяющие обычный (не жирный и не курсивный) шрифт, и жирный шрифт соответственно;
- размер шрифта: определяется значением поля size или выражением size+2.

В интерфейсе KeyListener объявлено три абстрактных метода: keyPressed(), keyReleased() и keyTyped().У каждого метода один аргумент, являющийся ссылкой на объект события класса KeyEvent. Метод keyPressed() вызывается при нажатии клавиши, метод keyReleased() вызывается при отпускании клавиши, и метод keyTyped() обрабатывает ввод символа. Мы реализуем обработку события, связанную с отпусканием клавиши. Поэтому с «полезным» кодом описывается метод keyReleased(), а методы keyPressed() и keyTyped() описываются с пустым телом.

Код метода keyReleased() очень простой и нам уже знаком: командой L.setText(T.getText()) считывается текст из текстового поля и присваивается метке.

В конструкторе класса MyFrame на этот раз мы, для уменьшения объема кода, просто указываем числовые значения для координат и размеров компонентов. Для всех компонентов (метка, поле и кнопка) обработчиком регистрируется объект окна. Но речь идет об обработке разных событий. Понять, каких именно, можно по названию метода, которым регистрируется обработчик. Например, при выполнении команды

B.addActionListener(this) в кнопке регистрируется обработчик для события класса ActionEvent, а командой B.addMouseListener(this) в кнопке регистрируется обработчике для события класса MouseEvent. Просто в данном случае так вышло, что обработчик в обоих случаях один и тот же (но обрабатывает он разные события по-разному). Командой L.addMouseListener(this) в метке регистрируется обработчик события класса MouseEvent, а командой T.addKeyListener(this) в поле регистрируется обработчик события KeyEvent.

Классы-адаптеры

Вы убедили меня. Я понял, что вам еще нужен.

Из к/ф «Старики-разбойники»

В рассмотренном выше примере при реализации в классе интерфейсов KeyListener и MouseListener нам приходилось некоторые методы описывать с пустым телом. Не описать мы их не могли, поскольку в таком случае соответствующий класс был бы абстрактным. Данный подход не очень удобный, поскольку подразумевает написание некоторого «лишнего» кода. Выходом может стать использование *классов-адаптеров*.

Основные классы-адаптеры

Классы-адаптеры содержат реализацию интерфейса (или интерфейсов), используемых при обработке событий, с пустой реализацией для методов интерфейса (название класса-адаптера содержит ключевое слово *Adapter*). Поэтому вместо явной реализации интерфейса с описание всех его методов, можно воспользоваться наследованием класса-адаптера, переопределив лишь те методы, которые реально используются. Классы адаптеры существуют для многих интерфейсов, используемых при обработке событий.

НА ЗАМЕТКУ

Например, интерфейс KeyListener реализуется в классе-адаптере KeyAdapter. А вот для интерфейса ActionListener класса-адаптера нет.

Некоторые из классов-адаптеров перечислены в табл. 14.4.

Далее мы рассмотрим небольшой пример использования классов-адаптеров на практике.

Табл. 14.4. Некоторые классы-адаптеры

Класс-адаптер	Реализуемые интерфейсы
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener MouseMotionListener MouseWheelListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowFocusListener WindowListener WindowStateListener

Использование классов-адаптеров

Для интерфейсов KeyListener и MouseListener есть классы-адаптеры KeyAdapter и MouseAdapter соответственно. В представленном в листинге 14.4 программном коде решается та же задача, что и в предыдущем случае (см. листинг 14.3), но теперь класс MyFrame не реализует интерфейсы, а вместо этого для каждого из компонентов (метка, поле и кнопка) создается отдельный обработчик (для кнопки даже два). Обработчики создаются на основе анонимных классов, наследующих классы-адаптеры. Исключением является обработчик события класса ActionEvent, который регистрируется в кнопке и создается реализацией интерфейса AcvtionListener (поскольку для этого интерфейса нет класса-адаптера). Благодаря тому, что в классах-адаптерах методы из соответствующих интерфейсов описаны с пустым кодом, при создании обработчика можно ограничиться описанием только нескольких методов из интерфейса. Как все это выглядит на практике, показано в представленном ниже программном коде.

Листинг 14.4. Программный код проекта UsingAdaptersApplication

```
// Импорт классов:
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import java.awt.event.*;
// Класс для создания окна:
```

```
class MyFrame extends JFrame{
  // Метка:
  private JLabel L;
  // Текстовое поле:
  private JTextField T;
  // Кнопка:
  private JButton B;
  // Тип шрифта для кнопки:
  private String name="Arial";
  // Размер шрифта для кнопки:
  private int size=15;
  // Название кнопки:
  private String exit="Закрыть";
  // Конструктор:
  MyFrame(){
    // Вызов конструктора суперкласса:
    super("Окно с текстовым полем");
    // Положение и размеры окна:
    setBounds(250,250,300,160);
    // Реакция на щелчок системной пиктограммы:
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    // Окно постоянных размеров:
    setResizable(false);
    // Отключение менеджера компоновки:
    setLayout(null);
    // Создание метки:
    L=new JLabel();
    // Положение и размеры метки:
    L.setBounds(10,10,275,30);
    // Выделение метки с помощью эффекта "вдавливания":
    L.setBorder(BorderFactory.createBevelBorder(BevelBorder.LOWERED));
    // Регистрация в метке обработчика для
```

```
// события класса MouseEvent:
L.addMouseListener(new MouseAdapter(){
  // Метод вызывается, когда курсор оказывается
  // над областью метки:
  public void mouseEntered(MouseEvent e){
    // Для метки применяется эффект "поднятия":
    L.setBorder(BorderFactory.createBevelBorder(BevelBorder.RAISED));
    // Применение выравнивания по правому краю
    // для текста в метке:
    L.setHorizontalAlignment(JLabel.RIGHT);
  }
  // Метод выполняется, когда курсор мыши покидает
  // область метки:
  public void mouseExited(MouseEvent e){
    // Применение эффекта "вдавливания" к метке:
    L.setBorder(BorderFactory.createBevelBorder(BevelBorder.LOWERED));
    // Применение выравнивания по левому краю
    // для текста в метке:
    L.setHorizontalAlignment(JLabel.LEFT);
  }
});
// Создание текстового поля:
T=new JTextField();
// Положение и размеры поля:
T.setBounds(10,50,275,30);
// Регистрация в поле обработчика для
// события класса KeyEvent:
T.addKeyListener(new KeyAdapter(){
  // Метод вызывается при отпускании клавиши
  // на клавиатуре:
  public void keyReleased(KeyEvent e){
    // К метке применяется текст
```

```
        // из текстового поля:
        L.setText(T.getText());
      }
    });
    // Создание кнопки:
    B=new JButton(exit);
    // Положение и размеры кнопки:
    B.setBounds(60,90,175,30);
    // Отмена режима отображения фокуса для кнопки:
    B.setFocusPainted(false);
    // Шрифт для текста кнопки:
    B.setFont(new Font(name,Font.PLAIN,size));
    // Синий цвет для текста кнопки:
    B.setForeground(Color.BLUE);
    // Регистрация в кнопке обработчика для
    // события класса ActionEvent:
    B.addActionListener(new ActionListener(){
      // Метод для обработки щелчка на кнопке:
      public void actionPerformed(ActionEvent e){
        // Завершение выполнения программы:
        System.exit(0);
      }
    });
    // Регистрация в кнопке обработчика для
    // события класса MouseEvent:
    B.addMouseListener(new MouseAdapter(){
      // Метод выполняется, когда курсор мыши покидает
      // область кнопки:
      public void mouseExited(MouseEvent e){
        // Текст (обычный) для кнопки:
        B.setText(exit);
        // Синий цвет для текста кнопки:
```

```
        B.setForeground(Color.BLUE);
        // Шрифт (обычный) для кнопки:
        B.setFont(new Font(name,Font.PLAIN,size));
      }
      // Метод вызывается, когда курсор оказывается над
      // областью кнопки:
      public void mouseEntered(MouseEvent e){
        // Текст (подчеркнутый) для кнопки:
        B.setText("<html><u>"+exit+"</u></html>");
        // Красный цвет для текста кнопки:
        B.setForeground(Color.RED);
        // Шрифт (жирный) для текста кнопки:
        B.setFont(new Font(name,Font.BOLD,size+2));
      }
    });
    // Добавление в окно метки:
    add(L);
    // Добавление поля в окно:
    add(T);
    // Добавление кнопки в окно:
    add(B);
    // Отображение окна:
    setVisible(true);
  }
}
// Главный класс:
class UsingAdaptersDemo{
  // Главный метод:
  public static void main(String[] args){
    // Создание окна:
    new MyFrame();
  }
}
```

В плане функциональности программа такая же, как и в предыдущем случае. Изменился лишь способ реализации обработчиков.

Резюме

Извините, читала — увлеклась. Такая книжка интересная попалась.

Из к/ф «Безумный день инженера Баркасова»

- Существуют различные способы создания обработчиков событий. Обработчики могут создаваться на основе классов, реализующих определенный интерфейс или наследующих класс-адаптер.
- Один обработчик может использоваться одновременно для нескольких компонентов. В таком случае в обработчике обычно выполняется идентификация компонента, на котором произошло событие.
- В одном компоненте может быть зарегистрировано сразу несколько обработчиков.

Глава 15
ГРАФИЧЕСКИЕ КОМПОНЕНТЫ

Форму будете создавать под моим личным контролем. Форме сегодня придается большое... содержание.

Из к/ф «Чародеи»

В этой главе мы кратко рассмотрим элементарные способы использования некоторых основных (наиболее часто используемых) компонентов графического интерфейса. Наш подход состоит в том, что мы будем решать однотипные задачи, но каждый раз используем различные компоненты графического интерфейса.

Раскрывающийся список

Ну и что, что квартет? Добавьте сюда ещё людей — будет большой, массовый квартет.

Из к/ф «Карнавальная ночь»

Мы создаем программу, выполнение которой приводит к отображению окна, в котором есть *раскрывающийся список* с названиями животных (*лиса*, *волк*, *медведь* и *енот*). При выборе пункта в раскрывающемся списке отображается изображение соответствующего зверя. Также в окне есть кнопка **ОК**, щелчок на которой приводит к закрытию окна.

НА ЗАМЕТКУ

Для корректного выполнения программы в папке d:\books\pictures должны находиться файлы с изображениями. Размер каждого изображения составляет 150 пикселов в ширину и 100 пикселов в высоту. Названия файлов такие: fox.jpg (изображение лисы), wolf.jpg (изображение волка), bear.jpg (изображение медведя) и raccoon.jpg (изображение енота).

Для реализации раскрывающегося списка используем объект класса JComboBox. При создании объекта класса JComboBox аргументом

конструктору класса передается массив, элементы которого становятся пунктами раскрывающегося списка.

В объекте раскрывающегося списка регистрируется обработчик события ItemEvent. Обработчиком регистрируется объект окна. Класс, на основе которого создается объект окна, реализует интерфейс ItemListener. Из этого интерфейса описывается метод itemStateChanged(). Аргументом метода является ссылка на объект события класса ItemEvent. Метод вызывается при изменении состояния раскрывающегося списка (когда выбирается новый пункт в списке). Код программы представлен в листинге 15.1.

Листинг 15.1. Программный код проекта UsingComboBoxApplication

```
// Импорт классов:
import javax.swing.*;
import java.awt.event.*;
// Класс для создания объекта окна реализует
// интерфейс ItemListener:
class MyFrame extends JFrame implements ItemListener{
  // Раскрывающийся список:
  private JComboBox CB;
  // Кнопка:
  private JButton B;
  // Метка:
  private JLabel L;
  // Массив с названиями животных:
  private String[] animals=new String[]{"Лиса","Волк","Медведь","Енот"};
  // Массив с названиями файлов с изображениями:
  private String[] files=new String[]{"fox.jpg","wolf.jpg","bear.jpg","raccoon.jpg"};
  // Путь к каталогу с изображениями:
  private String path="d:/books/pictures/";
  // Массив с изображениями:
  private ImageIcon[] imgs;
  // Метод вызывается при изменении состояния списка:
  public void itemStateChanged(ItemEvent e){
    // Метке присваивается новое изображение:
```

```
    L.setIcon(imgs[CB.getSelectedIndex()]);
  }
  // Конструктор:
  MyFrame(){
    // Настройка параметров окна:
    super("Раскрывающийся список");
    setBounds(250,250,300,150);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setResizable(false);
    setLayout(null);
    // Создание массива из ссылок на объекты изображений:
    imgs=new ImageIcon[files.length];
    // Создание объектов изображений
    // и заполнение массива:
    for(int k=0;k<imgs.length;k++){
      imgs[k]=new ImageIcon(path+files[k]);
    }
    // Создание метки с изображением:
    L=new JLabel(imgs[0]);
    // Положение и размеры метки:
    L.setBounds(10,10,150,100);
    // Добавление метки в окно:
    add(L);
    // Создание и добавление в окно метки с текстом:
    JLabel lbl=new JLabel("Сделайте выбор:");
    lbl.setBounds(170,10,120,20);
    add(lbl);
    // Создание раскрывающегося списка:
    CB=new JComboBox(animals);
    // Положение и размеры списка:
    CB.setBounds(170,40,120,30);
    // В списке выбирается начальный пункт:
    CB.setSelectedIndex(0);
```

```
    // Регистрация обработчика в списке:
    CB.addItemListener(this);
    // Добавление списка в окно:
    add(CB);
    // Создание кнопки:
    B=new JButton("OK");
    // Положение и размеры кнопки:
    B.setBounds(170,80,120,30);
    // Регистрация обработчика в кнопке:
    B.addActionListener(e->System.exit(0));
    // Добавление кнопки в окно:
    add(B);
    // Отображение окна на экране:
    setVisible(true);
  }
}
// Главный класс:
class UsingComboBoxDemo{
  // Главный метод:
  public static void main(String[] args){
    // Создание объекта окна:
    new MyFrame();
  }
}
```

При запуске на выполнение программы отображается окно, представленное на рис. 15.1.

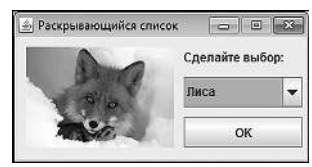


Рис. 15.1. Окно с раскрывающимся списком отображается при выполнении программы

В левой части она находится область с изображением (по умолчанию это изображение лисы), а в правой части окна есть статическая метка с текстом, раскрывающийся список и кнопка **ОК**. Список можно раскрыть и выбрать там новый пункт, как это показано на рис. 15.2.

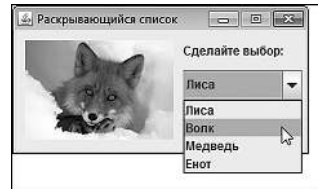


Рис. 15.2. Выбор пункта в раскрывающемся списке

На рис. 15.3 показано окно с изображением волка.

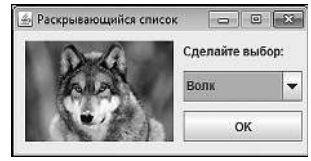


Рис. 15.3. Окно с изображением волка

На рис. 15.4 в окне показано изображение медведя.

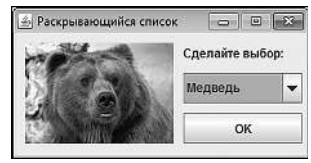


Рис. 15.4. Окно с изображением медведя

Окно с изображением енота показано на рис. 15.5.

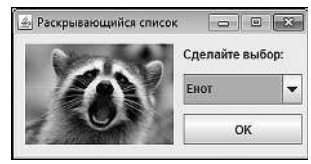


Рис. 15.5. Окно с изображением енота

Большинство команд в программном коде должно быть понятно читателю. Прокомментируем лишь те, что связаны с использованием раскрывающегося списка. А именно, в классе MyFrame объявляется закрытое поле CB, являющееся ссылкой на объект класса JComboBox (объект раскрывающегося списка). Также есть поля для кнопки и метки, в которой предполагается размещать изображение.

Текстовый массив animals содержит название животных (эти названия отображаются в раскрывающемся списке). В текстовый массив files заносятся названия файлов с изображениями животных. Текстовое поле path содержит текст "d:/books/pictures/", определяющий место размещения файлов с изображениями. Также мы используем массив imgs ссылок типа ImageIcon на объекты изображений.

В конструкторе командой imgs=new ImageIcon[files.length] создается массив объектных ссылок класса ImageIcon, а количество элементов в массиве определяется количеством элементов в массиве files. Затем запускается оператор цикла, в котором для каждого изображения создается объект, и ссылка на него присваивается очередному элементу массива imgs (при создании метки аргументом конструктору передается ссылка на объект первого изображения).

Объект для раскрывающегося списка создается командой CB=new JComboBox(animals). Список формируется на основе элементов массива animals. Положение и размеры списка задаются с помощью метода setBounds(). Командой CB.setSelectedIndex(0) в списке выбирается начальный пункт (пункт с нулевым индексом) в списке (хотя по умолчанию он и так бы был выбран). Командой CB.addItemListener(this) обработчиком события ItemEvent регистрируется объект окна. При изменении состояния раскрывающегося списка (выборе нового пункта в списке) выполняется метод itemStateChanged(). В теле метода выполняется команда L.setIcon(imgs[CB.getSelectedIndex()]). В этой команде из объекта метки L вызывается метод setIcon(), которым для метки задается изображение. Изображение для метки определяется элементом массива imgs с индексом, определяемым значением выражения CB.getSelectedIndex(). Данное выражение является вызовом метода getSelectedIndex() из объекта списка CB. Методом getSelectedIndex() в качестве результата возвращается индекс пункта в раскрывающемся списке, который выбран на данный момент. Таким образом, в метке отображается изображение из массива imgs с таким же индексом, что и пункт, выбранный в раскрывающемся списке.

В окно раскрывающийся список добавляется командой add(CB).

НА ЗАМЕТКУ

Интерфейс ItemListener является функциональным. Поэтому вместо явного описания метода itemStateChanged() мы могли бы прибегнуть к помощи лямбда-выражения.

Пунктами в раскрывающемся списке могут быть не только текстовые значения, но и, например, изображения. В листинге 15.2 представлена версия рассмотренной выше программы, но теперь в раскрывающемся списке выбирается изображение, а в текстовой метке отображается название выбранного животного.

Листинг 15.2. Программный код проекта MoreComboBoxApplication

```
// Импорт классов:
import javax.swing.*;
import java.awt.event.*;
// Класс для создания объекта:
class MyFrame extends JFrame{
  // Раскрывающийся список:
  private JComboBox CB;
  // Кнопка:
  private JButton B;
  // Метка:
  private JLabel L;
  // Массив с названиями животных:
  private String[] animals=new String[]{"Лиса","Волк","Медведь","Енот"};
  // Массив с названиями файлов с изображениями:
  private String[] files=new String[]{"fox.jpg","wolf.jpg","bear.jpg","raccoon.jpg"};
  // Путь к каталогу с изображениями:
  private String path="d:/books/pictures/";
  // Массив с изображениями:
  private ImageIcon[] imgs;
  // Конструктор:
  MyFrame(){
    // Настройка параметров окна:
    super("Раскрывающийся список");
```

```
setBounds(250,250,300,150);
setDefaultCloseOperation(EXIT_ON_CLOSE);
setResizable(false);
setLayout(null);
// Создание массива из ссылок на объекты изображений:
imgs=new ImageIcon[files.length];
// Создание объектов изображений
// и заполнение массива:
for(int k=0;k<imgs.length;k++){
  imgs[k]=new ImageIcon(path+files[k]);
}
// Создание раскрывающегося списка:
CB=new JComboBox(imgs);
// Положение и размеры списка:
CB.setBounds(10,10,150,100);
// Задаем выбранный в списке пункт:
CB.setSelectedIndex(0);
// Регистрация обработчика в списке:
CB.addItemListener(e->L.setText(animals[CB.getSelectedIndex()]));
// Добавление списка в окно:
add(CB);
// Создание и добавление в окно метки с текстом:
JLabel lbl=new JLabel("Ваш выбор:");
lbl.setBounds(170,10,120,20);
add(lbl);
// Создание метки с названием животного:
L=new JLabel(animals[0],JLabel.CENTER);
// Положение и размеры метки:
L.setBounds(170,40,120,30);
// Рамка вокруг метки:
L.setBorder(BorderFactory.createEtchedBorder());
// Добавление метки в окно:
add(L);
// Создание кнопки:
```

```
    B=new JButton("OK");
    // Положение и размеры кнопки:
    B.setBounds(170,80,120,30);
    // Регистрация обработчика в кнопке:
    B.addActionListener(e->System.exit(0));
    // Добавление кнопки в окно:
    add(B);
    // Отображение окна на экране:
    setVisible(true);
  }
}
// Главный класс:
class MoreComboBoxDemo{
  // Главный метод:
  public static void main(String[] args){
    // Создание объекта окна:
    new MyFrame();
  }
}
```

В данном случае при создании объекта раскрывающегося списка аргументом конструктору класса `JComboBox` передается массив `imgs` ссылок на объекты изображений. Обработчик для раскрывающегося списка регистрируется с использованием лямбда-выражения `e->L.setText(animals[CB.getSelectedIndex()])`. В соответствии с этим выражением текстовой метке присваивается текст из массива `animals` с таким же индексом, как индекс пункта, выбранного в раскрывающемся списке.

На рис. 15.6 показано, как выглядит окно, отображаемое при запуске программы на выполнение.

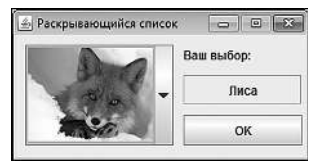


Рис. 15.6. Окно с изображением лисы

Теперь изображение отображается непосредственно в раскрывающемся списке. Соответственно, при щелчке на пиктограмме для раскрытия списка пунктами списка являются изображения, как показано на рис. 15.7.

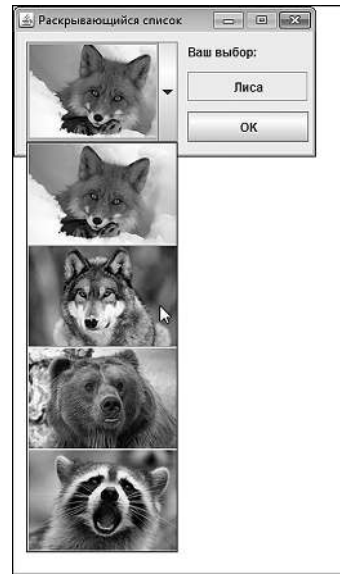


Рис. 15.7. Выбор пункта-изображения в раскрывающемся списке

При выборе пункта-изображения в раскрывающемся списке, это изображение отображается в списке, а в метке справа отображается название выбранного животного. На рис. 15.8 показано окно с изображением волка.

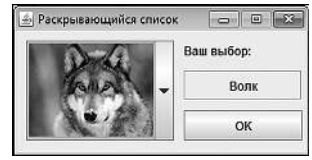


Рис. 15.8. Окно с изображением волка

На рис. 15.9 представлено окно с изображением медведя.

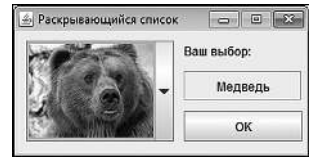


Рис. 15.9. Окно с изображением медведя

Как выглядит окно с изображением енота, показано на рис. 15.10.

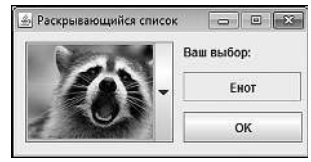


Рис. 15.10. Окно с изображением енота

Далее мы рассмотрим возможность использования других графических компонентов для реализации программы, похожей на рассмотренные выше.

Список выбора

Это что, вы факты излагаете? Или это ирония?

Из к/ф «Старый знакомый»

Кроме раскрывающегося списка, есть еще обычный список, или *список выбора*. Если в раскрывающемся списке его содержимое отображается при щелчке на пиктограмме раскрытия списка, то список выбора отображается весь целиком, но в нем может быть выбран пункт (причем в общем случае даже не один, а сразу несколько). Реализуется список выбора через объект класса JList. Для объекта списка выполняется обработка события ListSelectionEvent, связанного с выбором пункта в списке. Для обработки данного события необходимо реализовать интерфейс ListSelectionListener, в котором объявлен метод valueChanged(). Аргументом методу передается объект события класса ListSelectionEvent, а вызывается метод каждый раз, когда пользователь выбирает пункт в списке выбора. Мы используем подход, при котором обработчиком регистрируется объект окна, и, соответственно, в классе, на основе которого создается объект окна, реализуется интерфейс ListSelectionListener и, соответственно, описывается метод valueChanged().

Теперь рассмотрим программный код, который представлен в листинге 15.3.

Листинг 15.3. Программный код проекта UsingListApplication

```
// Импорт классов:
import javax.swing.*;
```

```
import javax.swing.event.*;
import java.awt.event.*;
// Класс для создания объекта окна реализует
// интерфейс ListSelectionListener:
class MyFrame extends JFrame implements ListSelectionListener{
  // Список выбора:
  private JList LS;
  // Кнопка:
  private JButton B;
  // Метка:
  private JLabel L;
  // Массив с названиями животных:
  private String[] animals=new String[]{"Лиса","Волк","Медведь","Енот"};
  // Массив с названиями файлов с изображениями:
  private String[] files=new String[]{"fox.jpg","wolf.jpg","bear.jpg","raccoon.jpg"};
  // Путь к каталогу с изображениями:
  private String path="d:/books/pictures/";
  // Массив с изображениями:
  private ImageIcon[] imgs;
  // Метод вызывается при изменении
  // состояния списка выбора:
  public void valueChanged(ListSelectionEvent e){
    // Метке присваивается новое изображение:
    L.setIcon(imgs[LS.getSelectedIndex()]);
  }
  // Конструктор:
  MyFrame(){
    // Настройка параметров окна:
    super("Список выбора");
    setBounds(250,250,300,185);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setResizable(false);
    setLayout(null);
```

```
// Создание массива из ссылок на объекты изображений:
imgs=new ImageIcon[files.length];
// Создание объектов изображений
// и заполнение массива:
for(int k=0;k<imgs.length;k++){
  imgs[k]=new ImageIcon(path+files[k]);
}
// Создание метки:
L=new JLabel(imgs[0]);
// Положение и размеры метки:
L.setBounds(10,10,150,100);
// Добавление метки в окно:
add(L);
// Создание и добавление в окно метки с текстом:
JLabel lbl=new JLabel("Сделайте выбор:");
lbl.setBounds(170,10,120,20);
add(lbl);
// Создание списка выбора:
LS=new JList(animals);
// Положение и размеры списка:
LS.setBounds(170,35,120,75);
// Регистрация обработчика в списке:
LS.addListSelectionListener(this);
// Переход в режим выбора только
// одного элемента списка:
LS.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
// В списке выбирается начальный элемент:
LS.setSelectedIndex(0);
// Добавление списка выбора в окно:
add(LS);
// Создание кнопки:
B=new JButton("OK");
// Положение и размеры кнопки:
```

```
    B.setBounds(90,120,120,30);
    // Регистрация обработчика в кнопке:
    B.addActionListener(e->System.exit(0));
    // Добавление кнопки в окно:
    add(B);
    // Отображение окна на экране:
    setVisible(true);
  }
}
// Главный класс:
class UsingListDemo{
  // Главный метод:
  public static void main(String[] args){
    // Создание объекта окна:
    new MyFrame();
  }
}
```

Теперь вместо поля CB класса JComboBox в классе окна MyFrame мы используем поле LS класса JList. В конструкторе класса MyFrame объект для списка выбора создается командой LS=new JList(animals). Пункты списка выбора определяются массивом animals, переданным аргументом конструктору класса JList. Традиционно положение и размеры списка определяются с помощью метода setBounds(). Обработчик регистрируется с помощью метода addListSelectionListener(). Среди «специфических» можно выделить команду LS.setSelectionMode(ListSelectionModel.SINGLE_SELECTION), которой методу setSelectionMode() аргументом передается статическая константа SINGLE_SELECTION интерфейса ListSelectionModel. Такой аргумент метода означает, что список переводится в режим, при котором в списке можно выбрать только один пункт (по умолчанию используется режим, при котором можно выбрать сразу несколько пунктов в списке). Командой LS.setSelectedIndex(0) в списке выделяется пункт с нулевым индексом (начальный пункт в списке выбора). Командой add(LS) список добавляется в окно.

Что касается метода valueChanged(), вызываемого при изменении выбора в списке, то командой L.setIcon(imgs[LS.getSelectedIndex()]) метке L присваивается

изображение из массива imgs, индекс которого определяется выражением LS.getSelectedIndex() — это индекс выбранного в списке пункта.

При запуске программы на выполнение появляется окно, показанное на рис. 15.11.

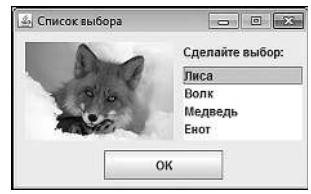


Рис. 15.11. Окно со списком выбора и изображением лисы

Если выбрать другой пункт в списке, появится и другое изображение. На рис. 15.12 показано окно с изображением волка.

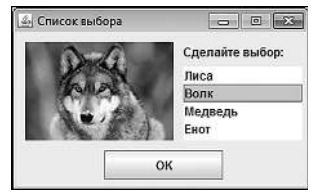


Рис. 15.12. Окно со списком выбора и изображением волка

Как выглядит окно с изображением медведя, показано на рис. 15.13.

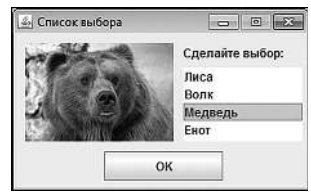


Рис. 15.13. Окно со списком выбора и изображением медведя

Наконец, на рис. 15.14 представлено окно с изображением енота.

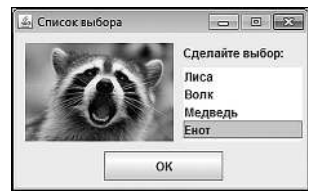


Рис. 15.14. Окно со списком выбора и изображением енота

Несложно заметить, что во всех случаях изображение, отображаемое в окне, соответствует пункту, выбранному в списке справа от изображения.

Группа переключателей

Бабу-Ягу со стороны брать не будем — воспитаем в своём коллективе.

Из к/ф «Карнавальная ночь»

Еще одним графическим компонентом, позволяющим выполнять выбор некоторой «позиции», характеристики или значения, является *группа переключателей*. Мы рассмотрим, как задача, аналогичная предыдущим, может быть реализована с использованием группы переключателей вместо списка или раскрывающегося списка. Но на сей раз начнем с результатов. На рис. 15.15 показано окно, отображаемое при выполнении программы.

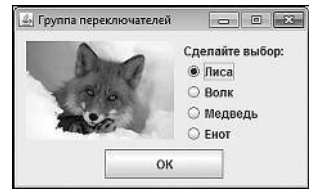


Рис. 15.15. Окно с группой переключателей отображается в начале выполнения программы

Окно, как видим, содержит уже традиционную метку с изображением, а справа от области метки размещена группа переключателей с названиями животных. Выбрать (или установить) можно только один переключатель из группы. Поэтому устанавливая какой-то определенный переключатель (из группы) мы автоматически отменяем установку текущего переключателя. На рис. 15.16 показано, как выглядит окно с изображением волка.

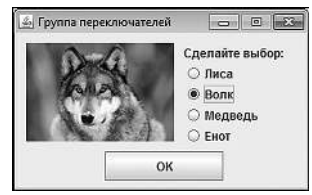


Рис. 15.16. Окно с группой переключателей содержит изображение волка

Кроме изображения, изменилось и состояние переключателей. На рис. 15.17 показано окно с изображением медведя.

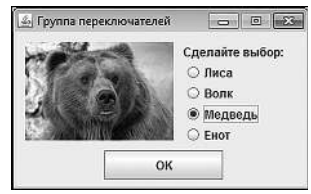


Рис. 15.17. Окно с группой переключателей содержит изображение медведя

Как выглядит окно с изображением енота, показано на рис. 15.18.

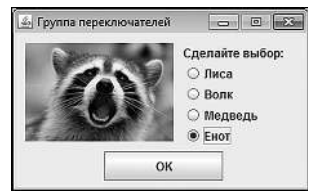


Рис. 15.18. Окно с группой переключателей содержит изображение волка

Теперь рассмотрим программный код, выполнение которого приводит к означенным результатам. Код представлен в листинге 15.4.

Листинг 15.4. Программный код проекта UsingRadioButtonApplication

```
// Импорт классов:
import javax.swing.*;
import java.awt.event.*;
// Класс для создания объекта окна реализует
// интерфейс ItemListener:
class MyFrame extends JFrame implements ItemListener{
  // Массив переключателей:
  private JRadioButton[] R;
  // Кнопка:
  private JButton B;
  // Метка:
  private JLabel L;
```

```
  // Массив с названиями животных:
  private String[] animals=new String[]{"Лиса","Волк","Медведь","Енот"};
  // Массив с названиями файлов с изображениями:
  private String[] files=new String[]{"fox.jpg","wolf.jpg","bear.jpg","raccoon.jpg"};
  // Путь к каталогу с изображениями:
  private String path="d:/books/pictures/";
  // Массив с изображениями:
  private ImageIcon[] imgs;
  // Метод вызывается при изменении
  // состояния переключателя:
  public void itemStateChanged(ItemEvent e){
    // Путем перебора и проверки состояния переключателей
    // определяется изображение для метки:
    for(int k=0;k<R.length;k++){
      // Если переключатель установлен:
      if(R[k].isSelected()){
        // Новое изображение для метки:
        L.setIcon(imgs[k]);
        // Завершение оператора цикла:
        break;
      }
    }
  }
  // Конструктор:
  MyFrame(){
    // Настройка параметров окна:
    super("Группа переключателей");
    setBounds(250,250,300,185);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setResizable(false);
    setLayout(null);
    // Создание массива из ссылок на объекты изображений:
```

```
imgs=new ImageIcon[files.length];
// Создание объектов изображений
// и заполнение массива:
for(int k=0;k<imgs.length;k++){
  imgs[k]=new ImageIcon(path+files[k]);
}
// Создание метки:
L=new JLabel(imgs[0]);
// Положение и размеры метки:
L.setBounds(10,10,150,100);
// Добавление метки в окно:
add(L);
// Создание и добавление в окно метки с текстом:
JLabel lbl=new JLabel("Сделайте выбор:");
lbl.setBounds(170,10,120,20);
add(lbl);
// Создание объекта для группы переключателей:
ButtonGroup BG=new ButtonGroup();
// Создание массива переключателей:
R=new JRadioButton[animals.length];
// Создание объектов переключателей и выполнение
// настроек:
for(int k=0;k<R.length;k++){
  // Создание объекта переключателя:
  R[k]=new JRadioButton(animals[k]);
  // Добавление переключателя в группу:
  BG.add(R[k]);
  // Положение и размеры переключателя:
  R[k].setBounds(170,31+21*k,120,20);
  // Регистрация в переключателе обработчика:
  R[k].addItemListener(this);
  // Добавление переключателя в окно:
```

```
      add(R[k]);
    }
    // Устанавливается первый переключатель:
    R[0].setSelected(true);
    // Создание кнопки:
    B=new JButton("OK");
    // Положение и размеры кнопки:
    B.setBounds(90,120,120,30);
    // Регистрация обработчика в кнопке:
    B.addActionListener(e->System.exit(0));
    // Добавление кнопки в окно:
    add(B);
    // Отображение окна на экране:
    setVisible(true);
  }
}
// Главный класс:
class UsingRadioButtonDemo{
  // Главный метод:
  public static void main(String[] args){
    // Создание объекта окна:
    new MyFrame();
  }
}
```

Один (отдельный) переключатель реализуется через объект класса JRadioButton. Мы используем механизм создания переключателей, при котором аргументом конструктору передается текст, отображаемый в области переключателя. Но мало создать отдельные переключатели — их нужно объединить в *группу*. Дело в том, что переключатель — компонент «коллективный». Специфика переключателя в том, что может быть установлен только один переключатель в группе. С другой стороны, окно или панель может содержать несколько групп переключателей. Следовательно, необходимо обозначить, какой переключатель к какой

группе относится. Поэтому при использовании переключателей кроме создания собственно переключателей еще создается объект для группы переключателей. Объект группы создается на основе класса ButtonGroup. Переключатели к группе добавляются с помощью метода add(), который вызывается из объекта группы, а аргументом методу передается ссылка на объект переключателя. Следует понимать, что «присоединение» переключателя к группе не влияет на его «пространственное» положение. Теоретически переключатели из одной группы могут находиться в разных частях контейнера. Другими словами, добавляя переключатель в группу, мы обозначаем, что определенные переключатели (относящихся к группе) взаимосвязаны в том плане, что среди них один и только один может быть установлен.

Что касается собственно программного кода, то в классе MyFrame объявлено закрытое поле R, представляющее собой массив ссылок на объекты класса JRadioButton. В конструкторе класса, кроме прочих знакомых по предыдущим примерам команд, инструкцией ButtonGroup BG=new ButtonGroup() создается объект BG для группы переключателей. Затем командой R=new JRadioButton[animals.length] создается массив из объектных ссылок класса JRadioButton. Размер массива определяется по количеству элементов в массиве animals, содержащем названия животных. После создания массива запускается оператор цикла, в котором с помощью индексной переменной k перебираются элементы массива R. За каждый цикл командой R[k]=new JRadioButton(animals[k]) создается объект для очередного переключателя, а текст при этом используется из массива с названиями животных. После создания объект переключателя с помощью команды BG.add(R[k]) добавляется в группу. Затем командой R[k].setBounds(170,31+21*k,120,20) переключатель добавляется в окно. Второй аргумент метода setBounds(), определяющий вертикальную координату переключателя в области окна, линейно зависит от индексной переменной k, благодаря чему переключатели размещаются один под другим с одинаковым отступом вдоль вертикали. Обработчиком события класса ItemEvent в переключателе регистрируется объект окна, для чего использована команда R[k].addItemListener(this). Наконец, командой add(R[k]) переключатель добавляется в окно. Уже после завершения оператора цикла с помощью инструкции R[0].setSelected(true) мы для установки выбираем первый переключатель. Поэтому при отображении окна черная точка отображается у первого (самого верхнего) переключателя.

Поскольку класс MyFrame реализует интерфейс ItemListener, то в классе описан метод itemStateChanged() с аргументом, являющимся ссылкой на объект

события класса ItemEvent. Метод вызывается при изменении состояния переключателя, в котором зарегистрирован соответствующий обработчик.

В теле метода запускается оператор цикла, в котором последовательно перебираются все переключатели, на которые есть ссылки в массиве R. В теле оператора цикла в условном операторе проверяется значение выражения R[k].isSelected(). Результатом выражения является значение true, если соответствующий переключатель установлен, и false в противном случае. Если переключатель установлен, то командой L.setIcon(imgs[k]) для метки применяется изображение из массива imgs с таким же индексом, как и индекс установленного переключателя из массива R. Поскольку далее нет смысла проверять состояние переключателей, то инструкцией break прекращается выполнение оператора цикла.

НА ЗАМЕТКУ

В данном случае для переключателей выполняется обработка события класса ItemEvent. Теоретически вместо этого можно было бы выполнить обработку, например, для события ActionEvent. В таком случае переключатель реагировал бы на щелчок мышью, но возникли бы проблемы с установкой и отменой установки для переключателей с помощью клавиатуры.

Опции и другие элементы

Докладчик сделает доклад. Коротенько так, минут на сорок, больше, я думаю, не надо...

Из к/ф «Карнавальная ночь»

Теперь мы рассмотрим еще одну версию программы, в которой с помощью нескольких графических компонентов в окне изменяется изображение. То есть общая идея остается той же, но реализация существенно меняется. Как минимум мы познакомимся с несколькими новыми компонентами, среди которых *опция* (реализуется через объект класса JCheckBox), *спиннер* (реализуется через объект класса JSpinner), *слайдер* (реализуется через объект класса JSlider), *панель с вкладками* (реализуется через объект класса JTabbedPane) и *опционная кнопка* (реализуется через объект класса JToggleButton). Также мы будем иметь дело с некоторыми нам незнакомыми пока что утилитами, которые имеют опосредованное отношение к графическим компонентам, но без которых сложно обойтись.

Прежде чем приступить к анализу программного кода, разумно кратко описать, каковы же результаты выполнения программы. Так будет легче понять смысл некоторых блоков кода.

Итак, при запуске программы на выполнение появляется окно, представленное на рис. 15.19.

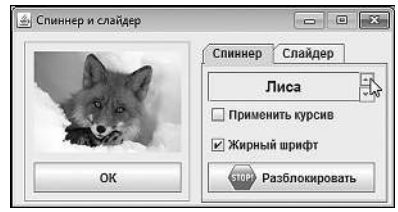


Рис. 15.19. Вид окна, отображаемого при выполнении программы

В левой части окна размещена метка с изображением лисы, под изображением находится кнопка **ОК**. Эти два компонента находятся не непосредственно в окне, а на панели, которая в свою очередь добавлена в окно. Границы панели выделены рамкой.

В правой части окна находится панель с двумя вкладками. На корешке одной вкладки отображается название **Спиннер**, а на корешке второй вкладки отображается название **Слайдер**. По умолчанию открыта первая вкладка. На этой вкладке в верхней части отображается спиннер: поле со значением **Лиса** и двумя маленькими пиктограммами со стрелками в правой части. Внизу есть две опции **Применить курсив** и **Жирный шрифт**. Для второй опции установлен флажок. Внизу на вкладке есть кнопка с названием **Разблокировать** и пиктограммой (красный шестиугольник). Кнопка опционная — при щелчке на кнопке она переходит в нажатое состояние, а при повторном щелчке возвращается в исходное состояние.

При щелчке на пиктограмме спиннера со стрелкой вверх в поле спиннера появляется называние **Волк** и соответствующим образом меняется изображение в окне. Ситуацию иллюстрирует рис. 15.20.

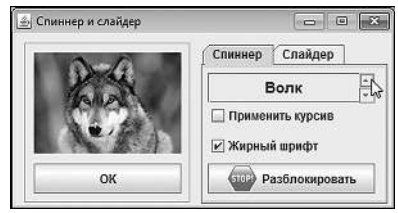


Рис. 15.20. Щелкая пиктограммы со стрелками в спиннере, меняем изображение в окне

Во второй вкладке с названием **Слайдер** находится слайдер: вертикальная «линейка» с ползунком, который может перемещаться вдоль вертикали и находится в одном из четырех положений (каждое положение подписано названием животного). Более того, изменение значение спиннера во вкладке **Спиннер** приводит к автоматическому синхронному изменению положения ползунка слайдера во вкладке **Слайдер**. На рис. 15.21 показано положение ползунка слайдера при условии, что в спиннере установлено значение **Волк**.

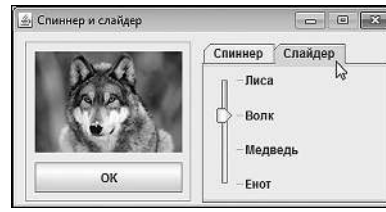


Рис. 15.21. С изменением состояния спиннера автоматически изменяется состояние слайдера во второй вкладке панели

Если изменить положение ползунка, то в соответствии с его новой позицией отображается изображение в метке. На рис. 15.22 ползунок слайдера установлен в позиции **Медведь**, и в окне отображается изображение медведя.

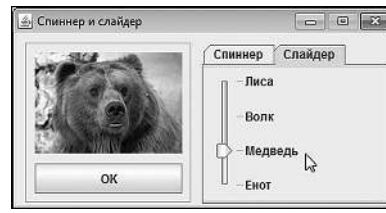


Рис. 15.22. Изменение изображения в окне перемещением ползунка слайдера

Если теперь перейти к вкладке **Спиннер**, то в спиннере также будет установлено значение **Медведь**, как это показано на рис. 15.23.

Опционная кнопка **Разблокировать** внизу вкладки **Спиннер**, как отмечалось, может находиться в двух состояниях: нажатом и не нажатом. В нажатом состоянии название кнопки изменяется на **Заблокировать**, и у кнопки также изменяется пиктограмма (зеленый прямоугольник). Но самое важное — это то, что при нажатой опционной кнопке содержимое в поле спиннера можно редактировать.

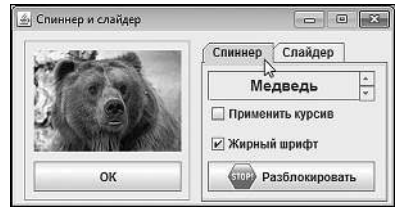


Рис. 15.23. При перемещении ползунка слайдера в первой вкладке синхронно изменяется значение в поле спиннера

На рис. 15.24 показано окно с открытой первой вкладкой, в которой нажата опционная кнопка, а в поле спиннера выделен текст (являющийся значением спиннера).

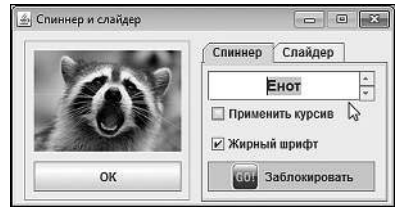


Рис. 15.24. При нажатой опционной кнопке на ней меняется текст и изображение, а поле спиннера становится доступным для редактирования

В таком режиме (когда можно редактировать содержимое поля спиннера), кроме изменения значения спиннера с помощью пиктограмм со стрелками, новое значение можно ввести непосредственно с клавиатуры. Правда, это значение должно совпадать с одним из допустимых (**Лиса**, **Волк**, **Медведь** или **Енот**), иначе изменения значения в поле спиннера не произойдет. Если еще раз нажать опционную кнопку, она вернется в исходное (не нажатое) состояние (с прежним названием и прежней пиктограммой). С помощью опций во вкладке **Спиннер** можно применять или отменять для поля спиннера эффект курсивного и/или жирного шрифта. На рис. 15.25 показано окно с открытой вкладкой **Спиннер**, в которой установлены флажки для опций **Применить курсив** и **Жирный шрифт**.

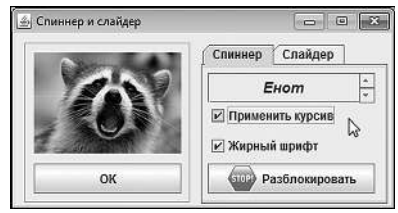


Рис. 15.25. Применение к тексту в поле спиннера жирного курсивного шрифта

Как следствие текст в поле спиннера отображается жирным курсивным шрифтом. Если оставить флажок только для опции **Применить курсив**, то шрифт будет курсивным, но не жирным, как показано на рис. 15.26.

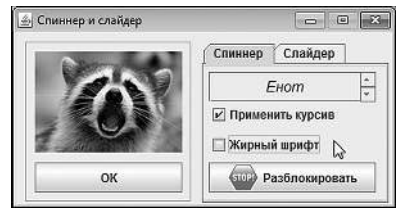


Рис. 15.26. Применение к тексту в поле спиннера только курсивного шрифта

На рис. 15.27 показана ситуация, когда отменены флажки обеих опций.

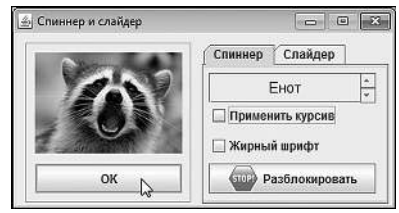


Рис. 15.27. Применение к тексту в поле спиннера обычного (не жирного и не курсивного) шрифта

В результате в поле спиннера название животного (в данном случае это **Енот**) отображается обычным (не жирным и не курсивным) шрифтом.

Наконец, щелчок на кнопке **OK** приводит к закрытию окна и завершению выполнения программы.

Теперь нам важно разобраться, как организован программный код. Вначале сделаем несколько общих замечаний. В первую очередь следует отметить, что основу программы составляет класс MyFrame, наследующий класс JFrame и реализующий интерфейсы ItemListener и ChangeListener. У класса несколько закрытых полей. Помимо уже известных нам по предыдущих примерах, имеется:

- поле SP, являющееся ссылкой на объект класса JSpinner (спиннер);
- поле SL, являющееся ссылкой на объект класса JSlider (слайдер);
- поле TB, представляющее собой ссылку на объект класса JToggleButton (опционная кнопка);

- два поля IT и BL для выполнения ссылки на объекты опций (создаются на основе класса JCheckBox);
- а также ряд вспомогательных полей и методов, которые мы обсудим по мере анализа кода.

Схема реализации программы такая. В конструкторе класса MyFrame создается панель (обычная), на нее добавляется метка с изображением и кнопка, щелчок на которой приводить к завершению выполнения программы. Эта панель помещается в окно. Затем создается панель с вкладками. Панель с вкладками реализуется через объект класса JTabbedPane. Для каждой вкладки (а их всего две) создается панель (обычная). На первую панель добавляется спиннер, две опции и опционная кнопка. Панель добавляется на первую вкладку панели с вкладками. Затем создается еще одна панель (обычная), на ней размещается слайдер, и данная панель добавляется второй вкладкой на панель с вкладками.

Обработка событий реализуется для кнопки, завершающей выполнение программы, спиннера, слайдера, двух опций и опционной кнопки. Для обычной и опционной кнопок обрабатываются события класса ActionEvent. Для каждой из кнопок регистрация обработчика выполняется с использованием лямбда-выражений. Для опций обрабатываются события класса ItemEvent, связанные с изменением состояния компонента. Соответственно, в классе MyFrame описывается метод itemStateChanged(). Для спиннера и слайдера используется один общий обработчик для события класса ChangeEvent. Отсюда в классе MyFrame имеется описание метода stateChanged(). Обработчиком как для опций, так и для слайдера со спиннером регистрируется объект окна.

Принцип обработки событий такой. Для обычной кнопки при щелчке на ней командой System.exit(0) завершается выполнение программы. При щелчке на опционной кнопке проверяется ее состояние (нажата, или не нажата). В каждом из этих случаев указывается название кнопки (названия кнопки в нажатом и не нажатом состояниях определяются значениями полей pressed и unpressed), а также определяется режим редактирования значения в поле спиннера (можно редактировать или нельзя). При изменении состояния любой из опций проверяются значения (установлена или не установлена) каждой опции, в соответствии с состоянием опций определяется стиль шрифта, и затем нужный шрифт применяется к полю спиннера. Для вычисления и применения шрифта для спиннера в классе MyFrame описан закрытый метод setSpinnerFont(). Наконец, обработчик для спиннера и слайдера реализует такой алгоритм обработки.

На основе объекта события определяется компонент, на котором произошло событие. В соответствии со значением, установленным в этом компоненте, задается значение для другого компонента и в области метки меняется изображение.

Теперь обратимся к программному коду, представленному в листинге 15.5.

Листинг 15.5. Программный код проекта SpinnerAndSliderApplication

```
// Импорт классов:
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
// Класс для создания объекта окна реализует интерфейсы
// ItemListener и ChangeListener:
class MyFrame extends JFrame implements ItemListener,ChangeListener{
  // Спиннер:
  private JSpinner SP;
  // Ссылка на объект редактора для спиннера:
  private JSpinner.DefaultEditor editor;
  // Название для шрифта спиннера:
  private String name="Arial";
  // Размер шрифта спиннера:
  private int size=15;
  // Слайдер:
  private JSlider SL;
  // Опционная кнопка:
  private JToggleButton TB;
  // Названия для опционной кнопки:
  private String pressed="Заблокировать";
  private String unpressed="Разблокировать";
  // Ссылки на пиктограммы для опционной кнопки:
  private ImageIcon locked,unlocked;
```

```
// Опции:
private JCheckBox IT,BL;
// Кнопка:
private JButton B;
// Метка:
private JLabel L;
// Массив с названиями животных:
private String[] animals=new String[]{"Лиса","Волк","Медведь","Енот"};
// Массив с названиями файлов с изображениями:
private String[] files=new String[]{"fox.jpg","wolf.jpg","bear.jpg","raccoon.jpg"};
// Путь к каталогу с изображениями:
private String path="d:/books/pictures/";
// Массив с изображениями:
private ImageIcon[] imgs;
// Закрытый метод для определения шрифта спиннера:
private void setSpinnerFont(){
  // Переменная для определения стиля шрифта:
  int style=Font.PLAIN;
  // Если установлена опция для применения курсива:
  if(IT.isSelected()){
    style|=Font.ITALIC;
  }
  // Если установлена опция для применения
  // жирного шрифта:
  if(BL.isSelected()){
    style|=Font.BOLD;
  }
  // Применение шрифта к полю спиннера:
  editor.getTextField().setFont(new Font(name,style,size));
}
// Метод вызывается при изменении состояния опций:
public void itemStateChanged(ItemEvent e){
  // Определение шрифта для спиннера:
```

```
    setSpinnerFont();
  }
  // Метод вызывается для обработки событий, связанных
  // с изменением состояния спиннера и слайдера:
  public void stateChanged(ChangeEvent e){
    // Локальная целочисленная переменная:
    int k;
    // Локальная текстовая переменная:
    String s;
    // Если событие произошло на спиннере:
    if(e.getSource()==SP){
      // Считывание значения в поле спиннера:
      s=(String)SP.getValue();
      // Определение индекса выбранного пункта:
      for(k=0;k<animals.length;k++){
        if(animals[k].equals(s)){
          // Присваивание значения слайдеру:
          SL.setValue(k);
          // Завершение оператора цикла:
          break;
        }
      }
    }
    else{ // Если событие произошло на слайдере:
      k=SL.getValue();
      // Присваивание значения для спиннера:
      SP.setValue(animals[k]);
    }
    // Изменение изображения:
    L.setIcon(imgs[k]);
  }
  // Конструктор:
  MyFrame(){
```

```
// Настройка параметров окна:
super("Спиннер и слайдер");
setBounds(250,250,390,200);
setDefaultCloseOperation(EXIT_ON_CLOSE);
setResizable(false);
setLayout(null);
// Создание панели:
JPanel pnl=new JPanel();
// Отключение менеджера компоновки панели:
pnl.setLayout(null);
// Положение и размеры панели:
pnl.setBounds(10,10,170,160);
// Рамка вокруг панели:
pnl.setBorder(BorderFactory.createEtchedBorder());
// Создание массива из ссылок на объекты изображений:
imgs=new ImageIcon[files.length];
// Создание объектов изображений
// и заполнение массива:
for(int k=0;k<imgs.length;k++){
  imgs[k]=new ImageIcon(path+files[k]);
}
// Создание метки с изображением:
L=new JLabel(imgs[0]);
// Положение и размеры метки:
L.setBounds(10,10,150,100);
// Добавление метки на панель:
pnl.add(L);
// Создание кнопки:
B=new JButton("OK");
// Положение и размеры кнопки:
B.setBounds(10,120,150,30);
// Регистрация обработчика в кнопке:
B.addActionListener(e->System.exit(0));
```

```
// Добавление кнопки на панель:
pnl.add(B);
// Добавление панели в окно:
add(pnl);
// Создание панели с вкладками:
JTabbedPane tp=new JTabbedPane();
// Положение и размеры панели с вкладками:
tp.setBounds(190,10,190,160);
// Создание панели:
JPanel one=new JPanel();
// Отключение менеджера компоновки панели:
one.setLayout(null);
// Создание объекта модели спиннера:
SpinnerModel sm=new SpinnerListModel(animals);
// Создание объекта спиннера:
SP=new JSpinner(sm);
// Положение и размеры спиннера:
SP.setBounds(5,5,170,30);
// Получение ссылки на объект редактора для спиннера:
editor=(JSpinner.DefaultEditor)SP.getEditor();
// Выравнивание содержимого поля спиннера по центру:
editor.getTextField().setHorizontalAlignment(JTextField.CENTER);
// Переход в режим запрета редактирования
// содержимого поля спиннера:
editor.getTextField().setEditable(false);
// Устанавливается значение спиннера:
SP.setValue(animals[0]);
// Регистрация обработчика для спиннера:
SP.addChangeListener(this);
// Добавление спиннера на панель:
one.add(SP);
// Создание опции (применение курсива):
IT=new JCheckBox("Применить курсив");
```

```
// Положение и размеры опции:
IT.setBounds(5,35,170,25);
// Регистрация обработчика в опции:
IT.addItemListener(this);
// Добавление опции (применение курсива) на панель:
one.add(IT);
// Создание опции (применение жирного шрифта):
BL=new JCheckBox("Жирный шрифт");
// Положение и размеры опции:
BL.setBounds(5,65,170,25);
// Установка флажка для опции:
BL.setSelected(true);
// Регистрация обработчика:
BL.addItemListener(this);
// Добавление опции на панель:
one.add(BL);
// Создание опционной кнопки:
TB=new JToggleButton(unpressed);
// Положение и размеры опционной кнопки:
TB.setBounds(5,95,170,30);
// Создание объектов изображений
// для опционной кнопки:
locked=new ImageIcon(path+"locked.png");
unlocked=new ImageIcon(path+"unlocked.png");
// Пиктограмма для опционной кнопки
// (если кнопка не нажата):
TB.setIcon(locked);
// Пиктограмма для опционной кнопки
// (если кнопка нажата):
TB.setSelectedIcon(unlocked);
// Регистрация обработчика для опционной кнопки:
TB.addActionListener(e->{
  // Если опционная кнопка нажата:
```

```
    if(TB.isSelected()){
      // Текст для кнопки:
      TB.setText(pressed);
      // Режим, разрешающий редактирование
      // содержимого в поле спиннера:
      editor.getTextField().setEditable(true);
    } // Если опционная кнопка не нажата:
    else{
      // Текст для кнопки:
      TB.setText(unpressed);
      // Режим, запрещающий редактирование
      // содержимого в поле спиннера:
      editor.getTextField().setEditable(false);
    }
  });
  // Добавление кнопки на панель:
  one.add(TB);
  // Применение шрифта для спиннера:
  setSpinnerFont();
  // Добавление панели на панель с вкладками
  // (в качестве первой вкладки):
  tp.add("Спиннер",one);
  // Создание панели:
  JPanel two=new JPanel();
  // Отключение менеджера компоновки панели:
  two.setLayout(null);
  // Создание слайдера:
  SL=new JSlider(JSlider.VERTICAL,0,3,1);
  // Положение и размеры слайдера:
  SL.setBounds(5,5,100,120);
  // Создание объекта для реализации
  // таблицы подстановок подписей для слайдера:
  Hashtable ht=new Hashtable();
```

```
    // Заполнение таблицы подстановок:
    for(int k=0;k<animals.length;k++){
      ht.put(new Integer(k),new JLabel(animals[k]));
    }
    // Применение таблицы подстановок для подписей:
    SL.setLabelTable(ht);
    // Инверсный способ отображения шкалы для слайдера:
    SL.setInverted(true);
    // Интервал между основными засечками:
    SL.setMajorTickSpacing(1);
    // Отображение засечек:
    SL.setPaintTicks(true);
    // Отображение подписей:
    SL.setPaintLabels(true);
    // Установка значения для слайдера:
    SL.setValue(0);
    // Регистрация обработчика для слайдера:
    SL.addChangeListener(this);
    // Добавление слайдера на панель:
    two.add(SL);
    // Добавление панели на панель с вкладками
    // (в качестве второй вкладки):
    tp.add("Слайдер",two);
    // Добавление панели с вкладками в окно:
    add(tp);
    // Отображение окна на экране:
    setVisible(true);
  }
}
// Главный класс:
class SpinnerAndSliderDemo{
  // Главный метод:
  public static void main(String[] args){
```

```
    // Создание объекта окна:
    new MyFrame();
  }
}
```

Прокомментируем наиболее сложные и важные фрагменты программного кода (прочие блоки кода или похожие на них уже встречались нам ранее и должны быть понятны читателю).

В конструкторе класса командой JPanel pnl=new JPanel() создается панель для последующего размещения на ней метки и кнопки **ОК**. То есть в данном случае мы метку с изображением и кнопку размещаем не непосредственно в окне, а на панели, которая затем размещается в окне. Поэтому когда для кнопки и метки определяются координаты, то это координаты внутри панели. Координаты панели, в свою очередь, задаются внутри окна. Также стоит заметить, что поскольку в панели компоненты размещаются путем явного указания координат, то командой pnl.setLayout(null) для панели отключается менеджер компоновки.

Еще одна панель, но уже с вкладками, создается командой JTabbedPane tp= new JTabbedPane(). Каждая вкладка данной панели создается отдельно. Мы используем подход, при котором для вкладки формируется панель (обычная) с компонентами, после чего панель добавляется во вкладку.

Панель для первой вкладки создается командой JPanel one=new JPanel(). Для нее отключается менеджер компоновки (команда one.setLayout(null)). Размеры и координаты для панели мы не указываем, поскольку при добавлении во вкладку панель автоматически размещается и масштабируется в соответствии с размерами объекта td.

При создании спиннера необходимо задать его *модель*. Модель спиннера — это объект, определяющий «поведение» спиннера. Мы используем одну из стандартных моделей и создаем объект sm модели спиннера на основе класса SpinnerModel (команда SpinnerModel sm=new SpinnerListModel(animals)). Аргументом конструктору класса SpinnerListModel передается текстовый массив animals. Поэтому спиннер, созданный на основе модели, определяемой объектом sm, в качестве значений может принимать текстовые значения из массива animals. Собственно спиннер создается командой SP=new JSpinner(sm). Аргументом конструктору класса JSpinner передан объект sm модели спиннера. Командой SP.setBounds(5,5,170,30) определяется положение и размеры спиннера. Поскольку спиннер добавляется

на панель one, то речь, очевидно, идет о координатах в области панели. Также для решения ряда «тактических» задач нам нужен доступ к полю спиннера (поле, в котором отображается значение спиннера). Для этого выполняется команда editor=(JSpinner.DefaultEditor)SP.getEditor(), которой в поле editor записывается ссылка на объект редактора для спиннера. Это объект, который «отвечает» за выполнение операций по отображению спиннера. Поле editor в классе MyFrame объявлено как закрытое поле типа JSpinner.DefaultEditor. Класс DefaultEditor является внутренним классом в классе JSpinner. Данный класс (имеется в виду DefaultEditor) является суперклассом для других классов, через которые реализуются объекты-редакторы. Поэтому гарантированно переменная типа JSpinner.DefaultEditor может ссылаться на объекты-редакторы производных классов. Этим обстоятельством мы и воспользовались. Что касается команды editor=(JSpinner.DefaultEditor)SP.getEditor(), то в ней из объекта спиннера SP вызывается метод getEditor(), который возвращает ссылку класса JComponent на объект редактора для спиннера. Класс JComponent является суперклассом для классов большинства «легких» компонентов, поэтому в переменную такого типа можно записать ссылку на объект практически любого компонента. Но мы в силу необходимости «сужаем» типизацию для полученной ссылки на объект-редактор и выполняем процедуру приведения к типу JSpinner.DefaultEditor, для чего данный тип в круглых скобках указан перед выражением, возвращающим результатом ссылку на редактор.

В команде editor.getTextField().setHorizontalAlignment(JTextField.CENTER) ссылка на объект-редактор используется для применения режима выравнивания по центру для поля спиннера (по умолчанию содержимое поля спиннера выравнивается по правому краю). В данном случае из объекта редактора вызывается метод getTextField(). Результатом является ссылка на поле спиннера. Из этой ссылки вызывается метод setHorizontalAlignment(), используемый для определения способа выравнивания текста в поле. Переданная аргументом методу статическая константа CENTER из класса JTextField означает, что выравнивание выполняется по центру.

Командой editor.getTextField().setEditable(false) поле спиннера переводится в режим, при котором нельзя редактировать значение в поле. Командой SP.setValue(animals[0]) значением спиннера устанавливается первое (элемент с нулевым индексом) текстовое значение из массива animals. Командой SP.addChangeListener(this) в спиннере обработчиком событий класса ChangeEvent регистрируется объект окна. Наконец, командой one.add(SP) спиннер добавляется на панель one.

Опции создаются командами IT=new JCheckBox("Применить курсив") и BL=new JCheckBox("Жирный шрифт"). Для них задаются положение и размеры, регистрируется обработчик (объект окна) событий класса ItemEvent, после чего опции добавляются на панель one.

> **НА ЗАМЕТКУ**
>
> По умолчанию для опций флажок не установлен. Мы с помощью команды BL.setSelected(true) переводим опцию BL в режим установленного флажка. Поэтому при отображении окна в начале выполнения программы у второй опции флажок установлен.

Опционная кнопка создается командой TB=new JToggleButton(unpressed). Аргументом конструктора указано текстовое поле unpressed с названием кнопки, когда она находится в не нажатом состоянии. Положение и размеры кнопки определяем командой TB.setBounds(5,95,170,30). Также с помощью инструкций locked=new ImageIcon(path+"locked.png") и unlocked=new ImageIcon(path+"unlocked.png") на основе заранее подготовленных изображений (файлы locked.png и unlocked.png в каталоге d:\books\pictures, путь к которому записан в текстовое поле path) создаются объекты для пиктограмм, отображаемых в области кнопки в не нажатом и нажатом состояниях. Затем командой TB.setIcon(locked) задается пиктограмма для кнопки, когда она в не нажатом состоянии. Пиктограмма для кнопки, когда она в нажатом состоянии, определяется командой TB.setSelectedIcon(unlocked). Регистрация обработчика события ActionEvent для кнопки выполняется с помощью лямбда-выражения. В теле метода, определяемого лямбда-выражением, проверяется условие TB.isSelected(). Значение данного выражения равно true, если кнопка нажата, и равно false если кнопка не нажата. И в том, и в другом случае вызываются одни и те же методы, но с разными аргументами. С помощью метода setEditable(), вызываемого из объекта поля спиннера (доступ к нему получаем через инструкцию editor.getTextField()) задается режим, позволяющий или запрещающий редактирование содержимого поля спиннера. С помощью метода setText() задается текст, отображаемый в области кнопки. В последнем случае мы используем текстовые поля pressed и unpressed.

Командой one.add(TB) опционная кнопка добавляется на панель one. Перед добавлением панели one в первую вкладку панели с вкладками tp, вызывается метод setSpinnerFont(). Этим методом «считывается» состояние опций и в соответствии с ними к полю спиннера применяется шрифт (подробнее код метода setSpinnerFont() анализируется позже).

Для добавления панели one в первую вкладку панели tp используем команду tp.add("Спиннер",one). Фактически, этой командой в панель tp добавляется вкладка, на корешке которой будет название "Спиннер", а собственно вкладка будет содержать панель one, со всеми ее компонентами.

Далее мы создаем еще одну панель two (команда JPanel two=new JPanel()) и отключаем для нее менеджер компоновки (команда two.setLayout(null)). Слайдер создается командой SL=new JSlider(JSlider.VERTICAL,0,3,1). Аргументы, переданные конструктору, означают следующее: статическая константа VERTICAL из класса JSlider определяет ориентирование слайдера вдоль вертикали (линейка слайдера размещена вертикально), значения 0 и 3 определяют диапазон возможных значений слайдера (имеются в виду значения, определяемые положением ползунка слайдера), а число 1 задает шаг дискретности в изменении позиции ползунка. Но нас такой вариант на самом деле не очень устраивает. Мы хотим, чтобы вместо целочисленных значений от 0 до 3 включительно вдоль линейки слайдера отображались названия животных из массива animals. Поэтому используем «подстановки»: вместо чисел в области слайдера будем использовать текстовые метки с названиями животных. Выполнить такую замену можно с помощью таблицы подстановки — специального объекта обобщенного класса Hashtable. Соответствующий объект ht создается командой Hashtable ht=new Hashtable(). После того, как объект создан, его необходимо заполнить (задать правила подстановок). Для этого запускается оператор цикла, в котором индексная переменная k пробегает значения от 0 до 3 (возможные значение слайдера). Для каждого значения k выполняется команда ht.put(new Integer(k),new JLabel(animals[k])). В результате в области слайдера вместо числовых значений будут отображаться метки с текстом из массива animals.

НА ЗАМЕТКУ

Поскольку речь идет об обобщенном классе Hashtable, то используются только ссылочные типы данных. Отсюда, например, использование класса-оболочки Integer.

После того как объект таблицы подстановок создан, он применяется к слайдеру с помощью команды SL.setLabelTable(ht).

Есть еще ряд настроек, которые выполняются для слайдера. Так, командой SL.setInverted(true) задается инверсный способ отображения шкалы для слайдера: по умолчанию начальное (минимальное) значение соответствует нижнему положению ползунка. Соответственно, если

не «инвертировать» линейку, то названия животных размещались бы снизу вверх, а так они размещаются сверху вниз.

Командой SL.setMajorTickSpacing(1) задается интервал между основными засечками на линейке слайдера (есть еще и дополнительные засечки, но мы их не используем). Чтобы отобразить засечки, используем команду SL.setPaintTicks(true), а для отображения подписей возле засечек задействована команда SL.setPaintLabels(true).

С помощью команды SL.setValue(0) ползунок слайдера устанавливается на значение 0, что в данном случае соответствует начальной (верхней) позиции на линейке слайдера (а значение 0, соответственно, при отображении подписи заменяется на текст Лиса).

Для слайдера обработчиком событий класса ChangeEvent регистрируется объект окна (команда SL.addChangeListener(this)). Командой two.add(SL) слайдер добавляется на панель, после чего с помощью команды tp.add("Слайдер",two) на основе панели формируется вторая вкладка панели tp. После этого командой add(tp) панель с вкладками добавляется в окно.

Так формируется внешний вид окна. Теперь проанализируем обработку событий. В первую очередь обратим внимание на закрытый метод setSpinnerFont(), используемый для определения шрифта для поля спиннера. Фактически речь идет об определении стиля шрифта, поскольку название и размер неизменны и определяются значениями закрытых полей name и size соответственно.

В теле метода объявляется локальная целочисленная переменная style с начальным значение Font.PLAIN (обычный стиль шрифта). Затем в условном операторе проверяется выражение IT.isSelected(), которое равно true если опция IT установлена. В таком случае выполняется команда style|=Font.ITALIC (аналог команды style=style|Font.ITALIC). Аналогично все происходит со второй опцией: если условие BL.isSelected() равно true (опция установлена), то выполняется команда style|=Font.BOLD. Здесь мы учитываем, что специфика статических констант PLAIN, ITALIC и BOLD такова, что результат выражения Font.PLAIN|Font.ITALIC определяет курсивный шрифт, результат выражения Font.PLAIN|Font.BOLD определяет жирный шрифт, а результат выражения Font.ITALIC|Font.BOLD определяет жирный курсивный шрифт.

После определения стиля для шрифта командой editor.getTextField().setFont(new Font(name,style,size)) для поля спиннера задается шрифт.

Соответственно, в теле метода itemStateChanged(), вызываемого при обработке событий, связанных с изменением состояния опции (любой из

двух) вызывается метод setSpinnerFont(). Метод «сканирует» состояние опций, и в соответствии с результатами «проверки» определяет шрифт для поля спиннера.

Метод stateChanged() вызывается для обработки событий, связанных с изменением состояния спиннера или слайдера. В теле метода объявлена целочисленная переменная k и текстовая переменная s. В условном операторе проверяется условие e.getSource()==SP. Истинность условие означает, что событие возникло на спиннере. В таком случае командой s=(String)SP.getValue() в спиннере считывается значение и записывается в переменную s.

ДЕТАЛИ

Метод getValue(), вызываемый из объекта спиннера SP, возвращает результатом ссылку класса Object на значение, отображаемое в поле спиннера. Но мы знаем, что на самом деле в поле спиннера отображается текстовое значение. Поэтому используем процедуру приведения к текстовому типу и размещаем перед выражением SP.getValue() инструкцию (String).

После того как значение спиннера определено, мы вычисляем индекс соответствующего значения в массиве animals. Для этого запускается оператор цикла, в котором переменная k пробегает значения индексов элементов из массива animals. Проверяя условие animals[k].equals(s), мы ищем совпадение между значением элемента из массива и значением, прочитанным в поле спиннера. Как только совпадение найдено, командой SL.setValue(k) соответствующим образом изменяется положение ползунка в слайдере и с помощью инструкции break выполнение оператора цикла прекращается. Напомним, что все это происходит, если событие произошло на спиннере. Если событие произошло на слайдере (данный случай обрабатывается в else-блоке условного оператора), то командой k=SL.getValue() в переменную k записывается значение слайдера, после чего командой SP.setValue(animals[k]) изменяется значение спиннера. Здесь важно отметить два обстоятельства. Во-первых, хотя в области слайдера подписями отображаются названия животных, это не есть значения слайдера. Возможные значения слайдера — целые числа в диапазоне от 0 до 3. Названия животных лишь «подстановки» при отображении подписей. Во-вторых, значения слайдера и способ подстановок при отображении подписей подобраны так, что каждое числовое значение слайдера соответствует индексу элемента в массиве animals, который имеет такое же название, как и соответствующая подпись на слайдере. В итоге, на каком

бы компоненте (спиннер или слайдер) ни произошло событие, переменная k получает значением индекс элемента из массива animals, и этот элемент определяет название животного, которое «выбрано».

После выполнения условного оператора командой L.setIcon(imgs[k]) в текстовой метке, в соответствии со значением индекса k, изменяется изображение.

ДЕТАЛИ

Есть один момент, на который стоит обратить внимание. Дело в том, что при изменении состояния, например, спиннера выполняется обработка события, которая подразумевает изменение состояния слайдера. Изменение состояния слайдера приводит к тому, что выполняется обработка события, подразумевающая изменение состояния спиннера. Эти последовательные «обработки» конечны, поскольку в итоге состояние слайдера и спиннера перестанет изменяться, но все равно команда применения к метке нового изображения выполняется на самом деле несколько раз подряд. В этом смысле можно было бы команду L.setIcon(imgs[k]) разместить не после условного оператора, а в его if-ветке. Тогда при изменении состояния слайдера изменение изображения в метке происходило бы не «напрямую», а через вызов обработчика при изменении состояния спиннера.

Резюме

— Почему прекратили передачу Шекспира?!
— По техническим причинам...

Из к/ф «Карнавальная ночь»

- Существует достаточно много графических компонентов, которые позволяют решать самые различные задачи, причем, как правило, несколькими способами.
- Большинство компонентов имеет набор методов со стандартными названиями, через которые задаются основные параметры компонента и выполняются настройки его функциональности.
- Выбор того или иного компонента для включения в графический интерфейс определяется постановкой решаемой задачи, особенностями компонента и обще структурой создаваемого приложения.

Глава 16
МЕНЮ И ПАНЕЛЬ ИНСТРУМЕНТОВ

Костюмы надо заменить, ноги изолировать.
Из к/ф «Карнавальная ночь»

В этой главе мы продолжим знакомиться с подходами, используемыми при создании приложений с графическим интерфейсом. В частности, мы узнаем, как в приложении создается меню и панель инструментов, а также познакомимся с некоторыми вспомогательными технологиями и утилитами.

Меню и панель инструментов

Далее мы рассмотрим пример с приложением, в котором используются такие компоненты интерфейса, как *меню* (или главное меню) и *панель инструментов*. Пункты меню размещаются на панели меню в верхней части окна (меню может быть только у контейнеров верхнего уровня, таких как JFrame или JApplet). Панель инструментов представляет собой панель с кнопками (как правило, без названий — только с пиктограммами), которая также обычно размещается в верхней части окна, хотя это и не обязательно. Прежде чем приступить к реализации этих компонентов на практике, рассмотрим кратко общие принципы их использования.

Использование меню

Такой компонент, как *меню*, знаком практически всем, кто имел дело с приложениями с графическим интерфейсом. Обычно в приложениях с меню в верхней части окна размещается специальная панель с названиями, каждое из которых можно раскрыть (щелчком мыши) и выбрать одну из команд, которые появляются в раскрывающемся списке.

Для реализации *панели меню* в программе создается объект класса JMenuBar. На панель меню размещаются *пункты меню* — это те названия, которые отображаются в верхней части окна приложения. Отдельные пункты меню реализуются через объекты класса JMenu.

НА ЗАМЕТКУ

Обычно отдельный пункт меню называют просто *меню*. Но это не очень удобно с «филологической» точки зрения. Поэтому мы в основном будем использовать термин *пункт меню*. При этом элементы, которые содержатся в каждом из пунктов меню, будем называть *командами меню*.

Каждый пункт меню содержит *команды меню*. При выборе той или иной команды меню выполняются некоторые действия (какие именно — определяется через механизм обработки событий). Обычные команды меню реализуются через объекты класса JMenuItem. Но кроме «обычных», существуют и «необычные» команды: имеются в виду *команды-опции* и *команды-переключатели*. Команды-опции содержат небольшое поле, в котором можно установить флажок, и, в общем и целом, напоминают опции, но только размещенные внутри пункта меню. Команды-переключатели аналогичны обычным переключателям, но находятся внутри пункта меню и отображаются при его раскрытии. Команды-опции (или опционные команды) реализуются с помощью объектов класса JCheckBoxMenuItem, а команды-переключатели реализуются с помощью объектов класса JRadioButtonMenuItem.

НА ЗАМЕТКУ

Собственно, кроме команд внутри пункта меню могут быть еще и *подменю* — это пункты меню, которые помещены внутрь других пунктов меню. Подменю, как и пункты меню, создаются на основе объектов класса JMenu. Просто пункт меню добавляется на панель меню, а подменю добавляется в пункт меню.

Кроме главного меню, нередко используют еще и *контекстное меню*. Обычно контекстное меню отображается в области компонента при щелчке на нем правой кнопкой мыши. Объект контекстного меню создается на основе класса JPopupMenu. Что касается «наполнения» контекстного меню, то команды для него создаются точно так же, как и команды для пунктов главного меню.

Панель инструментов

Еще один полезный и популярный компонент графического интерфейса — *панель инструментов*. На панели инструментов размещаются

кнопки, которые могут дублировать команды меню или иметь отдельную функциональную ценность. В принципе, панель меню можно «сконструировать» из стандартных средств, таких как панель и обычные кнопки, но в библиотеке Swing имеется специальный класс JToolBar, через объект которого реализуется панель инструментов. Кнопки для панели инструментов реализуют в виде объектов класса JButton, но, в отличие от обычных кнопок, обычно ограничиваются отображением в области кнопки лишь пиктограммы, без названия.

НА ЗАМЕТКУ

Поскольку у кнопки на панели инструментов нет отображаемого названия, а по рисунку угадать назначение кнопки не всегда просто, то используют полезный прием. Прием состоит в том, что при наведении курсора мыши на кнопку на панели инструментов автоматически появляется текстовая подсказка для данной кнопки. Для добавления подсказки к компоненту используют метод setToolTipText(). Метод вызывается из объекта компонента, для которого создается интерактивная подсказка, а аргументом методу передается текст подсказки.

Менеджеры компоновки и текстовая панель

Хотя во всех предыдущих примерах мы традиционно отключали менеджер компоновки, на самом деле это достаточно полезный инструмент при создании приложений с графическим интерфейсом.

Менеджеры компоновки

В общем случае менеджер компоновки «отвечает» за размещение компонентов в контейнере — таком, как окно или панель. В зависимости от «типа» менеджера компоновки различаются алгоритмы и механизмы, в соответствии с которыми размещаются компоненты. Для использования какого-то конкретного менеджера компоновки необходимо создать объект менеджера компоновки и передать его аргументом методу setLayout(), который вызывается из объекта контейнера. Нас, в свете рассматриваемого далее примера, интересует два типа менеджеров. Один создается на основе класса BorderLayout. Если в контейнере используется данный менеджер, то компоненты могут помещаться в центр контейнера, слева, справа, сверху и внизу контейнера. Место размещения компонента в контейнере определяется одной из статических констант класса

BorderLayout (соответственно CENTER, WEST, EAST, NORTH и SOUTH). Константа указывается вторым аргументом метода add(), которым компонент добавляется в контейнер.

НА ЗАМЕТКУ

В окне по умолчанию менеджером компоновки используется менеджер класса BorderLayout. Тем не менее мы, чтобы подчеркнуть факт использования данного менеджера, будем в программном коде явно размещать инструкции по применению менеджера компоновки.

Менеджер компоновки, объект которого создается на основе класса GridLayout, размещает компоненты в контейнере по принципу «таблицы». Рабочая область контейнера условно разбивается на ячейки, и при добавлении компонентов эти ячейки последовательно заполняются (слева направо и сверху вниз). Количество строк и столбцов, на которые условно разбивается область контейнера, указываются аргументами конструктора класса GridLayout при создании объекта менеджера компоновки.

Текстовая панель

Для отображения относительно больших фрагментов текста могут использоваться такие компоненты, как *текстовые области* или *текстовые панели*. В рассматриваемом далее примере использована текстовая панель. Текстовая панель позволяет отображать, кроме прочего, содержимое текстового файла. Это некоторая область, в которой отображается текст из файла или просто заданный в программе текст (во всяком случае, так мы будем использовать текстовую панель).

Объект текстовой панели создается на основе класса JTextPane. Текст, отображаемый в области панели, в общем случае можно редактировать. За возможность или невозможность редактирования содержимого текстовой панели «отвечает» метод setEditable(). Также следует заметить, что во многих случаях область текстовой панели недостаточна для отображения всего текста, предназначенного для отображения. В таком случае имеет смысл использовать полосы прокрутки, которые позволяют перемещать (прокручивать) текст, содержащийся в области панели. Мы можем прибегнуть к помощи такого компонента, как панель с полосами прокрутки. Панель с полосами прокрутки реализуется в виде объекта класса JScrollPane. А уже затем в область отображения этого объекта

добавляется объект текстовой панели. Ссылку на область отображения панели с полосами прокрутки получают через метод getViewport(). Именно такой подход мы используем в рассматриваемом далее примере.

Использование меню и панели инструментов

Мы рассмотрим пример, в котором используется главное меню, контекстное, панель инструментов и некоторое вспомогательные компоненты и утилиты.

Постановка задачи

Мы традиционно рассмотрим задачу по отображению изображений разных животных (*лиса*, *волк*, *медведь* и *енот*). Но только на этот раз основные операции будут выполняться с помощью команд главного меню, кнопок на панели инструментов и команд контекстного меню. В частности, генеральный план состоит в том, чтобы создать окно, в котором будет панель меню и панель инструментов. В нижней части окна будет кнопка **ОК**, щелчок на которой приводит к завершению выполнения программы. В центральной части окна будет отображаться панель с изображением и текстом. Для «показа» изображения традиционно задействуем метку. А вот текст отображается в текстовой панели, причем исходное текстовое содержимое «спрятано» в файлах на диске.

Возвращаясь к меню, отметим, что планируется наличие на панели меню трех пунктов. В пункте меню **Содержание** представлены команды для выбора названия животного, изображение и описание которого должно отображаться в главной области окна. В пункте меню **Вид** представлены команды-переключатели для установки цвета, которым в окне выделяется панель с изображением и текстом. Наконец, в пункте меню Программа будет две команды, одна из которых позволяет закрыть окно (и завершить выполнение программы), а щелчок на другой приводит к отображению диалогового окна с информацией о проекте.

На панели инструментов будет представлено четыре кнопки, которые предназначены для выполнения таких операций: завершение выполнения программы, загрузка начального текста и изображения, переход к следующей или предыдущей картинке (и соответствующему ей тексту).

Для метки с изображением предусматривается контекстное меню, команды которого позволяют выбрать животное для отображения информации о нем, а также там будет команда завершения выполнения программы. В общих чертах таков наш план на программу.

Анализ возможностей программы

Поскольку анализировать программный код легче, если понятно, как функционирует программы, мы сначала рассмотрим основные, концептуальные возможности приложения, которое рассматривается в данном примере.

Итак, при запуске программы на выполнение отображается окно, представленное на рис. 16.1.

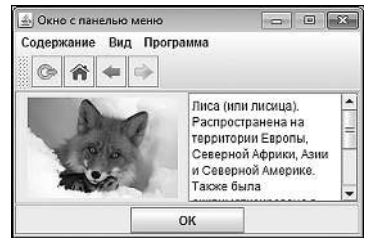


Рис. 16.1. Окно, которое отображается при запуске приложения на выполнение

В пункте меню **Содержание** есть четыре команды с названиями животных (**Лиса**, **Волк**, **Медведь** и **Енот**), как показано на рис. 16.2.

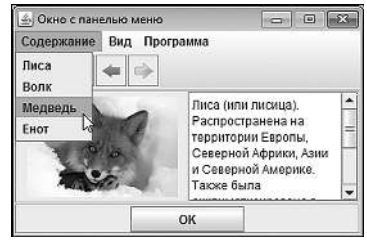


Рис. 16.2. Выбор команды **Медведь** в пункте меню **Содержание**

Если выбрать одну из команд, в центральной области окна откроется изображение соответствующего животного и небольшое описание справа от изображения. На рис. 16.3 показано, к какому результату приводит выбор в пункте меню **Содержание** команды **Медведь**.

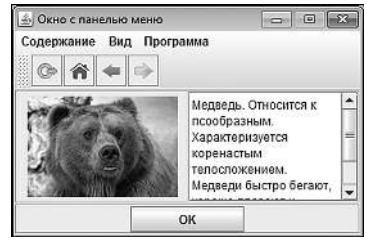


Рис. 16.3. Результат выбора команды **Медведь** в пункте меню **Содержание**

Изменять содержимое окна можно с помощью кнопок на панели инструментов. Так, щелчок на кнопке со стрелкой, направленной вправо, приводит к отображению информации о следующем (по отношению к текущей «позиции») животном в последовательности *лиса*, *волк*, *медведь* и *енот*. Щелчок на кнопке со стрелкой, направленной влево, приводит к отображению информации о «предыдущем» животном в указанной последовательности.

НА ЗАМЕТКУ

Используется принцип циклических переходов. Так, если, например, отображается информация о волке, то после последовательности щелчков на кнопке со стрелкой вправо будет отображаться информация о медведе, затем о еноте, о лисе, о волке, и так далее. Если щелкать кнопку со стрелкой влево, то отображается информация о лисе, еноте, медведе, волке и так далее.

На рис. 16.4 показана ситуация, когда в окне отображается информация о медведе, а курсор мыши наведен на кнопку на панели инструментов с изображением стрелки, направленной вправо (будет показано «следующее» изображение).

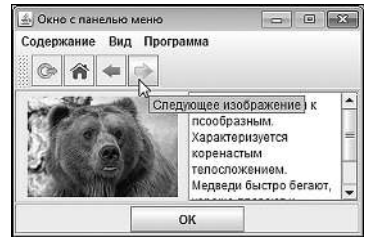


Рис. 16.4. Щелчок на кнопке панели инструментов для отображения следующего изображения и сопутствующего текста

При этом для кнопки отображается подсказка. После щелчка на кнопке в окне отображается информация о еноте, как показано на рис. 16.5.

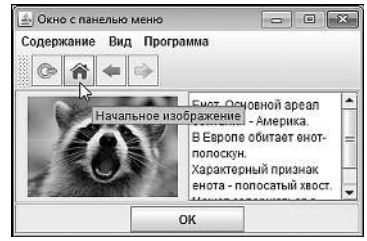


Рис. 16.5. Результат щелчка на кнопке отображения следующего изображения и щелчок на кнопке для отображения начального изображения

На том же рис. 16.5 курсор мыши наведен на кнопку с изображением «домика». Для этой кнопки, как и для всех прочих кнопок панели инструментов, отображается подсказка. Щелчок на кнопке приводит к тому, что отображается «начальная» информация — то есть информация о лисе. Результат щелчка на данной кнопке показан на рис. 16.6.

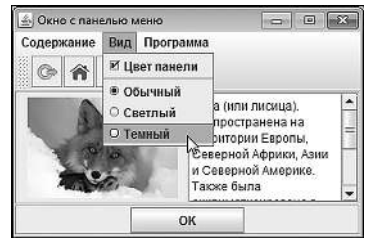


Рис. 16.6. Результат перехода к начальному изображению и выбор команды-переключателя **Темный** в пункте меню **Вид**

Также на рис. 16.6 раскрыт пункт меню **Вид**. Среди команд меню есть опция **Цвет панели** (по умолчанию флажок установлен), и три переключателя (**Обычный**, **Светлый** и **Темный**). Установив тот или иной переключатель, задаем цвет для области панели с изображением и текстом. На рис. 16.7 показано, как будет выглядеть окно, если установить переключатель **Темный**.

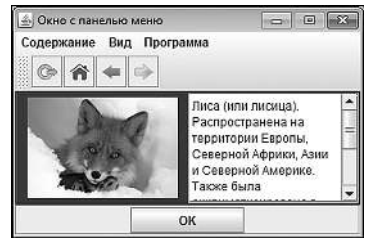


Рис. 16.7. Результат применения темного фона для панели с изображением и текстом

НА ЗАМЕТКУ

Если установлен переключатель **Обычный**, то цвет панели совпадает с цветом фона окна. Если установлен переключатель **Светлый**, то фон панели будет белым.

Следует отметить, что переключатели в пункте меню **Вид** доступны, только если установлен флажок опции **Цвет панели**. Если отменить флажок опции, переключатели станут недоступны, как показано на рис. 16.8.

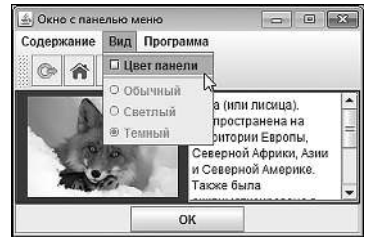


Рис. 16.8. При отмене флажка у команды-опции **Цвет панели** в пункте меню **Вид** приводит к тому, что команды-переключатели становятся недоступны

Вернув флажок на свое место, снова получим возможность изменять цвет фона панели.

Содержимое пункта меню **Программа** показано на рис. 16.9.

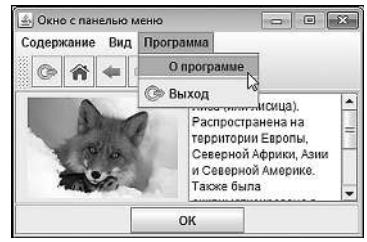


Рис. 16.9. Выбор команды **О программе** в пункте меню **Программа**

Там всего две команды: **О программе** и **Выход**, причем для последней отображается не только название, но и пиктограмма (такая же, как и для первой кнопки на панели инструментов). При щелчке на команде **Выход** завершается выполнение программы. При щелчке на команде **О программе** отображается диалоговое окно, представленное на рис. 16.10.

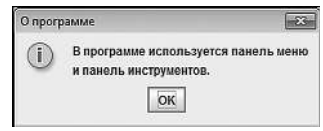


Рис. 16.10. Данное диалоговое окно отображается в результате щелчка на команде **О программе** в пункте меню **Программа**

Окно содержит информацию о проекте, и оно *модальное* — пока не закрыто данное окно, исходное окно программы будет недоступно (заблокировано).

Наконец, для метки предусмотрено контекстное меню. Если навести курсор мыши на область метки и щелкнуть правой кнопкой мыши, откроется список с командами (контекстное меню), как показано на рис. 16.11.

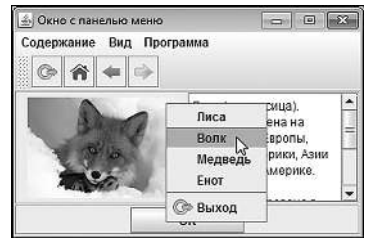


Рис. 16.11. Выбор команды **Волк** в контекстном меню для метки с изображением

В контекстном меню есть команды с названиями животных и команда **Выход** (с пиктограммой). Щелчок на команде **Выход** приводит к завершению программы, а щелчок на команде с названием животного приведет к тому, что будет отображена информация о нем. На рис. 16.12 показано, к какому результату приведет щелчок на команде **Волк** в контекстном меню.

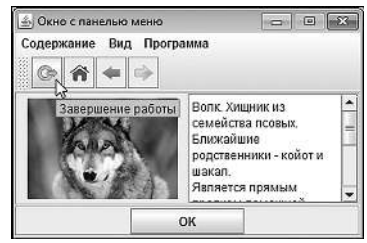


Рис. 16.12. Результат выбора команды **Волк** в контекстном меню. Курсор мыши наведен на кнопку, предназначенную для завершения выполнения программы

Еще одно замечание касается панели инструментов. Дело в том, что мы не закрепляем панель инструментов, что позволяет перемещать ее с помощью мыши (путем захвата и перетаскивания). На рис. 16.13 проиллюстрирован процесс «перетаскивания» панели в левую часть окна.

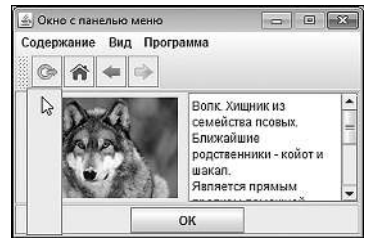


Рис. 16.13. Перемещение незакрепленной панели инструментов

На рис. 16.14 показано окно, в котором панель инструментов отображается слева в окне приложения.

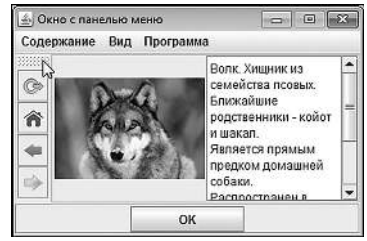


Рис. 16.14. Панель инструментов отображается в левой части окна

Панель можно вовсе «оторвать» от границ окна приложения, и тогда панель будет отображаться в отдельном окне, как это показано на рис. 16.15.

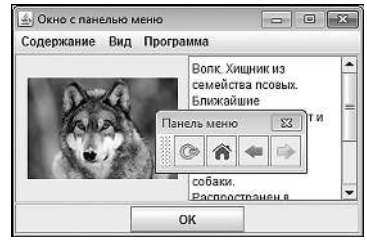


Рис. 16.15. Панель инструментов отображается в отдельном окне

Щелчок на системной пиктограмме приводит к тому, что панель возвращается в свое исходное положение у внутренней границы окна.

Программный код примера

Далее рассмотрим программный код, благодаря которому реализованы все вышеперечисленные возможности в плане манипуляций с окном, отображаемым при выполнении программы. Код представлен в листинге 16.1.

Листинг 16.1. Программный код проекта UsingMenuApplication

```
// Импорт классов:
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
// Класс окна реализует интерфейс ActionListener:
class MyFrame extends JFrame implements ActionListener{
  // Поле, определяющее изображение и текст,
  // которые следует отображать в окне:
  private int state;
  // Ссылка на панель, содержащую изображение
  // и текстовую панель:
  private JPanel pnl;
  // Ссылка на объект, определяющий рамку для панелей:
  private Border bdr;
  // Массив со значениями цвета для фона панели:
  private Color[] clr=new Color[3];
  // Кнопка для закрытия окна:
  private JButton btn;
  // Ссылки на пункты меню:
  private JMenu content,view,program;
  // Массив с изображениями:
  private ImageIcon[] imgs;
  // Массивы с названиями животных:
  private String[] engNames={"fox","wolf","bear","raccoon"};
```

```
private String[] cyrNames={"Лиса","Волк","Медведь","Енот"};
// Массив для записи названий файлов с текстом:
private String[] files;
// Ссылка на метку с изображением:
private JLabel lbl;
// Ссылка на текстовую панель:
private JTextPane tp;
// Ссылка на панель меню:
private JMenuBar mb;
// Ссылки на команды меню:
private JMenuItem about,exit;
// Массив со ссылками на команды меню:
private JMenuItem[] animals;
// Ссылка на опционную команду меню:
private JCheckBoxMenuItem color;
// Ссылки на команды меню, являющиеся переключателями:
private JRadioButtonMenuItem light,dark,ordinary;
// Ссылка на панель инструментов:
private JToolBar tb;
// Ссылка на кнопки, размещаемые на панели инструментов:
private MyButton exitBtn,nextBtn,prevBtn,startBtn;
// Ссылка на контекстное меню:
private JPopupMenu pm;
// Внутренний класс для кнопок, размещаемых
// на панели инструментов:
class MyButton extends JButton{
  // Конструктор:
  MyButton(String txt){
    // Вызов конструктора суперкласса:
    super(new ImageIcon("d:/books/pictures/"+txt));
    // Отмена рисования рамки фокуса:
    setFocusPainted(false);
  }
```

```
  }
  // Метод для установки изображения и текста:
  private void setContent(){
    // Применение изображения к метке:
    lbl.setIcon(imgs[state]);
    // Контролируемый код:
    try{
      // Для текстовой панели применяется
      // текст из файла:
      tp.setPage(files[state]);
    }
    // Обработка исключения:
    catch(IOException err){
      tp.setText("Информация недоступна");
    }
  }
  // Реализация метода из интерфейса ActionListener:
  public void actionPerformed(ActionEvent e){
    // Присваивание полю state значением целого числа,
    // которое получается преобразованием из
    // текстового формата команды действия для
    // компонента, на котором произошло событие:
    state=Integer.parseInt(((JMenuItem)e.getSource()).getActionCommand());
    // Вызов метода, устанавливающего изображение в метке
    // и текст в текстовой панели:
    setContent();
  }
  // Конструктор класса:
  MyFrame(){
    // Вызов конструктора суперкласса:
    super("Окно с панелью меню");
    // Положение и размеры окна:
    setBounds(250,250,350,230);
```

```
// Окно постоянных размеров:
setResizable(false);
// Реакция на щелчок системной пиктограммы:
setDefaultCloseOperation(EXIT_ON_CLOSE);
// Применение менеджера компоновки:
setLayout(new BorderLayout());
// Начальное значение целочисленного поля:
state=0;
// Тип рамки для панелей:
bdr=BorderFactory.createEtchedBorder();
// Используемые цвета:
clr[0]=getBackground();
clr[1]=Color.WHITE;
clr[2]=Color.DARK_GRAY;
// Массив с изображениями:
imgs=new ImageIcon[engNames.length];
// Массив с названиями текстовых файлов:
files=new String[engNames.length];
// Массив со ссылками на команды пункта меню:
animals=new JMenuItem[cyrNames.length];
// Заполнение массива с изображениями
// и массива с названиями файлов:
for(int k=0;k<engNames.length;k++){
  imgs[k]=new ImageIcon("d:/books/pictures/"+engNames[k]+".jpg");
  files[k]="file:///d:/books/files/"+engNames[k]+".txt";
}
// Создание объекта для панели инструментов:
tb=new JToolBar("Панель меню");
// Создание объектов для кнопок, размещаемых
// на панели инструментов и определение для них
// контекстных подсказок:
exitBtn=new MyButton("exit.png");
exitBtn.setToolTipText("Завершение работы");
```

```
startBtn=new MyButton("start.png");
startBtn.setToolTipText("Начальное изображение");
prevBtn=new MyButton("prev.png");
prevBtn.setToolTipText("Предыдущее изображение");
nextBtn=new MyButton("next.png");
nextBtn.setToolTipText("Следующее изображение");
// Добавление кнопок на панель инструментов:
tb.add(exitBtn);
tb.add(startBtn);
tb.add(prevBtn);
tb.add(nextBtn);
// Добавление панели инструментов в окно:
add(tb,BorderLayout.NORTH);
// Создание панели:
pnl=new JPanel();
// Рамка вокруг панели;
pnl.setBorder(bdr);
// Определение менеджера компоновки для панели:
pnl.setLayout(new GridLayout(1,2));
// Создание метки:
lbl=new JLabel();
// Режим выравнивания содержимого по центру:
lbl.setHorizontalAlignment(JLabel.CENTER);
// Добавление метки на панель:
pnl.add(lbl);
// Добавление панели в окно:
add(pnl,BorderLayout.CENTER);
// Область с полосами прокрутки:
JScrollPane sp=new JScrollPane();
// Создание текстовой панели:
tp=new JTextPane();
// Режим запрещает редактирование содержимого панели:
tp.setEditable(false);
```

```
// Добавление текстовой панели в область
// просмотра панели с полосами прокрутки:
sp.getViewport().add(tp);
// Добавление панели с полосами прокрутки
// на обычную панель:
pnl.add(sp);
// Создание кнопки:
btn=new JButton("OK");
// Отмена режима отображения рамки фокуса:
btn.setFocusPainted(false);
// Создание обычной панели:
JPanel p=new JPanel();
// Определение менеджера компоновки для панели:
p.setLayout(new GridLayout(1,3));
// Рамка вокруг панели:
p.setBorder(bdr);
// Добавление на панель "технической" панели:
p.add(new JPanel());
// Добавление на панель кнопки:
p.add(btn);
// Добавление на панель "технической" панели:
p.add(new JPanel());
// Добавление панели в окно:
add(p,BorderLayout.SOUTH);
// Создание панели меню:
mb=new JMenuBar();
// Создание пунктов меню:
content=new JMenu("Содержание");
view=new JMenu("Вид");
program=new JMenu("Программа");
// Создание команд меню:
about=new JMenuItem("О программе");
exit=new JMenuItem("Выход",exitBtn.getIcon());
```

```
// Добавление в пункт меню команд:
program.add(about);
program.addSeparator(); // Добавление разделителя
program.add(exit);
// Создание опционной команды меню:
color=new JCheckBoxMenuItem("Цвет панели",true);
// Создание команд меню, являющихся переключателями:
light=new JRadioButtonMenuItem("Светлый",false);
dark=new JRadioButtonMenuItem("Темный",false);
ordinary=new JRadioButtonMenuItem("Обычный",true);
// Создание объекта для группы переключателей:
ButtonGroup bg=new ButtonGroup();
// Добавление команд-переключателей в группу:
bg.add(ordinary);
bg.add(light);
bg.add(dark);
// Добавление в пункт меню опционной команды:
view.add(color);
// Добавление разделителя:
view.addSeparator();
// Добавление в пункт меню команд-переключателей:
view.add(ordinary);
view.add(light);
view.add(dark);
// Создание команд меню, реализованных через массив:
for(int k=0;k<animals.length;k++){
  // Создание объекта для команды меню:
  animals[k]=new JMenuItem(cyrNames[k]);
  // Определение текста для команды действия:
  animals[k].setActionCommand(""+k);
  // Регистрация обработчика:
  animals[k].addActionListener(this);
  // Добавление команды в пункт меню:
```

```
      content.add(animals[k]);
    }
    // Добавление пунктов меню на панель меню:
    mb.add(content);
    mb.add(view);
    mb.add(program);
    // Добавление в окно панели меню:
    setJMenuBar(mb);
    // Создание объекта для контекстного меню:
    pm=new JPopupMenu();
    // Заполнение контекстного меню командами:
    for(int k=0;k<cyrNames.length;k++){
      // Добавление команды в контекстное меню
      // с одновременным определением текста для
      // команды действия:
      pm.add(new JMenuItem(cyrNames[k])).setActionCommand(""+k);
      // Регистрация обработчика:
      ((JMenuItem)pm.getComponent(k)).addActionListener(this);
    }
    // Добавление разделителя:
    pm.addSeparator();
    // Добавление команды в контекстное меню
    // с одновременной регистрацией обработчика:
    pm.add(new JMenuItem("Выход",exitBtn.getIcon())).addActionListener(e->exitBtn.
doClick());
    // Регистрация контекстного меню для метки:
    lbl.setComponentPopupMenu(pm);
    btn.addActionListener(e->System.exit(0));
    // Регистрация обработчика для кнопок на панели
    // инструментов:
    exitBtn.addActionListener(btn.getActionListeners()[0]);
    nextBtn.addActionListener(e->{
      state=(state+1)%(engNames.length);
```

```
    setContent();
  });
  prevBtn.addActionListener(e->{
    state=state==0?engNames.length-1:(state-1);
    setContent();
  });
  startBtn.addActionListener(e->{
    state=0;
    setContent();
  });
  // Регистрация обработчиков для команд меню:
  exit.addActionListener(exitBtn.getActionListeners()[0]);
  about.addActionListener(e->{
    // Отображение диалогового окна:
    JOptionPane.showMessageDialog(
      this,  // Родительское окно
      // Текст в области окна (текст сообщения):
      "В программе используется панель меню\nи панель инструментов.",
      // Название окна:
      "О программе",
      // Тип пиктограммы:
      JOptionPane.INFORMATION_MESSAGE
    );
  });
  // Регистрация обработчика для метки. При щелчке
  // на правой кнопке мыши в области метки
  // отображается контекстное меню:
  lbl.addMouseListener(new MouseAdapter(){
    public void mousePressed(MouseEvent e){
      // Проверяем, какая нажата кнопка мыши:
      if(e.isPopupTrigger()){
        // Отображение контекстного меню в месте
        // размещения курсора:
```

```
            pm.show(e.getComponent(),e.getX(),e.getY());
          }
        }
      });
      // Регистрация обработчика для опционной команды:
      color.addActionListener(e->{
        // Если опция установлена:
        if(color.isSelected()){
          ordinary.setEnabled(true);
          light.setEnabled(true);
          dark.setEnabled(true);
        }
        // Если опция не установлена:
        else{
          ordinary.setEnabled(false);
          light.setEnabled(false);
          dark.setEnabled(false);
        }
      });
      // Регистрация обработчиков
      // для команд-переключателей:
      ordinary.addActionListener(e->pnl.setBackground(clr[0]));
      light.addActionListener(e->pnl.setBackground(clr[1]));
      dark.addActionListener(e->pnl.setBackground(clr[2]));
      // Заполнение центральной панели:
      setContent();
      // Отображение окна:
      setVisible(true);
    }
}
// Главный класс:
class UsingMenuDemo{
  // Главный метод:
```

```
  public static void main(String[] args){
    // Создание объекта окна:
    new MyFrame();
  }
}
```

В принципе, код содержит в основном уже знакомые нам по предыдущим программам конструкции, а существенно новые утилиты в общих чертах описывались в начале главы. Вместе с тем, имеет смысл уделить внимание анализу программного кода.

Анализ программного кода

Программа состоит из import-инструкций, описания класса окна MyFrame и главного класса, в котором в методе main() создается анонимный объект окна, что приводит к отображению последнего. Также сразу отметим, что в классе окна MyFrame описывается внутренний класс MyButton, предназначенный для создания кнопок, размещаемых на панели инструментов.

Внутренний класс MyButton наследует класс JButton. Там описан лишь конструктор с текстовым аргументом (обозначен как txt). Предполагается, что этот аргумент — название файла с изображением, которое должно быть размещено на кнопке. При вызове конструктора суперкласса командой super(new ImageIcon("d:/books/pictures/"+txt)) фактически аргументом конструктору класса JButton передается выражение new ImageIcon("d:/books/pictures/"+txt). Это анонимный объект изображения, размещенного в соответствующем каталоге на диске. В результате указанное изображение будет на кнопке. Также командой setFocusPainted(false) к кнопке применяется режим, при котором не отображается рамка фокуса.

> **НА ЗАМЕТКУ**
>
> Размер и положение кнопок не задаем. Размер автоматически определяется размерами изображения, а положение автоматически определяется менеджером компоновки панели инструментов. В программе использованы изображения для кнопок размером 20 пикселов в ширину и 20 пикселов в высоту.

Важную роль в классе MyFrame играет закрытое целочисленное поле state. Значение этого поля определяет, для какого животного отображается информация в области окна. Значение 0 для поля означает, что отображается

информация о *лисе*, значение 1 означает, что отображается информация о *волке*, значение 2 подразумевает, что отображается информация о *медведе*, а при значении поля 3 отображается информация о *еноте*. Значение поля state напрямую используется в методе setContent(), которым (на основе значения поля state) определяется изображение и текст, отображаемые в окне.

В классе MyFrame объявляется несколько закрытых полей, большинство из которых предназначено для реализации графических компонентов. В частности, для доступа к компонентам интерфейса мы используем следующие поля:

- поле pnl, являющееся ссылкой класса JPanel на объект панели, в которую помещается метка с изображением и текстовая панель;
- поле btn, являющееся ссылкой класса JButton на кнопку в области окна, предназначенной для закрытия окна и завершения выполнения программы;
- поле lbl, являющееся ссылкой класса JLabel на метку с изображением;
- поле tp, представляющее собой ссылку класса JTextPane на текстовую панель;
- поле mb является ссылкой класса JMenuBar на объект панели меню;
- поле tb, которое является ссылкой класса JToolBar на объект панели инструментов;
- поле pm является ссылкой класса JPopupMenu на объект контекстной панели;
- через поля content, view и program (объектные ссылки класса JMenu) реализуются пункты главного меню (**Содержание**, **Вид** и **Программа** соответственно);
- поля about и exit являются объектными ссылками класса JMenuItem и предназначены для реализации команд **О программе** и **Выход** из пункта меню **Программа**;
- команды из пункта меню **Содержание** реализуются через массив animals объектных ссылок класса JMenuItem;
- опционная команда **Цвет панели** из пункта меню **Вид** реализуется в виде поля color, являющегося ссылкой класса JCheckBoxMenuItem;
- команды-переключатели **Обычный**, **Светлый** и **Темный** из пункта меню **Вид** реализуются с помощью полей light, dark и ordinary, представляющих собой ссылки класса JRadioButtonMenuItem;
- поля exitBtn, nextBtn, prevBtn и startBtn являются ссылками внутреннего класса MyButton на кнопки, размещаемые на панели инструментов.

НА ЗАМЕТКУ

Практически для всех (за несколькими исключениями) графических компонентов, используемых в программе, предусмотрено поле со ссылкой на соответствующий объект. На самом деле острой необходимости в этом нет. Можно было ограничиться лишь теми компонентами, на которые выполняются прямые ссылки — например, при обработке событий. То есть для части компонентов можно было ограничиться использованием локальных объектных переменных в теле конструктора. Подход, который мы используем, призван явно показать, из каких компонентов «конструируется» окно программы.

Помимо перечисленных полей, ряд закрытых полей имеет «вспомогательный» характер. Вот они.

- В массив imgs объектных ссылок класса ImageIcon записываются ссылки на объекты изображений животных, отображаемых в окне программы.
- Текстовые массивы engNames и cyrNames содержат соответственно английские и русские названия для животных. Русские названия используем при формировании команд в главном и контекстном меню, а английские названия нужны для полуавтоматического формирования ссылок на файлы с изображениями животных и на файлы, содержащие описание животных.
- Текстовый массив предназначен для записи полных путей к текстовым файлам на диске, содержащим информацию, предназначенную для отображения в окне программы.
- Массив clr из трех элементов, являющихся ссылками класса Color, используется при определении цвета фона центральной панели (с изображением и текстом) в окне программы.
- В поле bdr записывается ссылка на объект класса Border, определяющий тип рамки, применяемой к панелям в области окна.

НА ЗАМЕТКУ

Для явного использования класса Border в заголовке программы подключается пакет javax.swing.border.

Все основные операции выполняются в конструкторе класса MyFrame. После вызова конструктора суперкласса, в результате чего, кроме прочего,

определяется название окна (определяется аргументом конструктора суперкласса), командами setBounds(250,250,350,230) и setResizable(false) задаются положение и размеры окна, а также выполняется переход в режим постоянных размеров. Командой setDefaultCloseOperation(EXIT_ON_CLOSE) определяем реакцию на щелчок системной пиктограммы (завершение выполнения программы). Далее командой setLayout(new BorderLayout()) для окна задается менеджер компоновки: аргументом методу satLayout() передается анонимный объект класса BorderLayout.

Полю state в конструкторе присваивается начальное нулевое значение (поэтому вначале будет отображаться информация для *лисы*). Полю bdr значением присваивается выражение BorderFactory.createEtchedBorder(). Результатом выражения возвращается ссылка на объект, определяющий «вдавленную» рамку. Такого типа рамку мы будем использовать для некоторых панелей в программе.

Массив clr заполняется в «явном» виде с использованием команд clr[0]=getBackground(), clr[1]=Color.WHITE и clr[2]=Color.DARK_GRAY. Здесь мы использовали метод getBackground(), который вызывается из объекта окна. Результатом метод возвращает ссылку на объект класса Color, определяющий цвет для фона объекта, из которого вызывается метод. Константа WHITE означает белый цвет, а константа DARK_GRAY соответствует темно-серому цвету.

Массив с изображениями животных создается командой imgs=new ImageIcon[engNames.length]. Размер массива определяется количеством элементов в текстовом массиве engNames с английскими названиями животных. Аналогично командой files=new String[engNames.length] создается массив для записи полных путей к текстовым файлам. Далее, командой animals=new JMenuItem[cyrNames.length] создаем массив с названиями для команд меню (из пункта меню **Содержание**). Массив изображений и массив с названиями файлов заполняются в рамках оператора цикла, в котором индексная переменная k пробегает значения индексом элементов массива engNames.length. Значение элементам массива imgs присваивается командой imgs[k]=new ImageIcon("d:/books/pictures/"+engNames[k]+".jpg"), а для массива files элементы получают значения в результате выполнения команды files[k]="file:///d:/books/files/"+engNames[k]+".txt". В обоих случаях используется элемент engNames[k] с английским названием животного, и на основе этого названия «конструируется» путь к соответствующему файлу. Для текстовых файлов полный путь начинается с инструкции file:///, поскольку таковы правила указания пути к файлам на диске при работе с текстовой панелью.

Командой tb=new JToolBar("Панель меню") создается объект для панели инструментов. Текстовое значение, переданное аргументом конструктору класса JToolBar, определяет название для панели инструментов. Это название содержится в строке названия панели, когда панель отображается в отдельном окне (см. рис. 16.15).

Далее следует серия команд, которыми создаются объекты класса MyButton — это объекты кнопок, добавляемых на панель инструментов. Для каждой из этих кнопок аргументом указывается имя файла с соответствующим изображением, помещаемым на кнопку, а с помощью метода setToolTipText() для каждой из кнопок задается интерактивная подсказка (отображается при наведении курсора мыши на кнопку). Для добавления кнопки на панель инструментов используется метод add(). Например, командой exitBtn=new MyButton("exit.png") создается объект для кнопки, изображение которой находится в файле с названием exit.png в каталоге d:\books\pictures (см. код конструктора внутреннего класса MyButton). Командой exitBtn.setToolTipText("Завершение работы") текст "Завершение работы" задается в качестве подсказки, а командой tb.add(exitBtn) кнопка добавляется на панель инструментов. Кнопки startBtn, prevBtn и nextBtn обрабатываются по той же схеме. Размещаются кнопки на панели инструментов слева направо в том порядке, как они туда добавляются.

Сама панель инструментов добавляется в окно с помощью команды add(tb,BorderLayout.NORTH). Второй аргумент BorderLayout.NORTH означает, что панель размещается вдоль верхней внутренней границы окна.

Командой pnl=new JPanel() создается окно для панели, которая будет занимать центральную часть окна и содержать метку с изображением и текстовую панель. С помощью инструкции pnl.setBorder(bdr) задаем рамку для панели. Менеджер компоновки для панели определяем командой pnl.setLayout(new GridLayout(1,2)). Аргументом методу setLayout() передан анонимный объект new GridLayout(1,2) менеджера компоновки, предназначенного для размещения двух компонентов (метка и текстовая панель) в одну строку. Метку создаем командой lbl=new JLabel() и задаем способ выравнивания ее содержимого по центру (инструкция lbl.setHorizontalAlignment(JLabel.CENTER)). Добавляется метка на панель командой pnl.add(lbl). После этого панель с помощью команды add(pnl,BorderLayout.CENTER) добавляется в центральную часть окна программы. Но на панели остается еще одна «вакансия» — для текстовой панели. Но если точнее, то мы создаем панель с полосами прокрутки (команда JScrollPane sp=new JScrollPane()), а затем «прячем» в эту панель собственно текстовую панель. Текстовая панель

создается командой tp=new JTextPane(). Командой tp.setEditable(false) мы сразу переводим панель в режим, при котором нельзя редактировать ее содержимое. «Упаковывание» текстовой панели в панель с полосами прокрутки выполняется командой sp.getViewport().add(tp). То есть мы не просто добавляем одну панель в другую, а в область просмотра панели с полосами прокрутки. Для получения доступа к области просмотра вызывается метод getViewport(). Наконец, командой pnl.add(sp) панель с полосами прокрутки и текстовой панелью добавляется в центральную панель окна.

Кнопку для окна (она там единственная, если не считать кнопок на панели инструментов) создаем командой btn=new JButton("OK"). Командой btn.setFocusPainted(false) для кнопки отменяется режим отображения рамки фокуса. После этого создаем панель командой JPanel p=new JPanel(), командой p.setLayout(new GridLayout(1,3)) задаем для этой панели менеджер компоновки, который размещает три компонента в одну строку. Для созданной панели командой p.setBorder(bdr) задаем рамку, после чего командами p.add(new JPanel()), p.add(btn) и p.add(new JPanel()) добавляем в эту панель три компонента: анонимную панель, кнопку, и еще одну анонимную панель. В итоге кнопка будет размещена посредине и занимает по ширине треть области панели p. Командой add(p,BorderLayout.SOUTH) панель с кнопкой добавляется в нижнюю часть окна программы.

НА ЗАМЕТКУ

В итоге получилось так: в верхней части окна размещена панель инструментов, в центральной части окна размещена панель с изображением и текстовой панелью, а в нижней части окна размещена панель, которая содержит панель, кнопку и еще одну панель. Последние две «технические» панели нам понадобились просто для того, чтобы заполнить нижнюю панель. Там нужно всего три компонента, но у нас был только один — кнопка, поэтому в качестве «недостающих» двух мы использовали панели, которые реализовали с помощью анонимных объектов.

Объект для панели меню создается командой mb=new JMenuBar(). Пункты меню создаем командами content=new JMenu("Содержание"), view=new JMenu("Вид") и program=new JMenu("Программа"). Команды меню для пункта меню **Программа** создаются командами about=new JMenuItem("О программе") и exit=new JMenuItem ("Выход",exitBtn.getIcon()), причем в последнем случае указано не только название для команды, но и пиктограмма. Пиктограмма такая же, как на

кнопке exitBtn на панели инструментов. Для получения ссылки на объект пиктограммы кнопки используем метод getIcon(). Команды добавляются в пункт меню с помощью инструкций program.add(about) и program.add(exit). Между ними посредством инструкции program.addSeparator() добавляется разделитель — декоративная горизонтальная линия, которую можно видеть, например, на рис. 16.9.

Опционная команда меню для пункта меню **Вид** создается инструкцией color=newJCheckBoxMenuItem("Цветпанели",true). Второй аргумент true означает, что флажок опции установлен. Для создания команд-переключателей используем инструкции light=new JRadioButtonMenuItem("Светлый",false), dark=new JRadioButtonMenuItem("Темный",false) и ordinary=new JRadioButtonMenuItem("Обычный",true). Второй логический аргумент определяет состояние переключателя (установлен или нет). Переключатели необходимо объединить в группу. Для этого командой ButtonGroup bg=new ButtonGroup() создаем объект группы и командами bg.add(ordinary), bg.add(light) и bg.add(dark) последовательно добавляем в него переключатели. Добавление команд в пункт меню **Вид** выполняется с помощью инструкций view.add(color), view.add(ordinary), view.add(light) и view.add(dark). Здесь мы также используем разделитель (инструкция view.addSeparator(), а сам разделитель можно видеть на рис. 16.6 и рис. 16.8).

Для пункта меню **Содержание** создание объектов для команд меню выполняется в полуавтоматическом режиме с использованием оператора цикла, в котором индексная переменная k пробегает весь набор значений индексов элементов массива animals. За каждый цикл командой animals[k]=new JMenuItem(cyrNames[k]) создается объект команды меню, и ссылка на него присваивается значением элементу массива animals. Затем командой animals[k].setActionCommand(""+k) для объекта команды меню задается *команда действия* — текстовая характеристика компонента, которая в данном случае является текстовым представлением для индекса соответствующего элемента в массиве animals. Мы команду действия будем использовать в обработчике события для «идентификации» команд меню. То есть выполнение данной операции продиктовано соображениями, относящимися к обработке событий.

НА ЗАМЕТКУ

Острой необходимости в использовании команды действия при обработке событий, в общем-то, нет. Просто мы выбрали такой способ реализации обработки событий. Но, в принципе, можно было использовать и иные подходы.

Командой animals[k].addActionListener(this) обработчиком событий класса ActionEvent для команды меню регистрируется объект окна (в классе MyFrame реализуется интерфейс ActionListener). Наконец, командой content.add(animals[k]) команда меню добавляется в пункт меню **Содержание**.

Командами mb.add(content), mb.add(view) и mb.add(program) пункты меню **Содержание**, **Вид** и **Программа** добавляются на панель меню. Панель меню добавляем в окно посредством инструкции setJMenuBar(mb).

Формирование контекстного меню начинается с создания объекта для контекстного меню. С этой целью использована команда pm=new JPopupMenu(). «Заполнение» контекстного меню командами начинается в операторе цикла. Там индексная переменная k пробегает значения индексов элементов массива cyrNames, и за каждый цикл выполняется две команды. Сначала командой pm.add(new JMenuItem(cyrNames[k])).setActionCommand(""+k) добавляется команда в контекстное меню и одновременно для этой команды меню задается текст команды действия (текстовое представление индексной переменной k). Проанализируем приведенную выше команду. В первую очередь следует учесть, что метод add() возвращает результат — это ссылка на объект, который добавляется в контекстное меню. А добавляется в контекстное меню объект, указанный аргументом метода add(). Аргументом метода указан анонимный объект, который создается инструкцией new JMenuItem(cyrNames[k]). То есть это объект для команды меню с названием, определяемым значением элемента cyrNames[k] текстового массива cyrNames. Этот объект добавляется в контекстное меню, а ссылка на него возвращается методом add(). Через эту ссылку вызывается метод setActionCommand() с аргументом, определяющим текст команды действия для добавленного в контекстное меню объекта.

Командой ((JMenuItem)pm.getComponent(k)).addActionListener(this) для добавленного в контекстное меню объекта команды меню регистрируется обработчик события класса ActionEvent. Таким обработчиком является объект окна. В данной команде мы из объекта контекстного меню pm вызываем метод getComponent() с аргументом k. Результатом метод возвращает ссылку на объект в контекстном меню с соответствующим индексом. Несложно понять, что это ссылка на только что добавленный в контекстное меню объект. Но поскольку ссылка возвращается класса Component, то мы выполняем явное приведение к типу JMenuItem. Через полученную ссылку на объект команды контекстного меню вызываем метод addActionListener(), которым регистрируется обработчик для объекта команды меню.

НА ЗАМЕТКУ

В этом случае при заполнении контекстного меню мы используем тот же подход, что и при заполнении пункта меню **Содержание**: добавляется объект команды меню, для него определяется текст команды действия, который совпадает с индексом соответствующего текстового элемента в массиве cyrNames, а обработчиком для объекта команды регистрируется объект окна. Разница в том, что команды для пункта меню Содержание реализуются через поля класса MyFrame, а при заполнении контекстного меню для команд меню используются анонимные объекты.

После добавления в контекстное меню команд с названиями животных, инструкцией pm.addSeparator() добавляем в контекстное меню разделитель, после чего командой pm.add(new JMenuItem("Выход",exitBtn.getIcon())).addActionListener(e->exitBtn.doClick()) в контекстное меню добавляется последняя команда.

Одновременной для этой команды выполняется регистрация обработчика события класса ActionEvent. Здесь мы также учитываем, что метод add() результатом возвращает ссылку на добавляемый в контекстное меню объект. В данном случае речь идет об анонимном объекте new JMenuItem("Выход",exitBtn.getIcon()). Это команда меню, название которой определяется первым аргументом "Выход" конструктора класса JMenuItem, а второй аргумент exitBtn.getIcon() является ссылкой на пиктограмму, отображаемую для кнопки exitBtn на панели инструментов. Для этого объекта (через ссылку, возвращаемую методом add()) вызывается метод addActionListener(). Аргументом метода указано лямбда-выражение e->exitBtn.doClick(). Данное лямбда-выражение определяет программный код, который будет выполняться при щелчке на соответствующей команде контекстного меню. Инструкция exitBtn.doClick() определяет действия, выполняемые в таком случае. Вызов метод doClick() из объекта кнопки exitBtn означает, что программными методами эмулируется щелчок на кнопке. Другими словами, обработка события для данной команды контекстного меню сводится к тому, что программными методами выполняется «щелчок» на кнопке на панели инструментов, предназначенной для завершения выполнения программы.

После того, как контекстное меню создано, мы его регистрируем в том компоненте, котором оно предназначено. Это метка. Поэтому используем команду lbl.setComponentPopupMenu(pm).

НА ЗАМЕТКУ

Зарегистрировать контекстное меню для компонента — это еще не все. Нужно реализовать еще и соответствующую обработку событий. Она выполняется далее.

Затем в конструкторе класса MyFrame следуют команды, связанные с регистрацией обработчиков событий. Так, командой btn.addActionListener(e->System.exit(0)) на основе лямбда-выражения регистрируется обработчик для кнопки **ОК**, размещенной в области окна. Регистрация обработчика для аналогичной кнопки на панели инструментов выполняется командой exitBtn.addActionListener(btn.getActionListeners()[0]). Здесь результатом выражения btn.getActionListeners() является массив обработчиков событий класса ActionEvent, зарегистрированных в кнопке btn. Поскольку такой обработчик всего один, то массив состоит из одного элемента и, значит, выражением btn.getActionListeners()[0] возвращается ссылка на обработчик, зарегистрированный в кнопке btn. Этот же обработчик регистрируется в кнопке exitBtn на панели инструментов.

Аналогичным образом регистрируется обработчик для команды **Выход** из пункта меню **Программа** (команда exit.addActionListener(exitBtn.getActionListeners()[0])). Только теперь формально регистрируется обработчик из кнопки на панели инструментов (который является обработчиком и для кнопки в области окна).

Для кнопки nextBtn обработчик регистрируется с передачей аргументом лямбда-выражения, содержащего в своем теле команды state=(state+1)%(engNames.length) и setContent(). Первой из приведенных команд значением полю state от целочисленного деления выражения state+1 на длину массива engNames (определяет количество животных в списке). Фактически здесь просто увеличивается на единицу текущее значение поля state, а вычисление остатка от деления делает процесс «циклическим». Вызовом метода setContent(), с учетом нового значения поля state, в области окна отображается «правильная» картинка и текст.

Аналогично реализован обработчик для кнопки prevBtn, но на этот раз новое значение поля state вычисляется командой state=state==0?engNames.length-1:(state-1). Здесь использован тернарный оператор: если текущее значение поля state нулевое, то новое значение равно engNames.length-1 (индекс последнего элемента в массиве engNames). В противном случае значение поля state уменьшается на единицу.

Для кнопки startBtn обработка щелчка на кнопке состоит в том, что полю state присваивается нулевое значение, после чего вызывается метод setContent().

Что же делает метод setContent()? В теле метода setContent() командой lbl.setIcon(imgs[state]) к метке lbl с изображением применяется изображение из массива imgs с индексом state. Далее в блоке контролируемого кода командой tp.setPage(files[state]) для текстовой панели определяется текстовый файл, содержание которого следует отобразить. Файл определяется через полный путь к нему. Путь к файлу содержится в элементе с индексом state текстового массива files. Поскольку метод setPage() может вызвать контролируемое исключение класса IOException (класс доступен после импорта пакета java.io), то мы выполняем обработку исключений. При возникновении ошибки данного класса командой tp.setText("Информация недоступна") в текстовой панели отображается соответствующий текст о невозможности получить информацию.

При регистрации обработчика для команды **О программе** (поле about) в пункте меню **Программа** использовано лямбда-выражение с инструкцией вызова диалогового окна. Для отображения диалогового окна использован статический метод showMessageDialog() из класса JOptionPane. Специфика ситуации в том, что первым аргументом методу showMessageDialog() передана ссылка this на объект окна программы. Поэтому окно программы является родительским для данного диалогового окна. Пока диалоговое окно не будет закрыто, родительское окно остается недоступным.

Обработчик для метки lbl регистрируется с целью определения способа отображения контекстного меню. Речь идет об обработчике события класса MouseEvent. Обработчик создается на основе анонимного класса, наследующего класс-адаптер MouseAdapter. При создании объекта обработчика описывается только метод mousePressed() (нажатие кнопки мыши) с аргументом e (ссылка на объект события класса MouseEvent). В теле метода в условном операторе проверяется условие e.isPopupTrigger(). Результатом данного выражения является true если был выполнен щелчок на кнопке мыши, используемой в соответствии с общими настройками системы для отображения контекстной справки (или контекстного меню). Обычно это правая кнопка мыши. Если так, то командой pm.show(e.getComponent(),e.getX(),e.getY()) отображается контекстное меню. Первым аргументом метода show() указана ссылка на компонент, вызвавший событие, а также координаты курсора мыши при щелчке (определяется выражениями e.getX() и e.getY()) — в этом месте будет отображаться контекстное меню.

Для опционной команды **Цвет панели** (поле color) из пункта меню **Вид** регистрация обработчика выполняется на основе лямбда-выражения. В теле выражения использован условный оператор, в котором проверяется, установлен ли флажок опции (условие color.isSelected()). Если так, то вызовом метода setEnabled() с аргументом true через ссылки ordinary, light и dark на объекты команд-переключателей соответствующие команды переводятся в активное состояние (будут доступны для использования). В противном случае метод вызывается ас аргументом false, и команды-переключатели будут неактивны (будут недоступны для использования).

Для команд переключателей **Обычный**, **Светлый** и **Темный** (соответственно поля ordinary, light и dark) из пункта меню **Вид** обработчики регистрируются командами ordinary.addActionListener(e->pnl.setBackground(clr[0])), light.addActionListener(e->pnl.setBackground(clr[1])) и dark.addActionListener(e->pnl.setBackground(clr[2])). В данном случае отличие лишь в цвете, который применяется для панели pnl.

После выполнения всех настроек в конструкторе класса MyFrame командой setContent() в центральную панель вставляется картинка и текст, а командой setVisible(true) отображается окно.

Нам осталось лишь проанализировать программный код метода actionPerformed(), описанного в классе MyFrame и вызываемого при щелчке на командах из пункта меню **Содержание** или на командах (кроме последней) в контекстном меню.

Код достаточно простой. Сначала выполняется команда state=Integer.parseInt(((JMenuItem)e.getSource()).getActionCommand()). Здесь из ссылки e (аргумент метода actionPerformed()) на объект события класса ActionEvent вызывается метод getSource(), возвращающий результатом ссылку на объект компонента, вызвавшего событие. Эта ссылка приводится к типу JMenuItem (мы точно знаем, что вызвавшим событие компонентом является команда меню). Через полученную ссылку вызывается метод getActionCommand(). Метод результатом возвращает текст команды действия. Но при создании объектов команд мы задавали текстом команды действия текстовое представление для целочисленного индекса. Чтобы получить значение этого индекса, всю описанную инструкцию передаем аргументом статическому методу parseInt() класса-оболочки Integer. Результат записываем в поле state. После этого вызывается метод setContent(), в результате чего в окне отображается «правильная» картинка и «правильный» текст.

Резюме

— Лектор готов?
— Готов лектор, давно готов.
— Выпускайте.

Из к/ф «Карнавальная ночь»

- При создании приложений с графическим интерфейсом могут использоваться, кроме прочего, такие компоненты, как главное меню, контекстное меню, панель инструментов.
- Для создания панели меню используют класс `JMenuBar`. Пункты меню создаются на основе класса `JMenu`. Команды для меню реализуют в виде объектов класса `JMenuItem` (обычные команды меню), `JCheckBoxMenuItem` (команды-опции) и `JRadioButtonMenuItem` (команды-переключатели).
- Контекстное меню реализуется с помощью объекта класса `JPopupMenu`. Команды для контекстного меню создаются так же, как и для главного меню.
- Панель инструментов реализуется с помощью объекта класса `JToolBar`. Кнопки, добавляемые на панель инструментов, реализуются стандартными методами — например, создаются с использованием класса `JButton`.

Глава 17
АППЛЕТЫ

Ну, что ж, заслушаем клоунов.

Из к/ф «Карнавальная ночь»

В этой главе мы познакомимся с *апплетами*. Апплеты являются программами особого типа. Это программы, которые выполняются под управлением браузера — программы, предназначенной для просмотра веб-страниц. В общих чертах схема использования апплетов выглядит примерно следующим образом. Откомпилированный код программы размещается на сервере, а доступ к этой программе пользователь получает с помощью браузера со своего клиентского компьютера. Технически это может выглядеть так, что посредством браузера открывается документ с гипертекстовой разметкой (HTML-разметка), и в этом документе содержится ссылка на апплет. В результате открытия документа в браузере апплет загружается на компьютер клиента и там выполняется (под управлением браузера). Чистый «выигрыш» от использования такой схемы состоит в том, что апплет уже откомпилирован, поэтому компилировать его уже не нужно. Теперь перейдем к более детальному рассмотрению способов создания и использования апплетов.

Знакомство с апплетами

У меня есть мысль, и я ее думаю.

Из м/ф «38 попугаев»

Прежде чем приступить к рассмотрению собственно программных кодов, которыми создаются апплеты, придется уделить внимание и некоторым «смежным» темам, не относящимся напрямую к языку программирования Java, но важным и просто критически необходимым в плане практической реализации апплета.

Общие принципы реализации апплета

Итак, мы в силу каких-то причин решили прибегнуть к помощи апплетов. Что нам понадобится в этом случае? Понятно, что необходимы

утилиты для создания кода на Java. Помимо этого, необходим браузер, поддерживающий технологию выполнения апплетов.

НА ЗАМЕТКУ

Браузеров существует достаточно много. У каждого браузера имеются свои настройки, в том числе и настройки безопасности. Выполнение программ, загруженных через сеть, считается потенциально опасным. Не исключено, что даже если браузер в принципе поддерживает Java-технологию, настройки браузера могут заблокировать такие возможности. Поэтому в начале работы с апплетами следует убедиться в наличии подходящего браузера и корректности его настроек. Как выполняются некоторые настройки, влияющие на использование Java-технологий (в частности, при работе с апплетами), описывается далее.

Нам понадобится, по меньшей мере, два «кода»: собственно код апплета и HTML-код документа, который содержит ссылку на апплет. Поэтому после того, как мы убедились в наличии необходимых средств разработки Java-кода и подходящего браузера, алгоритм создания апплета (в нашем случае) подразумевает реализацию следующих этапов.

- Создание программного кода апплета.
- Компиляция программного кода апплета.
- Создание HTML-документа со ссылкой на апплет.
- Открытие HTML-документа с помощью браузера.

Мы рассмотрим все эти этапы, но не совсем в той последовательности, как они приведены выше. Причина проста: понять принцип выполнения настроек проще, если есть некоторое представление о программном коде, который предполагается задействовать. После того, как мы в общих чертах опишем код и алгоритм создания апплета, мы вернемся к вопросу о выполнении настроек, разрешающих использование апплетов.

Чтобы понять идеологию использования в веб-документе апплета, можно представить апплет как некоторую «панель», которая включается в веб-документ. Но это не просто «панель», а функциональная панель. Ее функциональность, собственно, и определяется Java-кодом. В известном смысле, апплет — это Java-островок в море HTML-кода. Структура и свойства этого островка реализуется средствами Java, а инкапсулируется в веб-документ он с помощью HTML-кода. Веб-документ с этим

кодом открывается с помощью браузера, и если настройки безопасности браузера и исполнительной системы выполнены правильно (так что использование апплетов допускается), то внутри веб-документа мы получаем оазис Java-технологии.

Фактическое знакомство с апплетами начнем с того, что рассмотрим способы включения апплетов в веб-документ.

Добавление апплета в веб-документ

Документ с HTML-кодом представляет собой текст, содержащий специальные *дескрипторы*. Дескрипторы являются инструкциями для браузера и определяют способ отображения документа или отдельных его частей в окна браузера. Дескрипторы указываются внутри угловых скобок. Дескрипторы бывают парными и одинарными. Парные дескрипторы используются парой: есть открывающий дескриптор и закрывающий дескриптор. Например, документ с HTML-кодом начинается с дескриптора <html>, а заканчивается дескриптором </html>. Закрывающий дескриптор отличается от открывающего дескриптора наличием перед собственно названием дескриптора косой черты /. Это общее правило: если известно название открывающего дескриптора, то закрывающий дескриптор перед названием содержит косую черту. Скажем, блок текста, который следует выделить жирным шрифтом, выделяется дескрипторами и , для выделения текста курсивом используют дескрипторы и , а для выделения блока текста моноширинным шрифтом соответствующий блок выделяют дескрипторами <code> и </code>. Но нас, все же, интересуют апплеты, а не HTML-код. Поэтому мы будем использовать очень простую гипертекстовую разметку. В частности, веб-документы будут состоять из двух базовых блоков (оба они размещаются внутри документа, ограниченного дескрипторами <html> и </html>). Первый блок — блок заголовка (или заголовочный блок). Этот блок выделяется дескрипторами <head> и </head>. Второй блок — блок тела документа. Этот блок выделяется дескрипторами <body> и </body>. Основной код, в том числе и инструкция размещения апплета, размещается в блоке тела документа (<body>-блок). В блоке заголовка (<head>-блок) мы будем размещать блок названия документа. Блок названия документа размещается между дескрипторами <title> и </title>. Название документа отображается во вкладке браузера при открытии документа.

Таким образом, мы планируем использовать следующий шаблон для HTML-кода веб-документа:

```
<html>
  <head>
    <title>
      <!-- Название документа -->
    </title>
    <body>
      <!-- Блок тела документа -->
    </body>
  </head>
</html>
```

Стоит также заметить, что в HTML-коде все, что находится между инструкциями <!-- и -->, является комментарием.

Для вставки в веб-документ апплета используют дескриптор <applet>. Это открывающий дескриптор. Закрывающий дескриптор имеет вид </applet>. У дескриптора <applet> есть атрибуты. Атрибуты указываются в угловых скобках после имени дескриптора. Задаются атрибуты в формате *имя/значение*: указывается имя атрибута, а через знак равенства в двойных кавычках указывается значение атрибута. Нас интересует три атрибута. Значением атрибута code определяется название откомпилированного файла апплета. Также мы будем использовать атрибут width, определяющий ширину апплета, и атрибут height, задающий высоту апплета. Таким образом, речь идет о шаблоне вида:

```
<applet code="имя_файла" width="ширина" height="высота">
</applet>
```

Атрибуты можно указывать в произвольном порядке. Например, приведенный ниже код представляет собой инструкцию вставки в HTML-документ апплета, созданного на основе класса MyApplet.class (значение "MyApplet" для атрибута code), и этот апплет занимает область в 200 пикселов (значение "200" для атрибута height) в высоту и полностью всю ширину рабочей области окна браузера (значение "100%" для атрибута width):

```
<applet code="MyApplet" width="100%" height="200">
</applet>
```

В данном случае для атрибута code указано просто имя файла апплета (без расширения), хотя можно расширения файла указывать явно. При

этом подразумевается, что файл апплета находится в том же месте, где находится и файл веб-документа. Если файл апплета находится в месте, отличном от файла веб-документа, то в дескрипторе <applet> используется атрибут codebase, значением которого указывают путь к файлу апплета.

Приведенная выше команда по вставке апплета в блоке тела веб-документа означает, что в этом документе в соответствующем месте под апплет выделяется «полоска» на всю ширину рабочей области браузера. Высота «полоски» составляет 200 пикселов. Если класс апплета скомпилирован нормально, путь к классу указан правильно и настройки системы и браузера позволяют использовать апплеты, то в веб-документе появится апплет. Кокой именно — зависит от того, что мы напишем в программном коде, определяющем апплет.

Мы начнем с простого примера. Воспользуемся HTML-кодом, представленным в листинге 17.1.

Листинг 17.1. HTML-код файла MyFirstApplet.html

```
<html>
  <head>
    <title>
      Знакомимся с апплетами
    </title>
    <body>
      <h3>Веб-документ с апплетом</h3>
      Ниже представлен <b>апплет</b>.
      <hr>
      <applet code="MyApplet" width="100%" height="200">
      </applet>
      <hr>
      <i>Апплет создан с использованием класса</i> <code>JApplet</code>
    </body>
  </head>
</html>
```

В <title>-блоке документа указано название документа, которое отображается для документа при открытии его в браузере. Весь прочий код

относится к <body>-блоку. Между дескрипторами <h3> и </h3> размещается заголовок 3-го уровня (заголовок 1-го уровня выделяется дескрипторами <h1> и </h1>, заголовок 2-го уровня выделяется дескрипторами <h2> и </h2>, и так далее). Между дескрипторами и указывают блок текста, который необходимо выделить жирным шрифтом, для выделения текста курсивом используем пару дескрипторов и , а с помощью дескрипторов <code> и </code> к блоку текста применяется моноширинный шрифт (такой шрифт обычно используют для отображения программных кодов).

> **НА ЗАМЕТКУ**
>
> Дескриптор <code> не имеет никакого отношения к атрибуту code, используемому в дескрипторе <applet>. Это как раз тот случай, когда совпадение случайно.

Одинарный дескриптор <hr> является инструкцией вставки в веб-документ горизонтальной линии, которая по умолчанию отображается на всю ширину рабочей области окна браузера (то есть на всю ширину документа).

Также внутри <body>-блока есть команда добавления апплета. Речь о дескрипторах <applet> и </applet>. Соответствующий блок пустой, но внутри дескриптора <applet> с помощью инструкций code="MyApplet", width="100%" и height="200" определяются три атрибута: code, width и height. В соответствии со значениями, указанными для этих атрибутов, апплет реализуется на основе файла MyApplet.class, высота апплета составляет 200 пикселов, а ширина апплета совпадает с шириной рабочей области окна браузера. В принципе, если код апплета готов и откомпилирован, то достаточно открыть веб-документ, чтобы увидеть, как выглядит апплет «в жизни». Поэтому сейчас самое время обсудить способы создания программного кода для апплета.

Программный код апплета

Мы будем использовать подход, в рамках которого апплеты создаются на основе класса JApplet из библиотеки Swing. Общая используемая нами схема состоит в том, что на основе класса JApplet путем наследования создается public-класс, через который, собственно, и реализуется апплет. Еще раз подчеркнем, что для понимания сути происходящего имеет

смысл отождествлять апплет с некоторой панелью, только немножко «особенной». Вся «особенность» панели, в общем-то, сводится к тому, как панель используется. Например, если бы речь шла об обычной панели, то мы для нее создали бы объект и этот объект добавили, например, в окно. Окно, в свою очередь, создается в главном методе программы. Но особенность работы с апплетами в том, что метод main() не используется. Объяснение простое: апплет используется в веб-документе, поэтому все происходит под управлением браузера. Отсюда и «особый» способ обращения с апплетами. В частности, для апплета описывается метод init(). Метод init() вызывается автоматически при создании объекта апплета. В определенном смысле данный метод играет роль конструктора.

ДЕТАЛИ

Метод init() наследуется из класса JApplet. Но в классе JApplet у метода init() «пустая» реализация. Поэтому при наследовании в подклассе этот метод переопределяется. Есть еще несколько полезных при работе с апплетами методов, наследуемых из класса JApplet. Так, метод start() вызывается каждый раз при обращении к странице с апплетом. Метод stop() вызывается при уходе со страницы с апплетом, а метод destroy() автоматически вызывается перед завершением работы апплета.

В листинге 17.2 представлен программный код апплета, который мы собираемся использовать для отображения в веб-документе.

Листинг 17.2. Программный код проекта MyFirstApplet

```
// Импорт классов:
import javax.swing.*;
import java.awt.*;
// Класс апплета:
public class MyApplet extends JApplet{
  // Метод инициализации апплета:
  public void init(){
    // Создание метки:
    JLabel lbl=new JLabel("Синий текст на желтом фоне");
    // Применение шрифта к метке:
    lbl.setFont(new Font("Arial",Font.BOLD,30));
```

```
    // Цвет текста:
    lbl.setForeground(Color.BLUE);
    // Переход в режим непрозрачности:
    lbl.setOpaque(true);
    // Цвет фона метки:
    lbl.setBackground(Color.YELLOW);
    // Добавление метки в апплет:
    add(lbl);
  }
}
```

В данном случае на основе класса JApplet путем наследования создается класс MyApplet. Класс MyApplet является public-классом (описан с ключевым словом public, и это принципиальный момент).

> **НА ЗАМЕТКУ**
>
> Специфика public-классов в том, что в файле может быть только один public-класс, причем название файла должно совпадать с названием public-класса. Поэтому название файла, содержащего код класса для реализации апплета, должно совпадать с названием класса.
>
> При импорте пакетов из них импортируются только public-классы. Класс для реализации апплета описывается с ключевым словом public, поскольку класс предполагается использовать вне пределов его пакета.

В классе MyApplet описывается только метод инициализации апплета init(). В теле метода создается метка с текстом (команда JLabel lbl=new JLabel("Синий текст на желтом фоне")). Затем командой lbl.setFont(new Font("Arial",Font.BOLD,30)) для метки задается шрифт (шрифт определяется на основе анонимного объекта класса Font). Командой lbl.setForeground(Color.BLUE) к тексту метки применяется синий цвет. Командой lbl.setOpaque(true) для метки задается режим непрозрачности, при котором явно отображается область метки (по умолчанию область метки прозрачная). Это необходимо, поскольку мы хотим задать желтый цвет для фона метки. Желтый цвет для фона метки задаем командой lbl.setBackground(Color.YELLOW). Если при этом не перейти в режим непрозрачности, то фон останется прозрачным, и попытка задать для фона желтый цвет к желаемому результату не приведет. Наконец, с помощью команды add(lbl) метка добавляется в апплет.

НА ЗАМЕТКУ

Мы в программном коде явно не задаем размеры апплета, поскольку они определяются в HTML-документе через атрибуты width и height дескриптора <applet>. Для метки положение и размеры также явно не указаны. Но здесь следует учесть, что менеджер компоновки для апплета не отключается, поэтому при добавлении в апплет метки она автоматически масштабируется по размерам апплета. Так что в результате получается, что метка занимает всю область апплета.

Компиляция файла

Для реализации апплета мало набрать программный код класса, определяющего апплет. Необходимо этот файл скомпилировать. В принципе, можно компилировать файл апплета через командную строку. В таком случае в командной строке используется команда вида

```
javac.exe MyApplet.java
```

В результате, если программный код не содержит ошибок, получаем откомпилированный файл MyApplet.class, который следует разместить в том же каталоге, что и файл MyFirstApplet.html (веб-документ с апплетом).

Однако использовать командную строку — не самый удобный способ работы с Java-кодами. Поэтому кратко опишем возможную последовательность действий при создании апплетов с использованием среды разработки NetBeans. Итак, при работе со средой NetBeans начинаем с создания нового проекта (для этого можно воспользоваться, например, командой **New Project** из меню **File**). В результате открывается диалоговое окно **New Project**, представленное на рис. 17.1.

В этом окне в разделе **Categories** выбираем позицию **Java**, а в разделе **Projects** выбираем позицию **Java Application**. В следующем окне (рис. 17.2) **New Java Application** выполняем нужные настройки (название проекта и место хранения файлов проекта).

При этом главный класс создавать не нужно. Как следствие, флажок для опционного поля **Create Main Class** не устанавливаем и поле не заполняем (см. рис. 17.2). В результате создается пустой проект — в нем нет файлов. Нам следует файл добавить. Для этого в меню **File** выбираем команду **New File**, как показано на рис. 17.3.

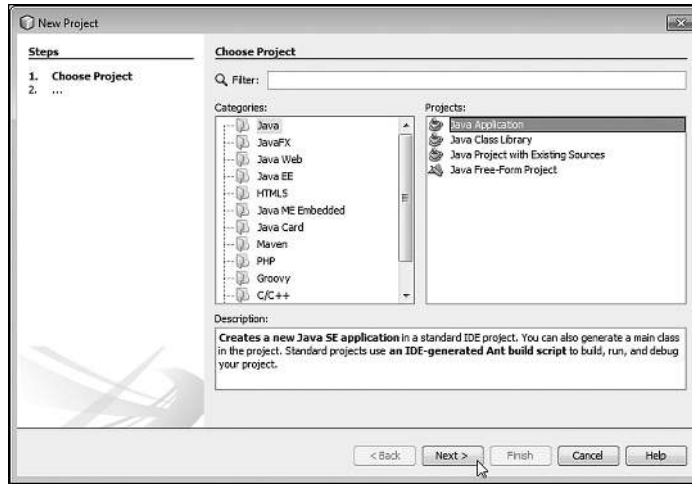


Рис. 17.1. Начальный этап создания апплета

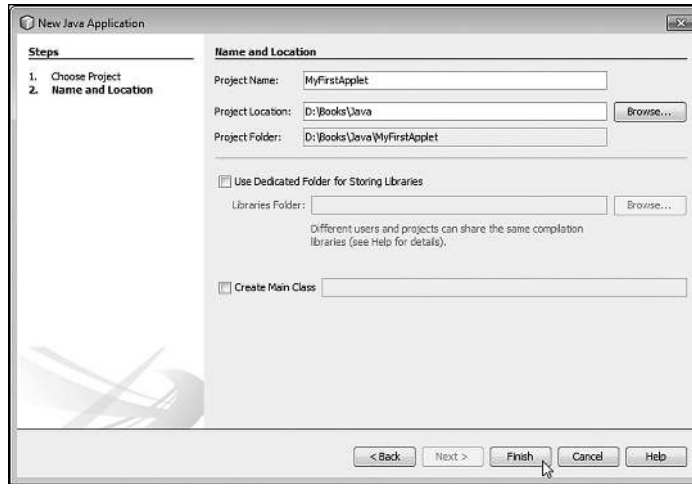


Рис. 17.2. При создании апплета главный класс не создается

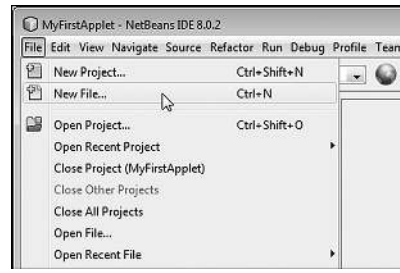


Рис. 17.3. Добавление файла в проект

Откроется окно **New File**, в котором в разделе **Categories** выбираем **Java**, а справа в разделе **File Class** можно выбрать позицию **JApplet** (хотя это и не принципиально). Процесс добавления файла в проект иллюстрируется на рис. 17.4.

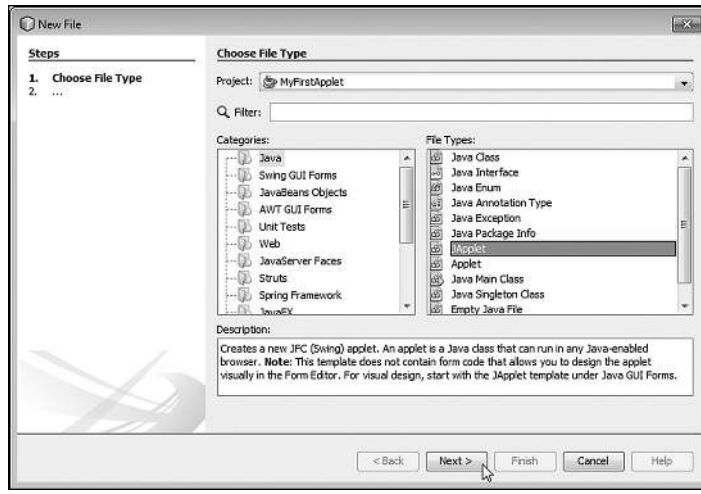


Рис. 17.4. Выбор типа добавляемого в проект файла

На следующем этапе необходимо указать название класса, описание которого будет содержаться в файле. Имя класса указывается в поле **Class Name**, как показано на рис. 17.5.

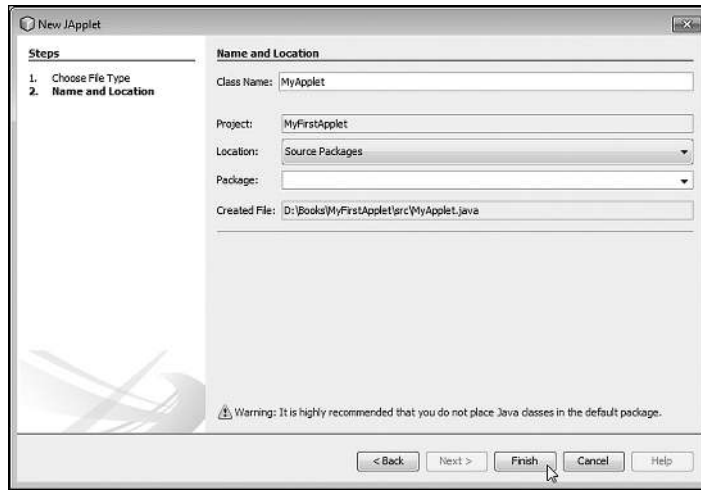


Рис. 17.5. Определение названия файла

После добавления файла в проект мы в окне среды разработки NetBeans в окне проектов выбираем созданный файл, а в правой части в окне редактора кодов вводим программный код. Окно среды разработки с редактором кодов, содержащим программный код из листинга 17.1, показано на рис. 17.6.

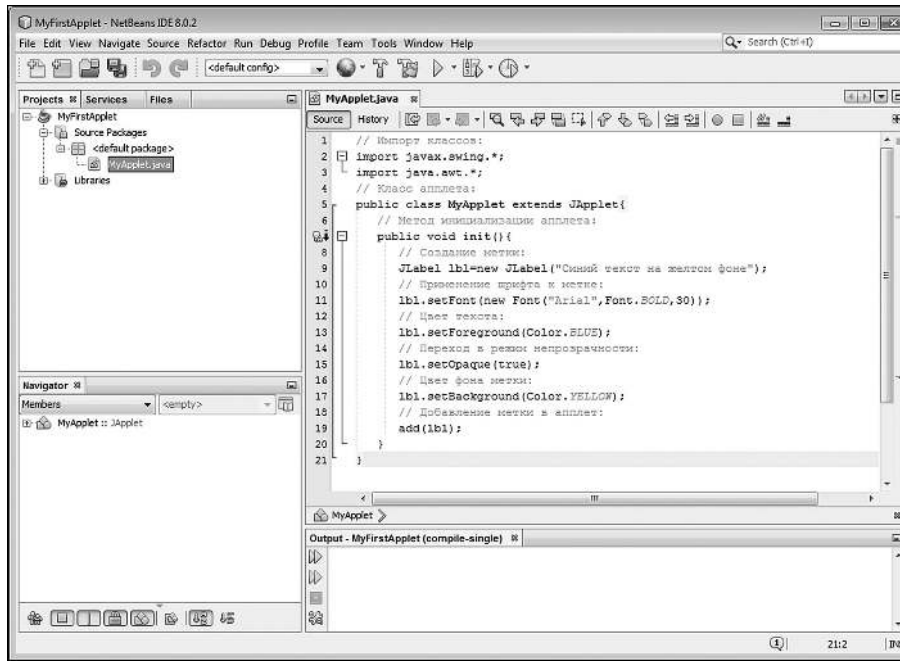


Рис. 17.6. Окно среды разработки NetBeans с кодом апплета

Теперь необходимо выполнить компиляцию файла. Самый простой способ состоит в том, чтобы в меню **Run** выбрать команду **Compile File**, как показано на рис. 17.7.

В таком случае если код не содержит ошибок, будет выполнена компиляция файла — что нам собственно и нужно. Полученный в результате компиляции файл по умолчанию находится в каталоге build\classes в папке проекта. В данном случае там должен появиться файл MyApplet.class. Данный файл следует скопировать в папку, в которой находится файл MyFirstApplet.html, после чего веб-документ с апплетом готов к использованию.

Но есть более «продвинутый» способ работы с апплетами: в меню **Debug** есть команда **Debug File** (рис. 17.8).

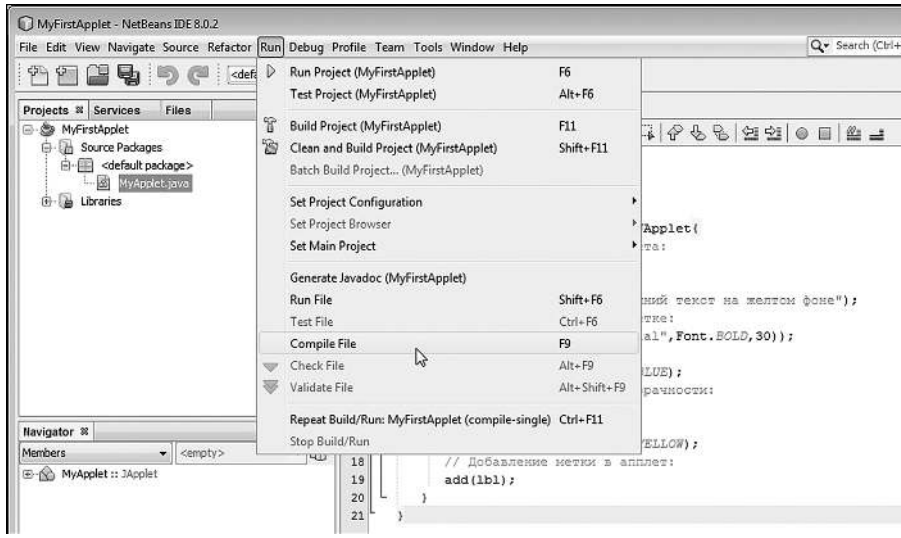


Рис. 17.7. Компиляция файла с кодом апплета

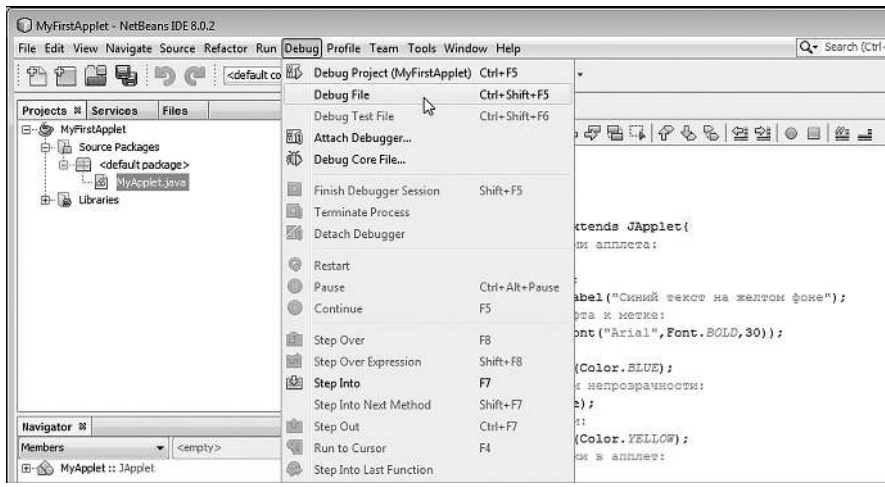


Рис. 17.8. Тестирование апплета

Данная команда позволяет выполнить предварительный просмотр апплета с помощью специальной утилиты просмотра апплетов. Утилита, при условии удачного компилирования файла, запускается автоматически.

На рис. 17.9 показано окно утилиты просмотра апплетов в процессе просмотра созданного нами апплета.

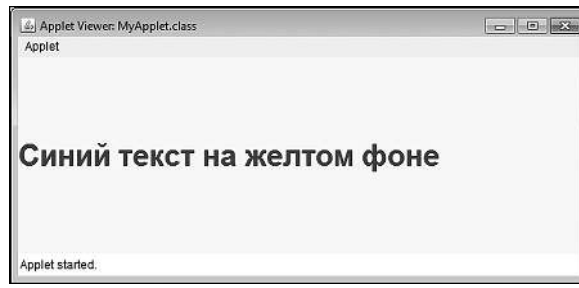


Рис. 17.9. Отображение апплета в окне просмотра апплетов

НА ЗАМЕТКУ

Также автоматически создается HTML-документ, в который «инкапсулируется» созданный апплет. Поэтому вид апплета можно оценить, открыв еще и данный HTML-документ. По умолчанию документ имеет такое же название, как и файл с кодом апплета (в данном случае файл с автоматически созданным HTML-документом называется MyApplet.html) и находится в папке build внутри папки проекта.

Как бы там ни было, после компиляции файла MyApplet.java копируем файл MyApplet.class в одну папку с файлом MyFirstApplet.html и открываем последний. Результат представлен на рис. 17.10.

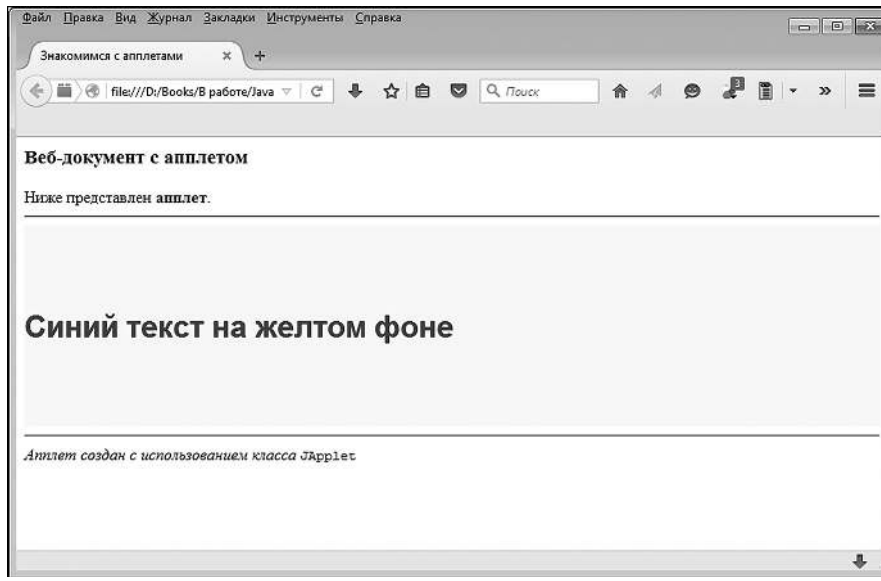


Рис. 17.10. В окне браузера открыт документ с апплетом

НА ЗАМЕТКУ

Можно пойти и иным путем: вместо копирования файла MyApplet.class в каталог с файлом MyFirstApplet.html указать в этом файле в значении атрибута codebase дескриптора <applet> путь к файлу MyApplet.class.

Настройки безопасности

Как отмечалось ранее, для использования апплетов необходимо убедиться в том, что настройки системы и браузера позволяют это сделать. Иначе можно получить неприятный сюрприз. Вариантов здесь достаточно много, мы лишь остановимся на двух моментах.

Во-первых, необходимо проверить настройки (и в первую очередь настройки безопасности) на *панели управления Java*. В разных операционных системах и для разных версий Java панель управления Java может открываться по-разному, но в итоге найти пути доступа к панели не так уж и сложно. На рис. 17.11 показано, как выглядит панель управления Java, открытая на вкладке **Security**.

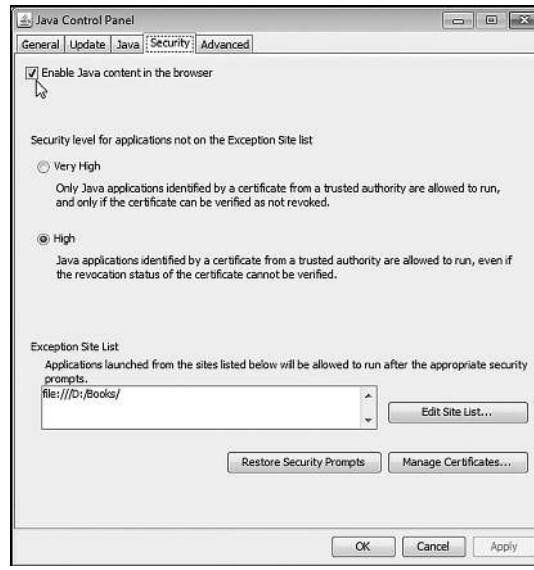


Рис. 17.11. Панель управления Java открыта на вкладке **Security**

В первую очередь необходимо проверить наличие флажка у опции **Enable Java content in the browser**. Также может понадобиться добавить в список

сайтов, для которых применяется исключение в виде разрешения использовать Java-технологии. Скажем, если веб-документ и используемые им классы для апплетов находятся в каталоге d:\books\, то с помощью кнопки **Edit Site List** добавляем в список исключений «сайт» file:///d:/books/.

Во-вторых, необходимо удостовериться, что в используемом браузере настройки позволяют использование апплетов. Но здесь все зависит от типа используемого браузера — то есть нужно обращаться к справке по конкретному браузеру. Например, для браузера *Mozilla Firefox* выполняется настройка плагина для работы с Java. Для этого в меню **Инструменты** выбираем команду **Дополнения**, в результате чего открывается вкладка **Управление дополнениями**, которую следует открыть в разделе **Плагины**, как показано на рис. 17.12.

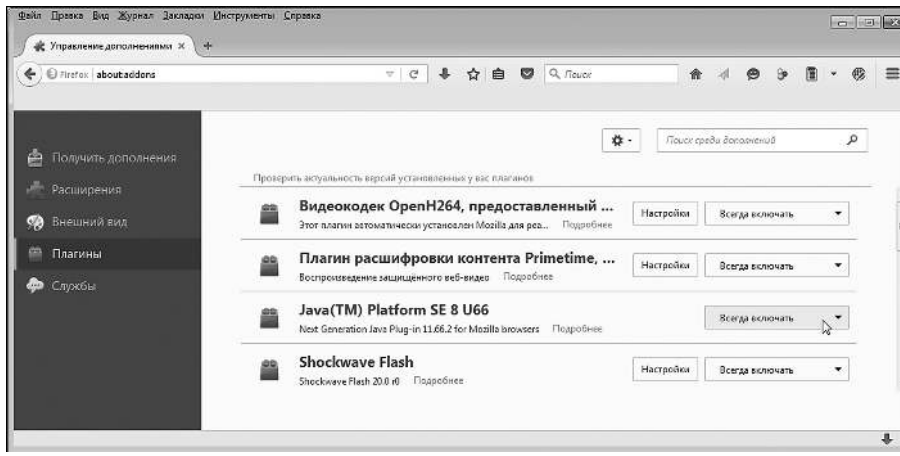


Рис. 17.12. В окне браузера открыта вкладка **Управление дополнениями**

На вкладке находим плагин для Java-платформы и выполняем нужные настройки.

Апплеты и обработка событий

— А что передать мой король?
— Передай твой король мой пламенный привет!
Из к/ф «Иван Васильевич меняет профессию»

Возвращаясь к результатам рассмотренного выше примера, следует отметить, что эффект от использования апплета не очень существенный:

закрасит область цветом и сделать цветную надпись в веб-документе можно и без использования апплетов. Сила апплетов — в их функциональности. А функциональность реализуется через обработку событий. Принципы обработки событий при работе с апплетами практически те же, что и при создании приложений с графическим интерфейсом (хотя конечно необходимо делать поправку на то, что речь идет об апплете).

Пример обработки событий в апплете

Далее мы рассмотрим небольшую модификацию предыдущего примера, но на этот раз добавим в апплет немного функциональности:

- размер метки в апплете меньше собственно размера апплета;
- при изменении размеров апплета (вследствие изменения размера окна браузера) автоматически изменяются размеры метки;
- вид метки в апплете меняется при наведении на область метки курсора мыши;
- при щелчке и удерживании нажатой кнопки (любой — правой или левой) мыши в области апплета текст в метке увеличивается в размере, а при отпускании кнопки мыши размер шрифта становится прежним.

Веб-документ с апплетом в данном случае практически такой же, как и в предыдущем случае. В листинге 17.3 представлен HTML-код веб-документа, который мы будем использовать для тестирования апплета (апплет в данном случае реализуется через файл MyNewApplet.class).

Листинг 17.3. HTML-код файла MySecondApplet.html

```
<html>
  <head>
    <title>
      Обработка событий
    </title>
    <body>
      <h3>Апплет с обработкой событий</h3>
      При наведении курсора мыши на апплет меняется цвет фона, текста и сам текст.
      <hr>
```

```
      <applet code="MyNewApplet" width="100%" height="200">
      </applet>
      <hr>
      <i>Апплет создан с использованием класса</i> <code>JApplet</code>
    </body>
  </head>
</html>
```

Фактически изменился только текст в веб-документе и название апплета (файла апплета). Программный код проекта, в котором реализуется интересующий нас апплет, представлен в листинге 17.4.

Листинг 17.4. Программный код проекта MySecondApplet

```
// Импорт классов:
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
// Класс апплета реализует интерфейсы
// MouseListener и ComponentListener:
public class MyNewApplet extends JApplet implements MouseListener,ComponentListener{
  // Поле — ссылка на метку:
  private JLabel L;
  // Поля для определения размера шрифта:
  private int small=25,big=35;
  // Название для шрифта:
  private String name="Arial";
  // Шрифт для метки, если над областью метки
  // находится курсор мыши:
  private Font onFnt=new Font(name,Font.ITALIC|Font.BOLD,small);
  // Шрифт для метки, если курсор мыши
  // не находится над областью метки:
  private Font offFnt=new Font(name,Font.BOLD,small);
  // Цвет для текста метки, если курсор мыши
```

```
// находится над областью метки:
private Color onFgr=Color.RED;
// Цвет для текста метки, если курсор мыши
// не находится над областью метки:
private Color offFgr=Color.BLUE;
// Цвет для фона метки, если курсор мыши
// находится над областью метки:
private Color onBgr=Color.WHITE;
// Цвет для фона метки, если курсор мыши
// не находится над областью метки:
private Color offBgr=Color.YELLOW;
// Текст для метки, если курсор мыши
// находится над областью метки:
private String onTxt="Красный текст на белом фоне";
// Текст для метки, если курсор мыши
// не находится над областью метки:
private String offTxt="Синий текст на желтом фоне";
// Метод для определения шрифта, текста,
// цвета текста и цвета фона для метки:
private void setAll(Color fgr,Color bgr,Font fnt,String txt){
  // Текст для метки:
  L.setText(txt);
  // Цвет текста для метки:
  L.setForeground(fgr);
  // Цвет фона для метки:
  L.setBackground(bgr);
  // Применение шрифта для метки:
  L.setFont(fnt);
}
// Метод вызывается при изменении размеров компонента
// (в данном случае апплета):
public void componentResized(ComponentEvent e){
```

```
    // Определение размеров метки:
    L.setSize(getWidth()-60,getHeight()-60);
  }
  // Неиспользуемые методы из интерфейса
  // ComponentListener с пустой реализацией:
  public void componentHidden(ComponentEvent e){}
  public void componentShown(ComponentEvent e){}
  public void componentMoved(ComponentEvent e){}
  // Метод вызывается при наведении курсора
  // мыши на апплет или метку в апплете:
  public void mouseEntered(MouseEvent e){
    // Если курсор наведен на метку:
    if(e.getSource()==L){
      // Присваивание текста, шрифта и цвета для
      // текста и фона метки:
      setAll(onFgr,onBgr,onFnt,onTxt);
    }
  }
  // Метод вызывается при "уходе" курсора
  // мыши из области апплета или метки в апплете:
  public void mouseExited(MouseEvent e){
    // Если курсор мыши "ушел" из области метки:
    if(e.getSource()==L){
      // Присваивание текста, шрифта и цвета для
      // текста и фона метки:
      setAll(offFgr,offBgr,offFnt,offTxt);
    }
  }
  // Метод вызывается при нажатии кнопки мыши:
  public void mousePressed(MouseEvent e){
    // Применение шрифта к метке:
    L.setFont(new Font(name,L.getFont().getStyle(),big));
```

```
}
// Метод вызывается при отпускании кнопки мыши:
public void mouseReleased(MouseEvent e){
  // Применение шрифта к метке:
  L.setFont(new Font(name,L.getFont().getStyle(),small));
}
// Неиспользуемый метод из интерфейса MouseListener
// с пустой реализацией:
public void mouseClicked(MouseEvent e){}
// Метод инициализации апплета:
public void init(){
  // Отключение менеджера компоновки:
  setLayout(null);
  // Создание метки:
  L=new JLabel();
  // Выравнивание текста по центру:
  L.setHorizontalAlignment(JLabel.CENTER);
  // Положение и размеры метки:
  L.setBounds(30,30,getWidth()-60,getHeight()-60);
  // Применение рамки вокруг метки:
  L.setBorder(BorderFactory.createEtchedBorder());
  // Переход в режим непрозрачности для метки:
  L.setOpaque(true);
  // Присваивание текста, шрифта и цвета для
  // текста и фона метки:
  setAll(offFgr,offBgr,offFnt,offTxt);
  // Регистрация обработчика событий
  // класса ComponentEvent в апплете:
  addComponentListener(this);
  // Регистрация обработчика событий
  // класса MouseEvent в апплете:
  addMouseListener(this);
```

```
    // Регистрация обработчика событий
    // класса MouseEvent в метке:
    L.addMouseListener(this);
    // Добавление метки в апплет:
    add(L);
  }
}
```

На рис. 17.13 показано окно браузера, в котором открыт веб-документ с апплетом.

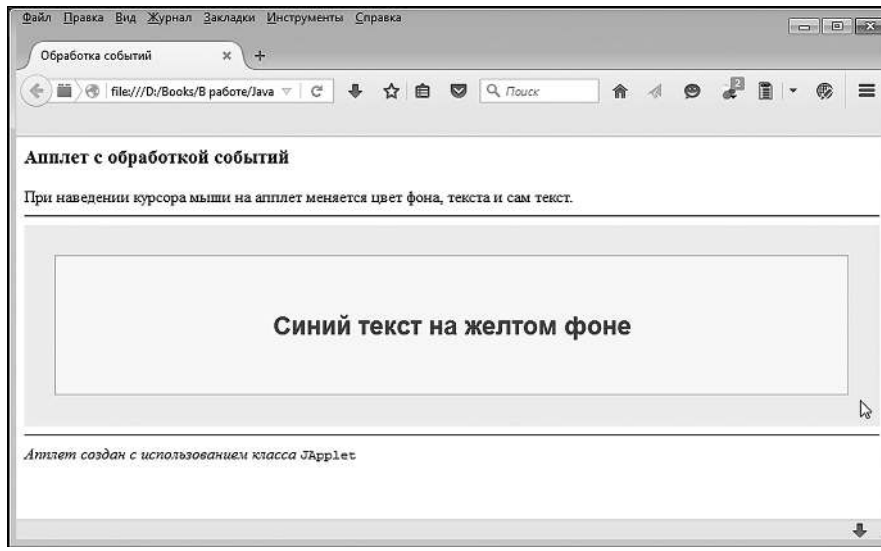


Рис. 17.13. При отображении окна, если курсор мыши не наведен на область метки, фон у метки желтый, а цвет текста метки синий

Ширина апплета определяется шириной рабочей области окна. По умолчанию область апплета закрашена в светло серый цвет. Внутри апплета есть метка, размеры которой теперь меньше размеров апплета. Фон области метки желтый, а текст отображается синим цветом. Если щелкнуть и удерживать кнопку мыши в области апплета, но вне области метки, то в таком случае увеличивается размер шрифта, как показано на рис. 17.14.

При отпускании кнопки размер шрифта восстанавливается. Если же навести курсор мыши на метку, то фон метки станет белым, а текст

становится красным и курсивным. Изменится и само текстовое значение. Ситуация проиллюстрирована на рис. 17.15. Там показано, как выглядит метка в апплете, когда курсор находится над областью метки.

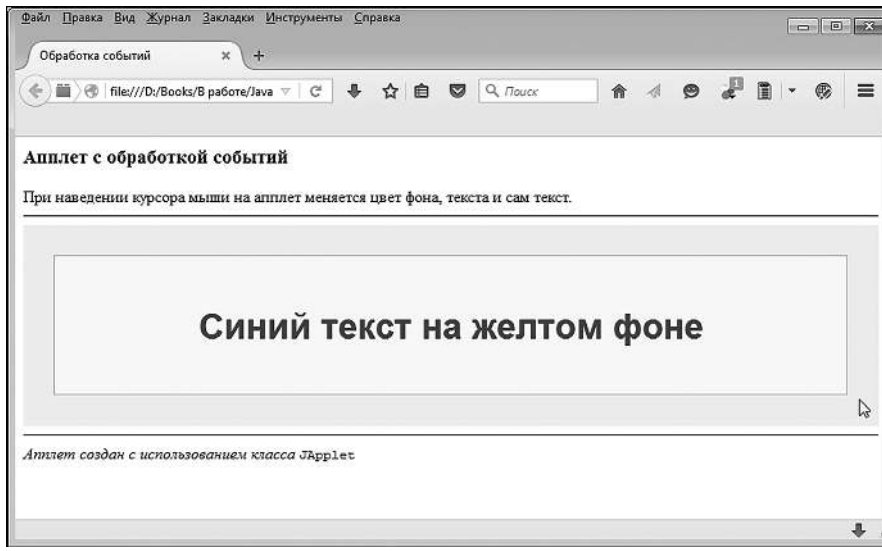


Рис. 17.14. При нажатии и удерживании кнопки мыши в области апплета, но вне области метки увеличивается размер шрифта метки

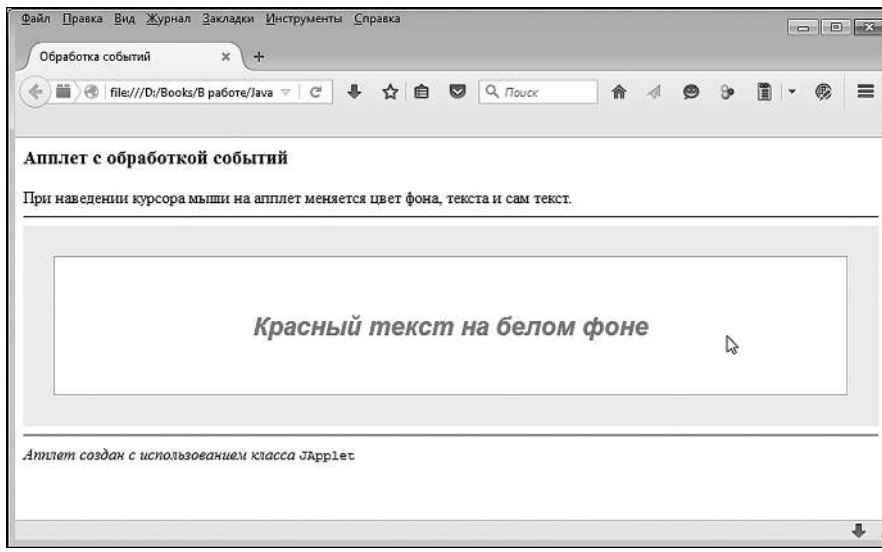


Рис. 17.15. При наведении курсора мыши на метку фон метки становится белым, а текст становится красным

Если в области метки нажать кнопку мыши и удерживать ее, то размер текста увеличивается, как это видно на рис. 17.16.

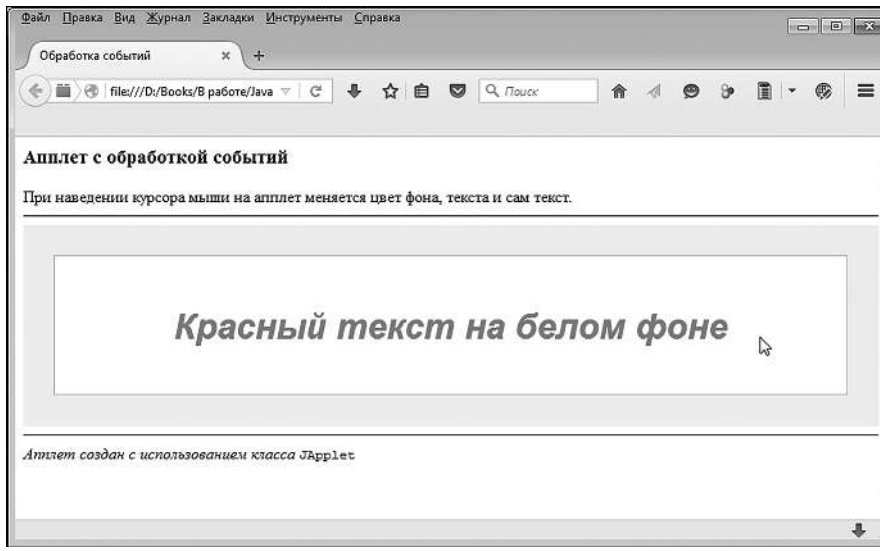


Рис. 17.16. При нажатии и удерживании кнопки мыши в области метки увеличивается размер шрифта метки

Если кнопку мыши отпустить, то размер текста вернется к исходному значению. Еще одна особенность проекта в том, что изменение размера окна, приводящее к изменению ширины апплета (высота апплета фиксирована), приводит к изменению размера метки. На рис. 17.17 показано, как изменится размер метки (вместе с размером апплета) при уменьшении ширины окна браузера. Размер текста остается постоянным, а размеры метки изменяются так, то между границей метки и границей апплета есть «зазор» шириной в 30 пикселов.

В классе апплета MyNewApplet реализуются интерфейсы MouseListener и ComponentListener. Мы так поступаем, поскольку хотим, чтобы объект апплета мог быть обработчиком для событий, связанных с манипуляциями мышью и событиями, связанными с манипуляциями с размерами апплета. В классе апплета объявляется несколько полей.

- Закрытое поле L является ссылкой на метку — объект класса JLabel.
- Целочисленные закрытые поля small и big определяют размер шрифта в области метки соответственно в обычном состоянии и при нажатой кнопке мыши (когда текст увеличивается в размере).

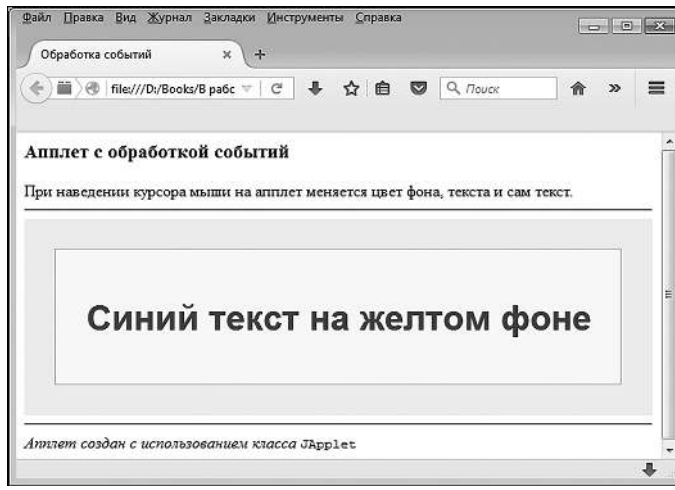


Рис. 17.17. При изменении размера окна вместе с размером апплета автоматически изменяется размер метки

- Закрытое текстовое поле name определяет название типа шрифта, применяемого к метке.
- Закрытые поля onFnt и offFnt (объектные ссылки класса Font) определяют шрифт для текста метки в случаях, если курсор мыши наведен на метку и когда курсор находится вне области метки.
- Цвет для текста и фона метки определяется закрытыми полями onFgr (цвет текста при наведенном на область метки курсоре), offFgr (цвет текста при не наведенном на область метки курсоре), onBgr (цвет фона при наведенном на область метки курсоре) и offBgr (цвет фона при не наведенном на область метки курсоре). Все эти поля являются объектными ссылками класса Color.
- Закрытые текстовые поля onTxt и offTxt определяют текст для метки в случае, если курсор мыши наведен на область метки и если курсор находится вне области метки.

Удобства ради, в классе описывается закрытый метод setAll(), предназначенный для определения таких параметров метки L, как цвет текста (первый аргумент метода), цвет фона (второй аргумент метода), шрифт (третий аргумент метода) и собственно текст для метки (четвертый аргумент метода).

В методе init() инициализации апплета командой setLayout(null) для апплета отключается менеджер компоновки. Командой L=new JLabel() создается

метка (пока пустая), для нее командой L.setHorizontalAlignment(JLabel.CENTER) задается способ выравнивания текст по центру в области метки. Инструкцией L.setBorder(BorderFactory.createEtchedBorder()) для метки применяется рамка, командой L.setOpaque(true) метка переводится в режим непрозрачности, а с помощью команды setAll(offFgr,offBgr,offFnt,offTxt) для метки задается текст, шрифт, а также цвет для текста и фона метки.

Положение и размеры метки задаем командой L.setBounds(30,30,getWidth()-60,getHeight()-60). Поскольку метка в итоге инструкцией add(L) добавляется в апплет, то координаты метки определяются по отношению к апплету. В данном случае при размещении метки в апплете делается отступ в 30 пикселов по вертикали и горизонтали от левого верхнего угла апплета. Методы getWidth() и getHeight() вызывается из объекта апплета и результатом возвращают соответственно ширину и высоту апплета (на момент вызова методов). Высота метки инструкцией getHeight()-60 определяется на 60 пикселов меньше высоты апплета. Ширина метки вычисляется инструкцией getWidth()-60. Поэтому при создании метки ее шири на 60 пикселов меньше ширины апплета.

НА ЗАМЕТКУ

Напомним, что в файле MySecondApplet.html высота апплета задана в 200 пикселов, а ширина апплета установлена равной 100% от ширины рабочей области окна. Поэтому при изменении размеров окна браузера высота апплета остается неизменной, а ширина меняется.

Проблема, однако, в том, что ширина апплета определяется размерами окна браузера, и этот размер в процессе работы с веб-документом может меняться. При этом ширина метки автоматически меняться не будет. Чтобы при изменении ширины апплета синхронно изменялась и ширина метки, командой addComponentListener(this) в апплете объект апплета регистрируется обработчиком для событий класса ComponentEvent, связанных в том числе с изменением размеров апплета.

Из всех методов интерфейса ComponentListener в классе апплета описывается с непустым телом только метод componentResized(), вызываемый для обработки события, связанного с изменением размера апплета. В теле метода выполняется команда L.setSize(getWidth()-60,getHeight()-60), которой ширина и высота метки устанавливается на 60 пикселов меньше соответственно текущей ширины и высоты апплета.

Прочие методы интерфейса ComponentListener (метод componentHidden() вызывается при скрытии компонента, метод componentShown() вызывается при отображении компонента и метод componentMoved() вызывается при перемещении компонента) описываются с пустым телом.

Кроме событий класса ComponentEvent, выполняется еще и обработка событий класса MouseEvent (манипуляции с мышью). Но дело в том, что обработчиком событий данного класса регистрируется объект апплета, и регистрируется и для самого апплета (команда addMouseListener(this)), и для метки (команда L.addMouseListener(this)). В интерфейсе MouseListener всего пять абстрактных методов, из которых мы описываем четыре: не у дел остается метод mouseClicked(), который вызывается для обработки щелчка кнопкой мыши — этот метод описывается с пустым телом.

Метод mousePressed() вызывается при нажатии, а метод mouseReleased() — при отпускании кнопки мыши в области компонента, в котором зарегистрирован соответствующий обработчик. Обработчик зарегистрирован и в апплете, и в метке. Выполняемые при вызове методов операции не зависят от того, на каком компоненте произошло событие. Поэтому в теле указанных методов нет необходимости идентифицировать компонент (собственно апплет или метка), на котором произошло событие. Отсюда и такой простой код методов. В теле метода mousePressed() выполняется команда L.setFont(new Font(name,L.getFont().getStyle(),big)), которой для метки применяется шрифт с именем, определяемым полем name, текущим стилем и размером, определяемым значением поля big. Стиль текущего шрифта для метки получаем с помощью выражения L.getFont().getStyle(), в котором метод getStyle() вызывается из объекта шрифта метки, а ссылка на объект шрифта возвращается с помощью инструкции L.getFont(). Аналогично, в теле метода mouseReleased() с помощью команды L.setFont(new Font(name,L.getFont().getStyle(),small)) к метке применяется шрифт размера, определяемого значением поля small.

С методами mouseEntered() (вызывается при наведении курсора на область компонента, в котором зарегистрирован обработчик) и mouseExited() (вызывается при перемещении курсора за пределы области компонента, в котором зарегистрирован обработчик) ситуация сложнее: данные методы должны выполнять изменение параметров метки только если курсор наведен на область метки и, соответственно, покидает область метки. Проблема в том, что обработчик с данными методами зарегистрирован и для метки, и для апплета. Решение проблемы находим в том, что в каждом из методов в условном операторе ссылка e.getSource() (через

e обозначен аргумент метода) на объект, на котором произошло событие, сравнивается со ссылкой на метку L. Если ссылки совпадают, то вызывается метод setAll(), которым задаются нужные параметры для метки. В противном случае (если ссылки разные) не происходит ничего.

Передача апплету параметров

В апплет из веб-документа могут передаваться параметры. Проще говоря, мы можем непосредственно в HTML-коде указать некоторые значения, которые считываются апплетом. Для передачи параметров апплету в блоке, выделенном дескрипторами <applet> и </applet>, размещается дескриптор <param>. Внутри дескриптора указывается два атрибута: name и value. Значение атрибута name определяет имя параметра, а значение атрибута value определяет значение параметра. Таким образом, если мы хотим использовать в апплете параметр с некоторым именем, и задаем для этого параметра определенное значение, то в блоке апплета используется инструкция <param name=имя value=значение>.

> **НА ЗАМЕТКУ**
>
> Обычно название параметра и его значение заключается в двойные кавычки. Дескриптор <param> является одинарным, поэтому для него закрывающего дескриптора нет.

Шаблон для кода вставки в веб-документ апплета с параметрами выглядит следующим образом:

```
<applet атрибуты>
  <param name=имя_параметра value=значение_параметра>
</applet>
```

Если апплету передается несколько параметров, то для каждого параметра используется отдельный дескриптор <param>. Для считывания значения параметра в апплете используется метод getParameter(). Аргументом методу передается текстовое значение с названием параметра. Результатом метод возвращает текстовое значение, определяющее значение параметра.

Таким образом, к какому бы типу по своей природе ни относилось значение, оно все равно считывается как текст. Поэтому обычно после

считывания значения параметра, его текстовое представление необходимо преобразовать в тот формат, который подразумевается при использовании параметра.

Далее рассматривается небольшой пример использования апплета с параметрами. Фактически речь идет о модификации предыдущего примера. То есть в общем и целом решается та же задача. Но теперь цвет фона для метки, когда на нее не наведен курсор, определяется параметром, передаваемым в апплет. Также через параметр задается размер шрифта для текста метки.

Кроме этого, мы использовали несколько иную схему обработки событий: вместо реализации интерфейсов MouseListener и ComponentListener в классе апплета задействованы анонимные объекты обработчиков, которые создаются на основе анонимных классов, наследующих классы-адаптеры MouseAdapter и ComponentAdapter.

В веб-документе используется два однотипных апплета (реализуются через разные объекты, но созданные на основе одного класса). Каждому из апплетов передаются параметры, но для второго апплета команды, связанные с определением параметров, некорректны. Но к ошибке это не приводит, поскольку в классе апплета при считывании параметров применяется обработка исключительных ситуаций.

Рассмотрим HTML-код, представленный в листинге 17.5.

Листинг 17.5. HTML-код файла MyThirdApplet.html

```
<html>
  <head>
    <title>
      Передача параметров апплету
    </title>
    <body>
      <h3>Передача апплету параметров</h3>
      Апплету передаются параметры: цвет для текста метки и размер шрифта.<br>
      <b>Первый апплет:</b>
      <hr>
      <applet codebase="MyThirdApplet/build/classes" code="MyParamApplet.class"
width="100%" height="150">
```

```
        <param name="color" value="Черный">
        <param name="fontsize" value="35">
      </applet>
      <hr>
      <b>Второй апплет:</b>
      <hr>
      <applet codebase="MyThirdApplet/build/classes" code="MyParamApplet.class"
width="75%" height="25%">
        <param name="color" value="Малиновый">
      </applet>
      <hr>
      <i>Апплеты созданы с использованием класса</i> <code>JApplet</code>
    </body>
  </head>
</html>
```

Кроме уже упомянутых особенностей кода, отметим еще несколько моментов. Предполагается, что апплету передается два параметра: параметр color определяет цвет текста, отображаемого в меткие (если на нее не наведен курсор мыши), а параметр fontsize задает размер шрифта для текста метки (если в области апплета не удерживается нажатой кнопка мыши). Для первого апплета параметры определяются с помощью дескрипторов <param name="color" value="Черный"> и <param name="fontsize" value="35">. Ширина первого апплета устанавливается равной ширине рабочей области окна браузера (значение 100%), а высота первого апплета равна 150 пикселов. Для второго апплета ширина устанавливается равной 75% от ширины рабочей области окна браузера, а высота второго апплета составляет 25% от высоты рабочей области. Для второго апплета определен только один параметр (инструкция <param name="color" value="Малиновый">). Такая ситуация в общем случае приводит к ошибке. На примере второго апплета мы опробуем обработку исключительных ситуаций в апплете.

В обоих апплетах использованы атрибуты code со значением "MyParamApplet.class" (имя файла с откомпилированным кодом для апплета) и codebase со значением "MyThirdApplet/build/classes" (каталог, в котором хранятся файлы для апплета).

НА ЗАМЕТКУ

Поскольку в данном случае используется апплет с параметрами, то при компиляции файла с апплетом создаются и некоторые дополнительные файлы, которые необходимы при использовании апплета в веб-документе. Хотя ссылка выполняется только непосредственно на файл апплета.

Программный код для класса апплета представлен в листинге 17.6.

Листинг 17.6. Программный код проекта MyThirdApplet

```
// Импорт классов:
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
// Класс апплета:
public class MyParamApplet extends JApplet{
  // Поле — ссылка на метку:
  private JLabel L;
  // Поля для определения размера шрифта (определяются
  // на основе параметра, переданного апплету):
  private int small,big;
  // Название для шрифта:
  private String name="Arial";
  // Шрифты для метки:
  private Font onFnt,offFnt;
  // Цвет для текста метки:
  private Color onFgr=Color.RED;
  // Значение поля определяется на основе параметра,
  // переданного апплету:
  private Color offFgr;
  // Цвет для фона метки:
  private Color onBgr=Color.WHITE;
  private Color offBgr=Color.YELLOW;
```

```
  // Текст для метки:
  private String onTxt="Красный текст на белом фоне";
  // Значения полей определяются на основе параметра,
  // переданного апплету:
  private String offTxt,clr;
  // Метод для определения параметров метки:
  private void setAll(Color fgr,Color bgr,Font fnt,String txt){
    // Текст для метки:
    L.setText(txt);
    // Цвет текста для метки:
    L.setForeground(fgr);
    // Цвет фона для метки:
    L.setBackground(bgr);
    // Применение шрифта для метки:
    L.setFont(fnt);
  }
 // Метод инициализации апплета:
  public void init(){
    // Контролируемый код. Считывание параметра апплета:
    try{
      // Значение параметра:
      clr=getParameter("color");
      // Проверка значения:
      if(clr.equalsIgnoreCase("розовый")){
        offFgr=Color.PINK;
      }
      else{
        if(clr.equalsIgnoreCase("зеленый")){
          offFgr=Color.GREEN;
        }
        else{
          if(clr.equalsIgnoreCase("черный")){
```

```
          offFgr=Color.BLACK;
        }
        else{
          clr="Синий";
          offFgr=Color.BLUE;
        }
      }
    }
  }
  // Обработка исключения:
  catch(Exception e){
    clr="Синий";
    offFgr=Color.BLUE;
  }
  // Контролируемый код. Считывание параметра апплета:
  try{
    // Размер шрифта:
    small=Integer.parseInt(getParameter("fontsize"));
  }
  // Обработка исключения:
  catch(Exception e){
    small=20;
  }
  // Увеличенный размер шрифта:
  big=small+10;
  // Определение шрифтов:
  onFnt=new Font(name,Font.ITALIC|Font.BOLD,small);
  offFnt=new Font(name,Font.BOLD,small);
  // Определение текста для метки:
  offTxt=clr+" текст на желтом фоне";
  // Отключение менеджера компоновки:
  setLayout(null);
```

```
// Создание метки:
L=new JLabel();
// Выравнивание текста по центру:
L.setHorizontalAlignment(JLabel.CENTER);
// Положение и размеры метки:
L.setBounds(30,30,getWidth()-60,getHeight()-60);
// Применение рамки вокруг метки:
L.setBorder(BorderFactory.createEtchedBorder());
// Переход в режим непрозрачности для метки:
L.setOpaque(true);
// Определение параметров метки:
setAll(offFgr,offBgr,offFnt,offTxt);
// Регистрация обработчика событий
// класса ComponentEvent в апплете:
addComponentListener(new ComponentAdapter(){
  // Метод вызывается при изменении
  // размеров апплета:
  public void componentResized(ComponentEvent e){
    // Размеры метки:
    L.setSize(getWidth()-60,getHeight()-60);
  }
});
// Регистрация обработчика событий
// класса MouseEvent в апплете:
addMouseListener(new MouseAdapter(){
  // Метод вызывается при нажатии кнопки мыши:
  public void mousePressed(MouseEvent e){
    // Применение шрифта к метке:
    L.setFont(new Font(name,L.getFont().getStyle(),big));
  }
  // Метод вызывается при отпускании кнопки мыши:
  public void mouseReleased(MouseEvent e){
    // Применение шрифта к метке:
```

```
        L.setFont(new Font(name,L.getFont().getStyle(),small));
      }
    });
    // Регистрация обработчика событий
    // класса MouseEvent в метке:
    L.addMouseListener(new MouseAdapter(){
      // Метод вызывается при наведении курсора
      // мыши на метку:
      public void mouseEntered(MouseEvent e){
        // Параметры метки:
        setAll(onFgr,onBgr,onFnt,onTxt);
      }
      // Метод вызывается при "уходе" курсора
      // мыши из области метки:
      public void mouseExited(MouseEvent e){
        // Параметры метки:
        setAll(offFgr,offBgr,offFnt,offTxt);
      }
    });
    // Регистрация еще одного обработчика событий
    // класса MouseEvent в метке:
    L.addMouseListener(getMouseListeners()[0]);
    // Добавление метки в апплет:
    add(L);
  }
}
```

Как выглядит окно браузера с открытым в нем веб-документом с апплетами, показано на рис. 17.18.

Каждый из двух апплетов в документе обладать «характеристиками» апплета, таким же, как в предыдущем примере. Скажем, если навести курсор на область метки любого из апплетов, текст станет красным и курсивным, а область метки — белой, как показано на рис. 17.19.

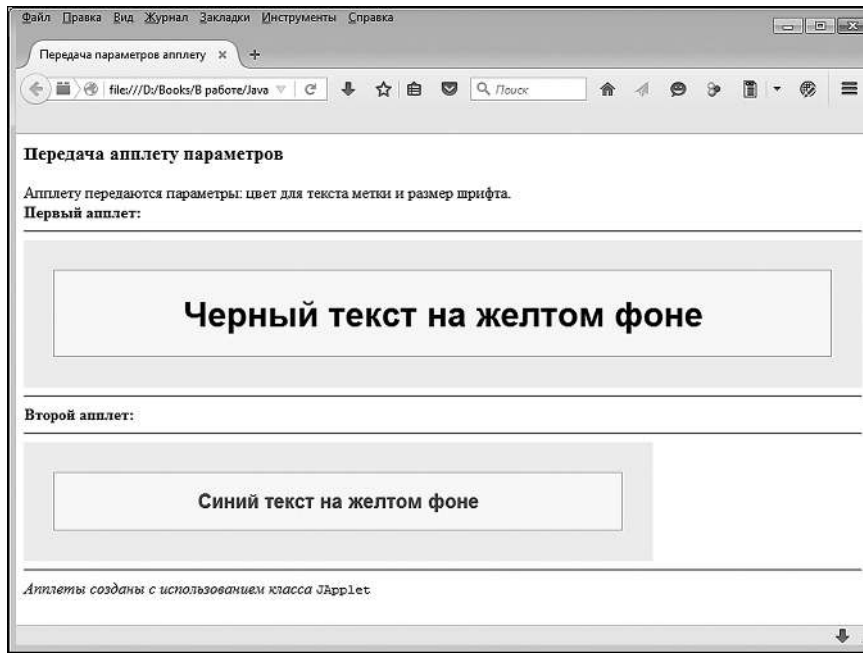


Рис. 17.18. Окно браузера с открытым в нем документом с двумя однотипными апплетами

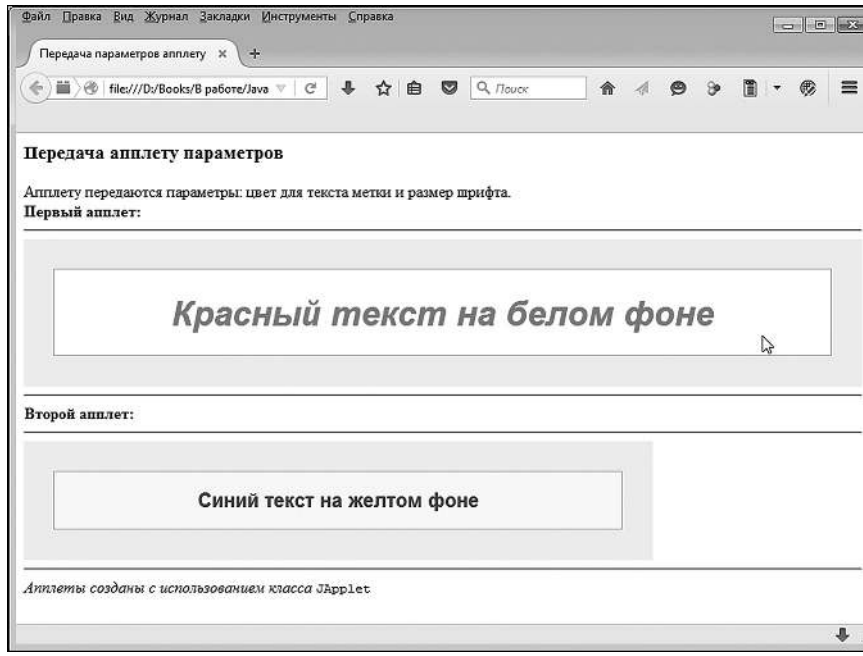


Рис. 17.19. Курсор мыши наведен на область метки в первом апплете

Если нажать и удерживать кнопку мыши (когда курсор находится над областью метки), текст увеличивается в размере (рис. 17.19), а при отпускании кнопки мыши возвращается к исходному размеру.

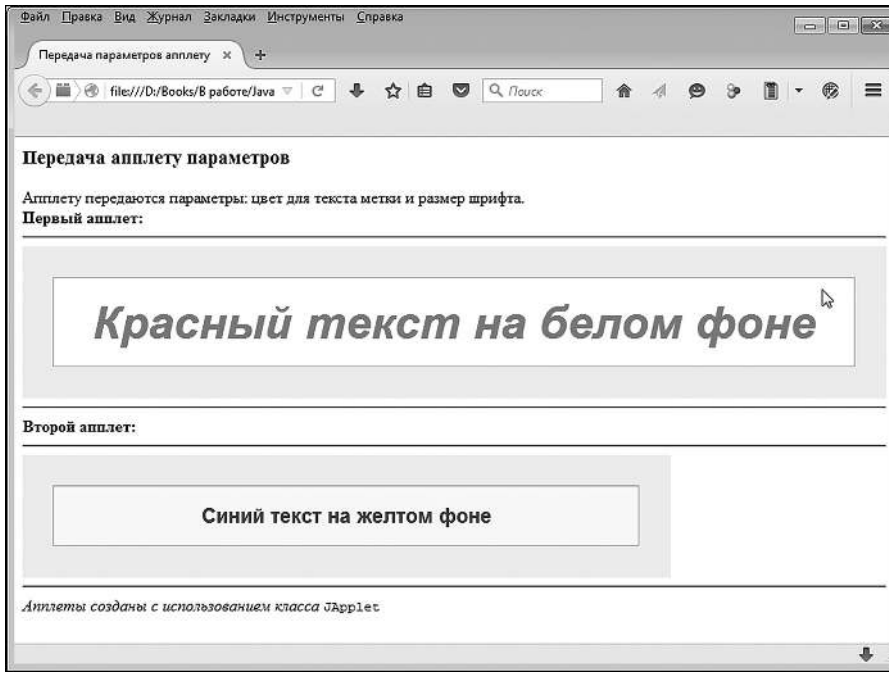


Рис. 17.20. При удерживании кнопки мыши в нажатом состоянии (когда курсор находится над областью метки) увеличивается размер текста в метке

Если удерживать нажатой кнопку мыши, когда курсор находится над областью апплета, но вне области метки, шрифт также увеличивается (рис. 17.21).

Что касается код программы (см. листинг 17.5), то многие команды должны быть понятны читателю — возможно, за исключением нескольких. Их и прокомментируем.

Как отмечалось, для считывания значения параметров используется метод getParameter(). Например, инструкцией getParameter("color") считывается и возвращается результатом значение (текстовое) параметра color. Результат записывается в текстовую переменную clr. Текст проверяется на предмет совпадения с названием одного из цветов, и при наличии совпадения соответствующее значение присваивается полю offFgr. Если совпадение не найдено или если возникла ошибка, используется синий цвет.

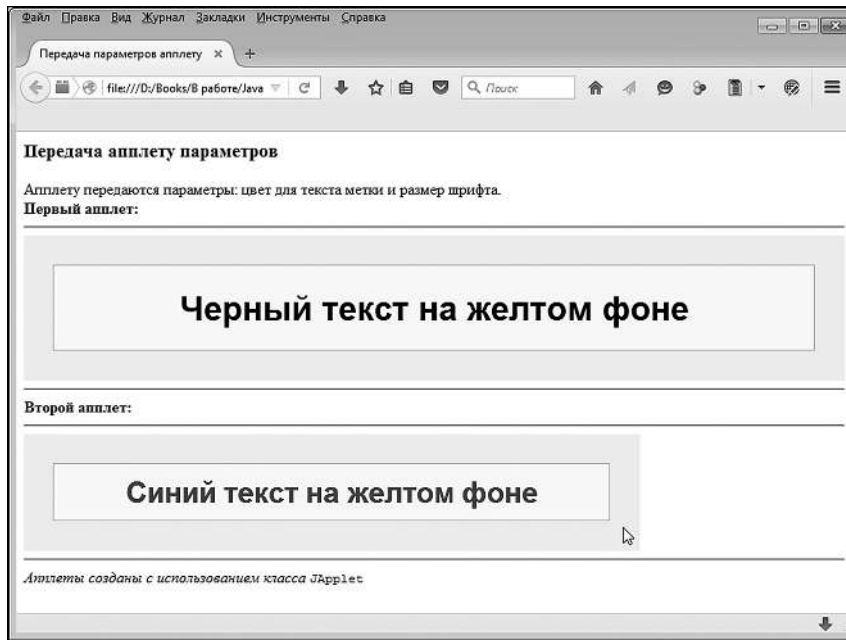


Рис. 17.21. При удерживании кнопки мыши в нажатом состоянии (когда курсор мыши находится над областью апплета) увеличивается размер текста в метке этого апплета

Инструкцией getParameter("fontsize") возвращается значение (текстовое) для размера шрифта (значение параметра fontsize). Мы предполагаем, что это число, поэтому инструкция указана аргументом метода Integer.parseInt() для преобразования текста в число, а результат преобразования записывается в поле small. Если в процессе получения значения параметра возникла ошибка, для поля small используется значение 20. В любом случае значение поля big вычисляется как small+10.

При регистрации обработчиков событий в апплете и метке мы используем нехитрый, но показательный прием, суть которого сводится к следующему.

- Для апплета обработчик событий класса ComponentEvent создается на основе анонимного класса, наследующего класс-адаптер ComponentAdapter. В этом классе все методы из интерфейса ComponentListener описаны с пустым кодом. Благодаря этому при создании обработчика мы ограничиваемся описанием лишь метода componentResized().
- Обработчик событий класса MouseEvent в апплете создается на основе анонимного класса, реализующего класс-адаптер MouseAdapter. При

создании обработчика описываются только методы mousePressed() и mouseReleased().

- При создании на основе анонимного класса (наследующего класс-адаптер MouseAdapter) и регистрации обработчика событий класса MouseEvent в метке описываются только методы mouseEntered() и mouseExited(). Но это не все. Дело в том, что следует еще и определить методы mousePressed() и mouseReleased(), причем определяются они так же, как при создании и регистрации соответствующего обработчика для апплета. Чтобы не дублировать дважды одинаковый код, мы для метки регистрируем два обработчика, причем оба — для событий класса MouseEvent. Первый обработчик создается явно (с использованием класса-адаптера), а второй «получаем» из апплета. Для этого из апплета вызывается метод getMouseListeners(). Результатом метод возвращает ссылку на массив из обработчиков событий класса MouseEvent, зарегистрированных в апплете. Мы точно знаем, что обработчик там один, и это именно тот обработчик, который нам нужен. Ссылку на данный объект-обработчик получаем с помощью выражения getMouseListeners()[0]. Выражение getMouseListeners()[0] передается аргументом методу addMouseListener(), вызываемому из метки L для регистрации в ней еще одного обработчика.

НА ЗАМЕТКУ

Вообще, при использовании команды **Debug File** из меню **Debug** среды NetBeans если речь идет об апплете с параметрами, выполнить предварительный просмотр в силу очевидных причин проблематично.

Однако в данном случае, поскольку в классе апплета выполняется обработка ошибок, в результате чего для параметров фактически определяются значения по умолчанию, предварительный просмотр апплета доступен.

Апплет с элементами управления

Апплет может содержать элементы управления, такие как кнопки, списки, опции, переключатели, меню и ряд других. Рассмотрим небольшой пример, в котором используется апплет с элементами управления.

Для проверки работы апплета используем HTML-документ, код которого представлен в листинге 17.7.

Листинг 17.7. HTML-код файла MyFourthApplet.html

```
<html>
  <head>
    <title>
      Апплет с графическими компонентами
    </title>
    <body>
      <h3>Апплет с графическими компонентами</h3>
      Апплет содержит <b>графические компоненты</b>.<br>
      <hr>
      <applet codebase="MyFourthApplet/build/classes" code="MyGUIApplet.class"
width="350" height="210">
      </applet>
      <hr>
      <i>Апплет создан с использованием класса</i> <code>JApplet</code>
    </body>
  </head>
</html>
```

Особенность ситуации в том, что мы указали фиксированные размеры апплета в пикселах. Прежде, чем приступить к разбору программного кода апплета, рассмотрим его функциональные возможности. На рис. 17.22 показано, как выглядит окно браузера, в котором открыт документ с апплетом.

Апплет содержит панель меню с двумя пунктами: **Шрифт** и **Цвет**. В области апплета есть метка с текстом (вначале красным цветом отображается текст **Красный текст на сером фоне**), две опции (**Курсивный шрифт** и **Жирный шрифт**), кнопка **Сброс**, а также группа из трех переключателей (**Красный**, **Черный** и **Синий**).

В пункте меню **Шрифт** представлено три команды (**Курсив**, **Жирный** и **Сброс**), причем две из них являются опционными (**Курсив** и **Жирный**). Перед командой **Сброс** есть горизонтальная черта (разделитель). Апплет с открытым пунктом меню **Шрифт** представлен на рис. 17.23.

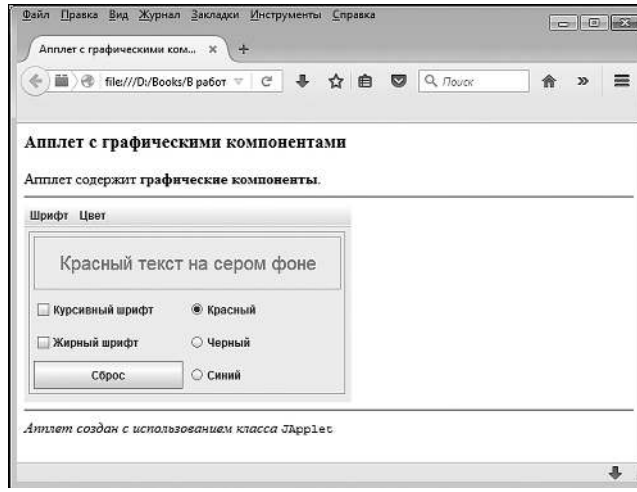


Рис. 17.22. Начальный вид окна с апплетом

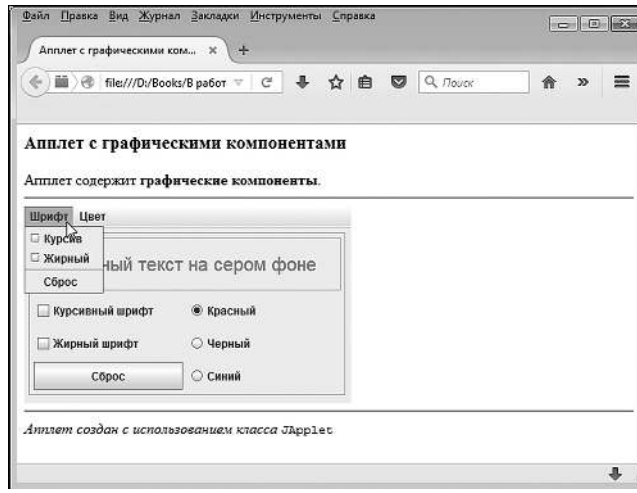


Рис. 17.23. Содержимое пункта меню **Шрифт** апплета

Пункт меню **Цвет** содержит группу из трех команд-переключателей (**Красный**, **Черный** и **Синий**), как показано на рис. 17.24.

Выбирая команды меню и/или задавая настройки элементов управления в области апплета, меняем вид и содержание текста в области метки. Так, на рис. 17.25 показано, каков будет результат, если в области апплета установлен переключатель **Черный**, а также установлен флажок опции **Жирный шрифт**.

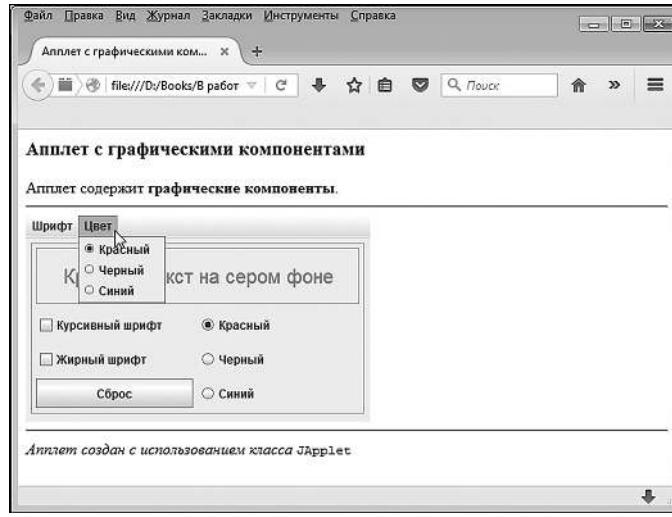


Рис. 17.24. Содержимое пункта меню **Цвет** апплета

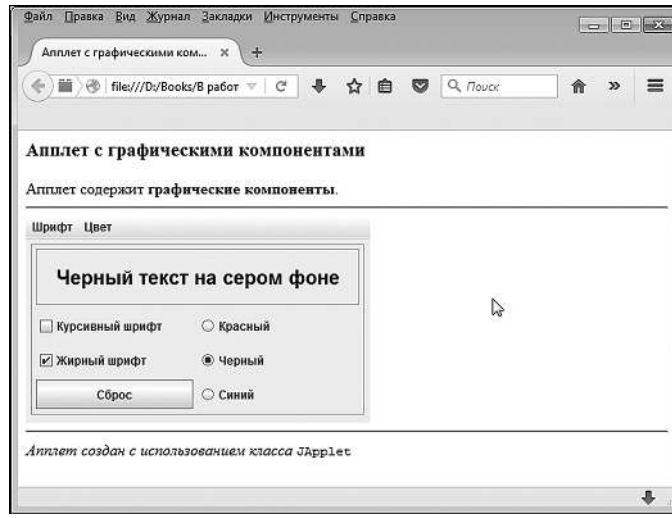


Рис. 17.25. При изменении состояния элементов управления меняется текст и шрифт метки: в области метки отображается черный текст жирного стиля

Видим, что название текста изменилось (теперь метка содержит текст **Черный текст на сером фоне**), цвет текста черный, а стиль — жирный.

На рис. 17.26 установлен переключатель **Синий**, а также установлены флажки опций **Курсивный шрифт** и **Жирный шрифт**.

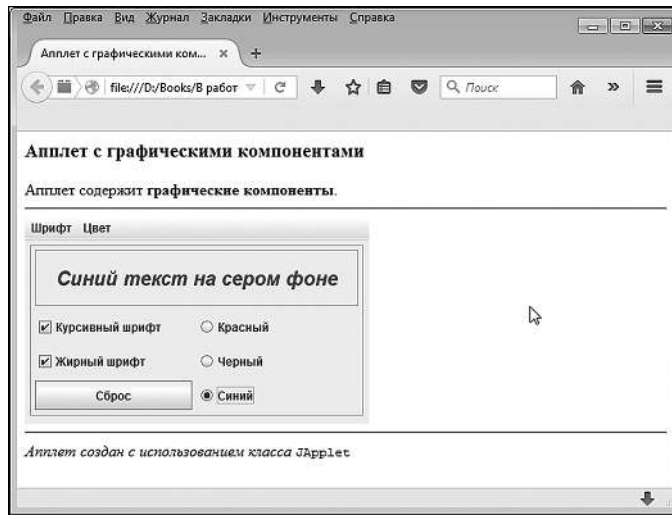


Рис. 17.26. При изменении состояния элементов управления меняется текст и шрифт метки: в области метки отображается синий текст жирного курсивного стиля

Как следствие в области метки отображается текст **Синий текст на сером фоне**, причем отображается он синим цветом, а стиль шрифта — жирный и курсивный. То есть все происходит строго в соответствии с настройками элементов управления. Более того, синхронно с изменением состояния элементов управления меняются настройки опционных команд и команд-переключателей в меню.

На рис. 17.27 показано состояние (настройки) опционных команд в пункте меню **Шрифт** при условии, что в области апплета установлены флажки для опций **Курсивный шрифт** и **Жирный шрифт**.

Настройки команд-переключателей в пункте меню **Цвет** (при условии, что в области апплета установлен переключатель **Синий**) показаны на рис. 17.28.

Верно и обратное — при изменении настроек команд меню автоматически меняются настройки элементов управления в области апплета. И, кроме этого, соответствующий образом меняется вид и содержание текстовой метки.

Наконец, если щелкнуть кнопку **Сброс** или выбрать одноименную команду в пункте меню **Шрифт**, то текстовая метка вернется в свое исходное состояние: красным цветом обычным (не жирным и не курсивным)

шрифтом будет отображаться текст **Красный текст на сером фоне**, как показано на рис. 17.29.

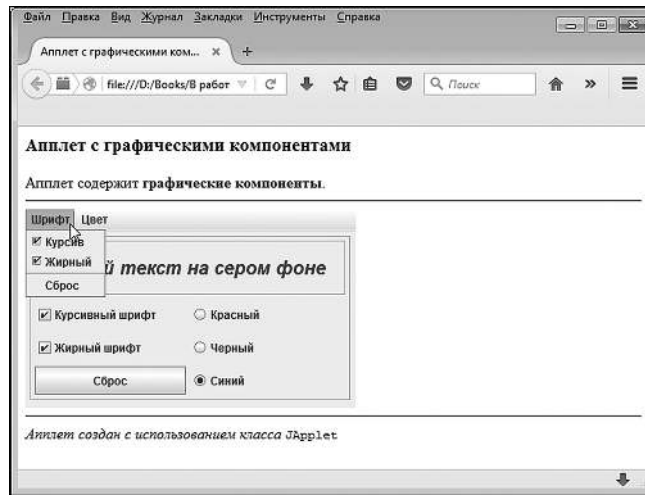


Рис. 17.27. Состояние опционных команд для пункта меню **Шрифт**

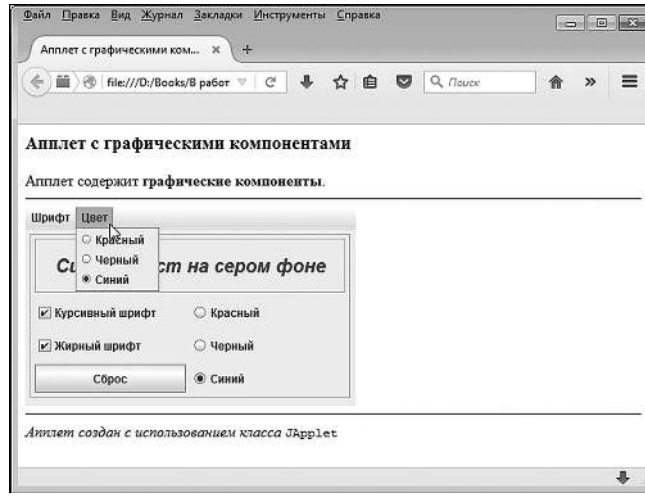


Рис. 17.28. Состояние команд-переключателей для пункта меню **Цвет**

При этом соответствующим образом будут выполнены настройки элементов управления в области апплета и команд меню.

Теперь рассмотрим программный код, с помощью которого реализован описанный выше апплет.

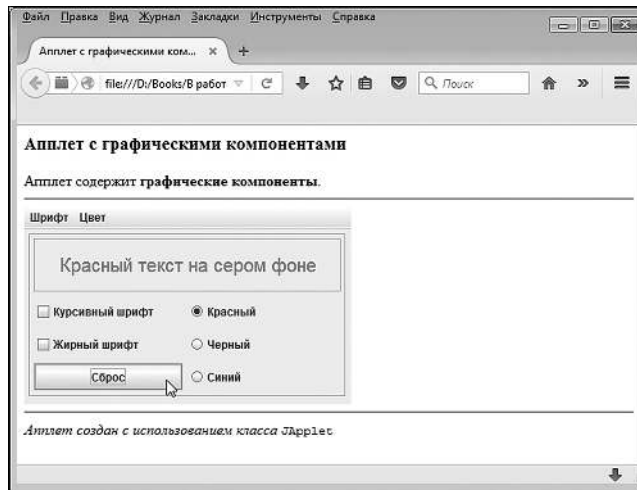


Рис. 17.29. Результат щелчка на кнопке **Сброс**

Сразу отметим, что код ориентирован на то, чтобы проиллюстрировать некоторые механизмы и подходы. Так что оптимальность кода в данном случае не была самоцелью. Код представлен в листинге 17.8.

Листинг 17.8. Программный код проекта MyFourthApplet

```
// Импорт классов:
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
// Класс апплета:
public class MyGUIApplet extends JApplet{
  // Панель меню:
  private JMenuBar mb;
  // Пункты меню:
  private JMenu font,color;
  // Опционные команды меню:
  private JCheckBoxMenuItem mItalic,mBold;
  // Команды меню — переключатели:
  private JRadioButtonMenuItem mRed,mBlack,mBlue;
  // Команда меню:
```

```
private JMenuItem mReset;
// Метка:
private JLabel text;
// Опции:
private JCheckBox italic,bold;
// Кнопка:
private JButton reset;
// Переключатели:
private JRadioButton red,black,blue;
// Метод для определения текста и шрифта для метки:
private void setAll(){
  // Цвет:
  Color clr;
  // Текст для метки:
  String txt=" текст на сером фоне";
  // Стиль (начальное значение):
  int style=Font.PLAIN;
  // Определение стиля:
  if(italic.isSelected()){
    style|=Font.ITALIC; // Курсив
  }
  if(bold.isSelected()){
    style|=Font.BOLD; // Жирный шрифт
  }
  // Уточнение текста и определение цвета:
  if(red.isSelected()){
    txt="Красный"+txt;
    clr=Color.RED;
  }
  else{
    if(black.isSelected()){
      txt="Черный"+txt;
```

```
        clr=Color.BLACK;
      }
      else{
        txt="Синий"+txt;
        clr=Color.BLUE;
      }
    }
    // Текст для метки:
    text.setText(txt);
    // Шрифт для метки:
    text.setFont(new Font("Arial",style,20));
    // Цвет текста метки:
    text.setForeground(clr);
  }
  // Метод инициализации апплета:
  public void init(){
    // Переменные:
    int w,h;
    // Ширина и высота апплета:
    w=getWidth();
    h=getHeight();
    // Отключение менеджера компоновки:
    setLayout(null);
    // Создание панели:
    JPanel pnl=new JPanel();
    // Положение и размеры панели:
    pnl.setBounds(5,5,w-10,h-35);
    // Рамка для панели:
    pnl.setBorder(BorderFactory.createEtchedBorder());
    // Отключение менеджера компоновки панели:
    pnl.setLayout(null);
    // Создание метки:
```

```
    text=new JLabel();
    // Режим выравнивания текста по центру:
    text.setHorizontalAlignment(JLabel.CENTER);
    // Положение и размеры метки:
    text.setBounds(5,5,pnl.getWidth()-10,pnl.getHeight()/3);
    // Рамка вокруг метки:
    text.setBorder(BorderFactory.createEtchedBorder());
    // Добавление метки на панель:
    pnl.add(text);
    // Создание опций:
    italic=new JCheckBox("Курсивный шрифт");
    bold=new JCheckBox("Жирный шрифт");
    // Создание кнопки:
    reset=new JButton("Сброс");
    // Положение и размеры опций и кнопки:
    italic.setBounds(text.getX(),text.getY()+text.getHeight()+5,text.getWidth()/2-5,30);
    bold.setBounds(italic.getX(),italic.getY()+italic.getHeight()+5,italic.getWidth(),italic.
getHeight());
    reset.setBounds(bold.getX(),bold.getY()+bold.getHeight()+5,bold.getWidth(),bold.
getHeight());
    // Добавление опций и кнопки на панель:
    pnl.add(italic);
    pnl.add(bold);
    pnl.add(reset);
    // Создание группы переключателей:
    ButtonGroup bg=new ButtonGroup();
    // Создание переключателей:
    red=new JRadioButton("Красный",true);
    black=new JRadioButton("Черный",false);
    blue=new JRadioButton("Синий",false);
    // Добавление переключателей в группу:
    bg.add(red);
```

```
    bg.add(black);
    bg.add(blue);
    // Положение и размеры переключателей:
    red.setBounds(italic.getX()+italic.getWidth()+5,italic.getY(),italic.getWidth(),italic.
getHeight());
    black.setBounds(red.getX(),bold.getY(),red.getWidth(),red.getHeight());
    blue.setBounds(black.getX(),reset.getY(),black.getWidth(),black.getHeight());
    // Добавление переключателей на панель:
    pnl.add(red);
    pnl.add(black);
    pnl.add(blue);
    // Создание панели меню:
    mb=new JMenuBar();
    // Создание пункта меню:
    font=new JMenu("Шрифт");
    // Создание опционных команд меню:
    mItalic=new JCheckBoxMenuItem("Курсив",false);
    mBold=new JCheckBoxMenuItem("Жирный",false);
    // Создание команды меню:
    mReset=new JMenuItem("Сброс");
    // Добавление опционных команд в пункт меню:
    font.add(mItalic);
    font.add(mBold);
    // Добавление разделителя:
    font.addSeparator();
    // Добавление команды в пункт меню:
    font.add(mReset);
    // Создание пункта меню:
    color=new JMenu("Цвет");
    // Создание группы переключателей:
    ButtonGroup mBG=new ButtonGroup();
    // Создание команд-переключателей:
```

```
mRed=new JRadioButtonMenuItem("Красный",true);
mBlack=new JRadioButtonMenuItem("Черный",false);
mBlue=new JRadioButtonMenuItem("Синий",false);
// Добавление команд-переключателей в группу:
mBG.add(mRed);
mBG.add(mBlack);
mBG.add(mBlue);
// Добавление команд-переключателей в пункт меню:
color.add(mRed);
color.add(mBlack);
color.add(mBlue);
// Добавление пунктов меню на панель меню:
mb.add(font);
mb.add(color);
// Добавление панели меню в апплет:
setJMenuBar(mb);
// Применение текста и шрифта к метке:
setAll();
// Добавление панели в апплет:
add(pnl);
// Регистрация обработчиков
// для событий класса ActionEvent:
mItalic.addActionListener(e->italic.setSelected(mItalic.isSelected()));
mBold.addActionListener(e->bold.setSelected(mBold.isSelected()));
italic.addActionListener(e->mItalic.setSelected(italic.isSelected()));
bold.addActionListener(e->mBold.setSelected(bold.isSelected()));
mRed.addActionListener(e->red.setSelected(mRed.isSelected()));
mBlack.addActionListener(e->black.setSelected(mBlack.isSelected()));
mBlue.addActionListener(e->blue.setSelected(mBlue.isSelected()));
red.addActionListener(e->mRed.setSelected(red.isSelected()));
black.addActionListener(e->mBlack.setSelected(black.isSelected()));
blue.addActionListener(e->mBlue.setSelected(blue.isSelected()));
// Регистрация обработчиков
```

```
    // для событий класса ItemEvent:
    italic.addItemListener(e->setAll());
    bold.addItemListener(italic.getItemListeners()[0]);
    red.addItemListener(italic.getItemListeners()[0]);
    black.addItemListener(italic.getItemListeners()[0]);
    blue.addItemListener(italic.getItemListeners()[0]);
    // Обработчик для кнопки:
    reset.addActionListener(e->{
      italic.setSelected(true);
      // Программный "щелчок" на опции:
      italic.doClick();
      bold.setSelected(true);
      // Программный "щелчок" на опции:
      bold.doClick();
      // Программный "щелчок" на переключателе:
      red.doClick();
    });
    // Обработчик для команды меню:
    mReset.addActionListener(e->reset.doClick());
  }
}
```

Основные блоки кода (особенно те, которыми создаются компоненты интерфейса) в принципе должны быть понятны читателю, так что мы остановимся лишь на наиболее значимых моментах.

Если не принимать во внимание import-инструкции, то код представляет собой описание класса апплета MyGUIApplet, наследующего класс JApplet.

В классе использовано довольно большое количество закрытых полей, большинство из которых являются ссылками на объекты компонентов графического интерфейса. В частности, мы используем следующие компоненты интерфейса.

- Панель меню, которая реализуется через поле mb (объектная ссылка класса JMenuBar).

- Пункты меню, реализуемые через поля font (пункт меню **Шрифт**) и color (пункт меню **Цвет**), являющиеся объектными ссылками класса JMenu.
- Опционные команды меню mItalic (команда-опция **Курсив**) и **mBold** (команда-опция **Жирный**) — эти компоненты реализуются с помощью объектов класса JCheckBoxMenuItem.
- С помощью объектных ссылок mRed, mBlack и mBlue класса JRadioButtonMenuItem реализуются команды-переключатели **Красный**, **Черный** и **Синий** соответственно.
- Команда **Сброс** представлена полем mReset, представляющим собой объектную ссылку класса JMenuItem.
- Метка с текстом реализована черед поле text (объектная ссылка класса JLabel).
- Опции в области апплета отождествляем с полями italic (опция **Курсив**) и bold (опция **Жирный**), являющимися объектными ссылками класса JCheckBox.
- Кнопка **Сброс**, которая размещается в области апплета, реализована через поле reset (объектная ссылка класса JButton).
- Группа переключателей (объекты класса JRadioButton) представлена полями red (**Красный**), black (**Черный**) и blue (**Синий**).

Важную роль играет метод setAll(), при вызове которого на основе состояния элементов управления в области апплета «вычисляются» характеристики для текста и шрифта метки. В частности, при вызове метода на основе настроек опций italic и bold определяется стиль шрифта, его цвет, а также собственно сам текст. После этого все определенные таким образом параметры применяются к метке text.

НА ЗАМЕТКУ

Таким образом, как бы не изменились настройки элементов управления в области апплета, для их практической реализации достаточно вызвать метод setAll(). Единственное, что не делает данный метод — так это не обеспечивает синхронное изменение настроек элементов управления в области апплета и команд меню.

Все прочее происходит в методе init() инициализации апплета. В теле метода объявляются две целочисленные переменные w и h. Командами

w=getWidth() и h=getHeight() значениями переменным присваиваются соответственно ширина и высота апплета.

> **НА ЗАМЕТКУ**
>
> Значение для ширины и высоты апплета задаются в HTML-документе. В данном случае эти значения считываются и используются в программном коде в методе инициализации апплета.

В область апплета помещается панель, которая создается командой JPanel pnl=new JPanel(). Вокруг панели рисуется рамка (команда pnl.setBorder(BorderFactory.createEtchedBorder())). Размеры панели устанавливаются на основании значений для ширины и высоты апплета. Все элементы управления добавляются на панель. Так, метка создается командой text=new JLabel(). Командой text.setHorizontalAlignment(JLabel.CENTER) для метки устанавливается режим выравнивания текста по центру (хотя самого теста в метке пока еще нет). Положение и размеры метки определяются командой text.setBounds(5,5,pnl.getWidth()-10,pnl.getHeight()/3). В соответствии с этой командой высота метки составляет треть от высоты панели, а ширина метки на 10 пикселов меньше ширины панели. Вокруг метки также отображается рамка.

Опции создаются командами italic=new JCheckBox("Курсивный шрифт") и bold=new JCheckBox("Жирный шрифт"). Для создания кнопки используем команду reset=new JButton("Сброс"). При вычислении положения и размеров этих компонентов использованы методы getX() и getY(), позволяющие вычислить координаты компонента, из объекта которого вызываются методы. Например, в результате выполнения команды italic.setBounds(text.getX(),text.getY()+text.getHeight()+5,text.getWidth()/2-5,30) для опции italic горизонтальная координата такая же, как горизонтальная координата метки text (инструкция text.getX()). Вертикальная координата для опции italic вычисляется выражением text.getY()+text.getHeight()+5 (значение выражения — вертикальная координата метки, плюс ее высота, и плюс еще 5 пикселов). Высота опции составляет 30 пикселов, а ширина опции на 5 пикселов меньше половины ширины метки.

> **НА ЗАМЕТКУ**
>
> Аналогичным образом вычисляются координаты и размеры для прочих графических компонентов в области апплета. Желающие могут самостоятельно провести анализ принципов, в соответствии с которыми компоненты размещаются на панели.

При создании переключателей вторым аргументом конструктору класса JRadioButton передается логическое значение, определяющее состояние переключателя (true означает, что переключатель установлен, а false — что не установлен). Для объединения переключателей в одну группу создается объект bg класса ButtonGroup, и объект каждого из переключателей добавляется в группу (добавляется в объект bg).

Объект для панели меню создается командой mb=new JMenuBar(). На панель меню добавляем пункты меню, которые реализуются через объекты font и color. Объект font создается командой font=new JMenu("Шрифт"). В пункт меню добавляются команды меню. Для добавления в пункт меню font мы создаем объекты для двух опционных команд и одной обычной. Объекты для опционных команд меню создаются инструкциями mItalic=newJCheckBoxMenuItem("Курсив",false) и mBold=new JCheckBoxMenuItem("Жирный",false). Второй аргумент конструктора определяет, установлен ли флажок опции. Объект для обычной команды меню создается командой mReset=new JMenuItem("Сброс"). После создания объектов для пунктов меню инструкциями font.add(mItalic), font.add(mBold) и font.add(mReset) выполняется «заполнение» пункта меню font. При этом после добавления двух опционных команд с помощью инструкции font.addSeparator() добавляем разделитель (горизонтальная линия, разделяющая команды меню).

Объект color создается командой color=new JMenu("Цвет"). Поскольку в этот пункт меню добавляются команды-переключатели, то для объединения этих переключателей в одну группу создаем объект mBG класса ButtonGroup. Объекты mRed, mBlack и mBlue класса JRadioButtonMenuItem (переключатели) добавляются в группу (реализованную через объект mBG) и в пункт меню color.

Добавление пунктов font и color на панель меню выполняется командами mb.add(font) и mb.add(color). Панель меню добавляется в апплет командой setJMenuBar(mb). Вызовом метода setAll() для метки применяется текст и шрифт.

Отдельный блок кода содержит регистрацию обработчиков для графических компонентов. Здесь имеет смысл сначала представить общую схему обработки событий, использованную в программе. Мы решаем две задачи: во-первых, необходимо синхронизировать элементы управления в области апплета и команды в меню, а, во-вторых, необходимо предусмотреть механизм изменения текста и шрифта для метки при изменении настроек управляющих элементов или команд меню. Эти две задачи решаются отдельно. Сначала регистрируются обработчики для событий класса ActionEvent. С помощью этих обработчиков решается задача

по синхронизации элементов управления в области апплета и команд в меню. Например, командой mItalic.addActionListener(e->italic.setSelected(mItalic.isSelected())) для опционной команды выбора курсивного стиля в пункте меню **Шрифт** регистрируется обработчик щелчка на команде такой, что для опции в области апплета, реализованной через поле italic, задается такая же настройка (определяющая, установлен или нет флажок опции), как для команды меню. В свою очередь, командой italic.addActionListener(e->mItalic.setSelected(italic.isSelected())) для опции применения курсивного стиля для шрифта регистрируется обработчик, которым в соответствующее состояние переводится и опционная команда в меню **Шрифт**. По такому же принципу организована обработка событий класса ActionEvent для прочих графических компонентов.

Для опций и переключателей в области апплета также регистрируются обработчики для событий класса ItemEvent. В частности, для опции italic обработчик регистрируется командой italic.addItemListener(e->setAll()). В соответствии с этим обработчиком при изменении состояния опции вызывается метод setAll(), что приводит к обновлению содержимого метки. Поскольку при изменении состояния прочих компонентов необходимо выполнить те же действия, то для всех этих компонентов регистрируется тот же обработчик, что и для опции italic. Для получения доступа к обработчику событий класса ItemEvent опции italic используем инструкцию italic.getItemListeners()[0].

> **НА ЗАМЕТКУ**
>
> Методом getItemListeners() возвращается массив обработчиков событий класса ItemEvent, зарегистрированных в компоненте, из объекта которого вызывается метод. В данном случае обработчик для событий класса ItemEvent один, и, следовательно, у него нулевой индекс.

Общий эффект получается такой: если изменяется состояние элемента управления в области апплета, то благодаря обработчику событий класса ItemEvent обновляется содержимое метки. Если же пользователь щелкает команду в меню, то вызывается обработчик события класса ActionEvent, что приводит к изменению состояния соответствующего элемента в области управления, а это, в свою очередь, благодаря обработчику события класса ItemEvent приводит к обновлению содержимого метки.

Для кнопки **Сброс** регистрируется обработчик события класса ActionEvent. При описании обработчика использован метод doClick(), которым

имитируется щелчок на компоненте, из которого вызывается метод. Например, если сначала командой italic.setSelected(true) установить флажок опции italic, то после выполнения команды italic.doClick() флажок будет отменен, а общий эффект такой, как если бы на опции выполнен щелчок. Это приводит к вызову обработчика класса ActionEvent и обработчика события класса ItemEvent.

Аналогично опции italic поступаем и с опцией bold. А вот для переключателей использована всего одна команда red.doClick(), которой программными методами выполняется «щелчок» на переключателе **Красный**.

> **НА ЗАМЕТКУ**
>
> Каждый раз при изменении состояния элементов управления, в том числе и в результате программных действий, в дело вступают обработчики, которыми обновляется содержимое метки.

Для команды **Сброс** в пункте меню **Шрифт** обработчик регистрируется командой mReset.addActionListener(e->reset.doClick()). В соответствии с кодом обработчика при щелчке на команде выполняется программный «щелчок» на кнопке **Сброс** в области апплета.

Резюме

Все время думать одну и ту же мысль нельзя.
Это очень вредно.

Из м/ф «38 попугаев»

- Апплет представляет собой программу, которая выполняется под управлением браузера. Апплет добавляется в веб-документ с помощью специального дескриптора <applet>. Также апплету могут передаваться параметры. Для передачи параметров апплету используется дескриптор <param>.
- Один из способов создания апплета подразумевает создание класса апплета путем наследования класса JApplet из пакета javax.swing.
- Для апплетов не описывается метод main(), а также не создается конструктор. Вместо этого в классе апплета описывается метод init(), вызываемый при создании апплета. Также могут использоваться методы start() (вызывается при обращении к странице с апплетом), stop()

(вызывается при уходе со страницы) и destroy() (вызывается перед завершением работы апплета).

- Для использования апплета необходимо откомпилировать файл с классом апплета. В веб-документе указывается ссылка на файл, получающийся в результате компиляции файла с классом апплета.
- Апплет может содержать практически те же компоненты графического интерфейса, что и обычное окно. Соответственно, для апплета выполняется обработка событий для графических компонентов, размещенных в области апплета.

Глава 18
ФАЙЛЫ И АРГУМЕНТЫ КОМАНДНОЙ СТРОКИ

— Так, значит, до Камелота мы не доберемся?
— Не доберемся. Потому что бездорожье полное.

Из к/ф «31 июня»

В этой главе мы рассмотрим две темы, которые формально между собой не связаны, но в некотором смысле все же имеют общие «точки соприкосновения». А именно, речь пойдет о передаче аргументов (или параметров) программе при запуске (аргументы/параметры командной строки) и работе с файлами. То есть в обоих случаях подразумевается, что программа получает данные из внешнего источника.

Аргументы командной строки

Куда? Эй, куда же вы все-то разбежались? Кто-нибудь, держите меня!

Из к/ф «Айболит-66»

При описании главного метода main() для этого метода описывается аргумент, который является текстовым массивом (массивом, состоящим из объектных переменных класса String). Вместе с тем ранее мы ни разу данным массив не использовали. Теперь пришло время наверстать упущенное.

Итак, аргументом методу main() передается текстовый массив, через значения которого реализуются параметры, которые передаются программе при запуске на выполнение. Дело все в том, что запуская программу на выполнение, ей можно передать параметры. Если бы мы запускали откомпилированную программу из командной строки, то после имени файла с откомпилированным главным классом программы через пробел указать передать некоторые параметры. Эти параметры считываются и их можно использовать в программе. Другими словами, в программном

коде метода main() можно обращаться к элементам массива, указанного аргументом метода. Сам массив формируется при запуске программы на выполнение на основе параметров, которые передаются программе. Как иллюстрацию такого подхода рассмотрим пример, в котором вычисляется итоговая сумма по вкладу. При этом все необходимые параметры передаются в метод main() через массив-аргумент.

НА ЗАМЕТКУ

Напомним, что если речь идет о начальной денежной сумме m, размещенной под процентную годовую ставку r на период времени t (в годах), то итоговая сумма по вкладу вычисляется как $m(1 + r/100)^t$. Собственно эта формула далее и используется при вычислении итоговой суммы по вкладу.

В частности, мы будем исходить из того, что в программу передаются следующие параметры:

- имя вкладчика (текстовое значение);
- начальная сумма вклада (значение типа double);
- процентная ставка (значение типа double);
- период времени (в годах), на который размещается вклад (значение типа int).

Программой выводятся в окно с сообщением все эти параметры, а также итоговая сумма по вкладу. Соответствующий несложный программный код представлен в листинге 18.1.

Листинг 18.1. Программный код проекта GetMoneyApplication

```
// Статический импорт:
import static javax.swing.JOptionPane.*;
// Главный класс:
class GetMoneyDemo{
  // Главный метод:
  public static void main(String[] args){
    // Имя вкладчика:
    String name;
    // Начальная сумма вклада:
```

```
double m;
// Процентная ставка:
double r;
// Время размещения вклада:
int t;
// Итоговая сумма по вкладу:
double val;
// Контролируемый код:
try{
  // Определение имени вкладчика:
  name=args[0]+" "+args[1];
  // Определение начальной суммы вклада:
  m=Double.parseDouble(args[2]);
  // Определение процентной ставки:
  r=Double.parseDouble(args[3]);
  // Определение интервала времени:
  t=Integer.parseInt(args[4]);
  // Вычисление итоговой суммы.
  // Начальное значение:
  val=m;
  // "Начисление процентов":
  for(int k=1;k<=t;k++){
    val*=(1+r/100);
  }
  // "Уточнение" результата:
  val=((int)(100*val))/100.0;
  // Формирование текста для отображения сообщения
  // в диалоговом окне:
  String txt="Имя:      "+name+"\n";
  txt+="Вклад:  "+m+"\n";
  txt+="Ставка:  "+r+"\n";
  txt+="Время:   "+t+"\n";
  txt+="Итог:      "+val;
```

```
      // Отображение окна с сообщением:
      showMessageDialog(null,
        txt, // Текст сообщения
        // Название окна:
        "Итоговая сумма по депозиту",
        // Тип окна:
        INFORMATION_MESSAGE
      );
    } // Обработка исключения:
    catch(Exception e){
      // Отображение сообщения:
      showMessageDialog(null,
        // Текст сообщения:
        "Неверно указаны параметры!\n"+e,
        // Название окна:
        "Ошибка",
        // Тип окна:
        ERROR_MESSAGE
      );
    }
  }
}
```

В главном методе программы объявляется несколько переменных, в которые предполагается заносить исходные расчетные данные:

- текстовая переменная `name` предназначена для записи имени вкладчика;
- в `double`-переменные `m` и `r` предполагается записывать соответственно значения для начальной суммы и процентной ставки;
- в целочисленную переменную `t` будет заноситься время, на которое размещается вклад;
- в переменную `val` типа `double` записывается значение итоговой суммы по вкладу.

Значения всех этих переменных определяются на основе параметров, которые передаются (как предполагается) программе. Поскольку количество и значение переданных программе параметров может не соответствовать количеству и типу объявленных переменных, все последующие вычисления размещаются в try-блок. В этом блоке командой name=args[0]+" "+args[1] формируется текстовое значение для переменной name. Здесь мы предполагаем, что имя вкладчика состоит из двух «слов», представляющих собой фактическое имя и фамилию вкладчика, которые передаются первым и вторым параметрами программе. Мы их объединяем в одно текстовое значение, использовав в качестве разделителя пробел. Значения переданных программе параметров получаем через ссылку на элементы массива args, объявленного аргументом метода main().

Далее командами m=Double.parseDouble(args[2]) и r=Double.parseDouble(args[3]) два следующих параметра используются для определения начальной суммы вклада и процентной ставки. Все параметры при передаче в программу интерпретируются как текстовые. Но мы предполагаем, что второй и третий параметры по своей сути являются действительными числами. Тогда соответствующие элементы массива args должны быть текстовым представлением действительных чисел. С помощью статического метода parseDouble() эти значения преобразуются в числа. Аналогичная процедура выполняется командой t=Integer.parseInt(args[4]), но только здесь речь идет о преобразовании в целочисленное значение.

НА ЗАМЕТКУ

Если любая из указанных операций приводит к ошибке (например, при неправильно указанном параметре), то такая ошибка будет перехвачена и обработана в catch-блоке. Типом перехватываемой ошибки в catch-блоке указан класс Exception. Поэтому в данном блоке перехватываются и обрабатываются исключения всех классов, являющихся подклассами класса Exception.

При возникновении ошибки появляется сообщение о неправильно переданных программе параметрах. Объект исключения также использован при обработке: принята во внимание та особенность, что для объектов исключений переопределен метод toString() и поэтому их можно «выводить на экран» и объединять с текстовыми значениями.

После определения значений переменных на основе параметров программы, вычисляется значение итоговой суммы по вкладу. Результат записывается в переменную val. Причем уже после окончания

собственно вычислений с помощью команды val=((int)(100*val))/100.0 в полученном значении оставляется только две цифры после десятичной точки.

ДЕТАЛИ

Команда val=((int)(100*val))/100.0 выполняется следующим образом. Сначала текущее значение переменной val умножается на 100. Полученное значение благодаря инструкции (int) приводится к целочисленному типу, что на самом деле означает отбрасывание дробной части в числе. Затем полученное значение делится на 100.0. При делении мы используем литерал с десятичной точкой, чтобы деление выполнялось на множестве действительных чисел (в противном случае выполняется целочисленное деление).

Наконец, формируется текст для отображения в окне сообщения. Текст поэтапно записывается в текстовую переменную txt. После этого появляется окно с информацией о вкладчике и вкладе.

Теперь несколько слов о собственно выполнении программы. Если мы просто запустим программу на выполнение (например, в среде NetBeans), то на экране появится окно, представленное на рис. 18.1.

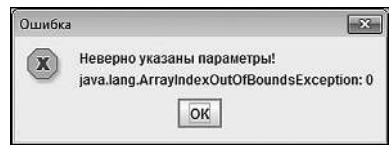


Рис. 18.1. При запуске программы без передачи параметров появляется сообщение об ошибке

Причина появления окна в том, что программе не передаются параметры, и при выполнении кода происходит ошибка (появившееся окно является следствием выполнения кода в catch-блоке). Чтобы программе при запуске передавались параметры, необходимо предпринять некоторые дополнительные усилия. Для этого в окне среды разработки NetBeans необходимо выбрать в меню **Run** подменю **Set Project Configuration**, в котором, в свою очередь, выбирается команда **Customize**, как показано на рис. 18.2.

В результате откроется окно **Project Properties**, представленное на рис. 18.3.

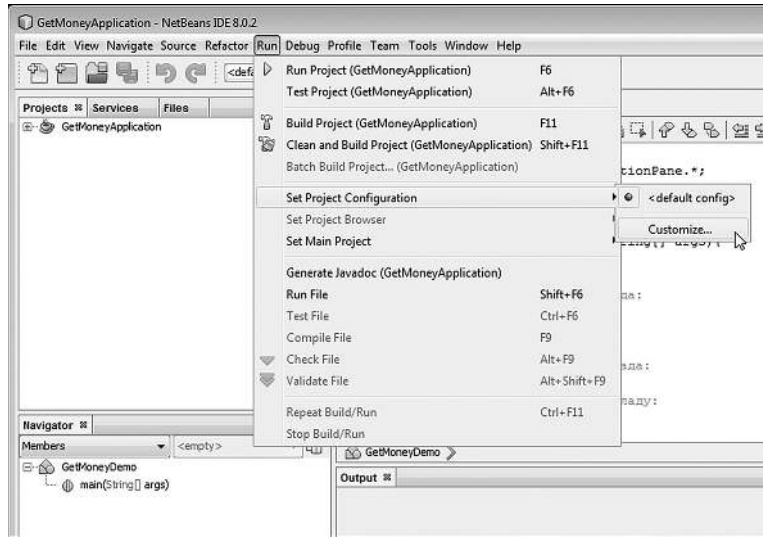


Рис. 18.2. Выбор команды **Customize** в подменю **Set Project Configuration** меню **Run**

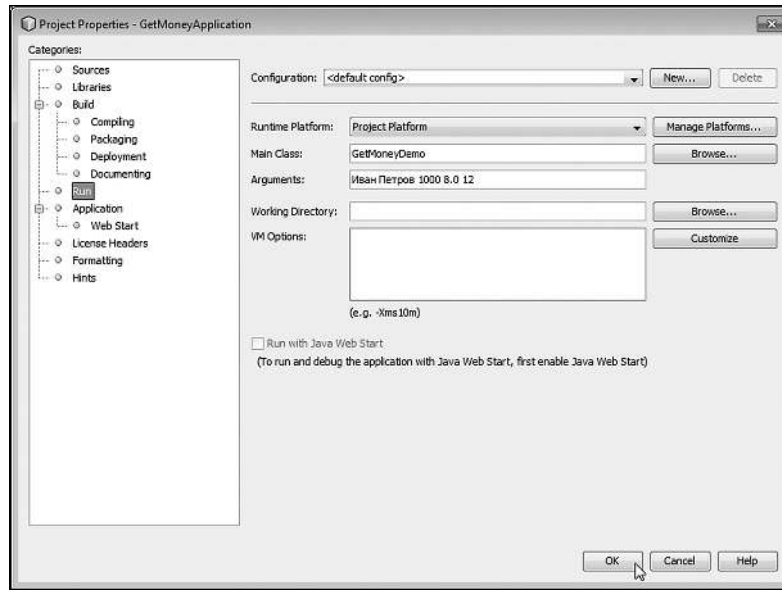


Рис. 18.3. Выполнение настроек в окне **Project Properties**

В левой части окна в разделе **Categories** выбирается пункт **Run** (он выбран по умолчанию), а в правой части окна в поле **Arguments** через пробел вводятся параметры, которые передаются программе при

запуске. В данном случае мы вводим в поле следующую последовательность:

```
Иван Петров 1000 8.0 12
```

Ввод данных подтверждается щелчком на кнопке **ОК**. После этого запускаем программу на выполнение. Появляется диалоговое окно, представленное на рис. 18.4.

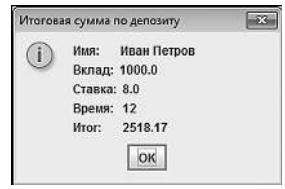


Рис. 18.4. Окно, которое появляется при запуске программы с передачей ей параметров

Как видим, параметры переданы в программу и обработаны. Для запуска программы с другими параметрами следует снова открыть окно **Project Properties** и указать новые значения для параметров в поле **Arguments**.

НА ЗАМЕТКУ

Стоит заметить, что если бы программа запускалась из командной строки, то с учетом фактических параметров, которые передаются программе, инструкция в командной строке выглядела бы следующим образом:

```
java.exe GetMoneyDemo Иван Петров 1000 8.0 12
```

Правда, запуск программы из командной строки сегодня в общем-то, является экзотикой.

Работа с файлами

Вы сюда приехали, чтобы записывать сказки, понимаете ли, а мы здесь работаем, чтобы сказку сделать былью, понимаете ли.

Из к/ф «Кавказская пленница»

Нередко приходится иметь дело с файлами, из которых программа получает данные, записывает в них данные или просто обрабатывает

информацию, связанную с файлами. Далее мы рассмотрим некоторые базовые операции, которые важны с прикладной точки зрения. Среди этих операций, разумеется, операции, связанные с файловым вводом и выводом.

Получение информации о файле

Если необходимо просто получить некоторую информацию о файле или директории, можем использовать класс File. Класс доступен после импорта пакета java.io.

Объект класса File создается несколькими способами. Например, при создании объекта класса File аргументом конструктору можно передать полный путь к файлу. Также отдельно первым аргументом может быть указан путь к родительской директории (той, в которой содержится файл или поддиректория), а вторым аргументом — имя файла. Вместо первого текстового аргумента, определяющего путь к родительской директории, может быть указан объект класса File, соответствующий этой директории.

Здесь важно отметить, что в принципе файл или директория, указанные при создании объекта класса File, могут и не существовать.

В принципе, объект класса File представляет собой некую «сущность», связанною с определенным местом в файловой иерархии, даже если такого места в иерархии нет. То есть на самом деле это не объект для файла, а объект для пути к файлу.

НА ЗАМЕТКУ

Существуют разные способы определения пути к файлу. Абсолютный, или полный, путь подразумевает, что в явном виде указан путь к файлу, начиная от корневой директории. Также используется такое понятие, как канонический путь. В данном случае имеется в виду то, что путь представлен в «классическом» виде, без использования некоторых специальных символов (вроде двойной точки и слеша ../, обозначающей директорию на один уровень выше от текущей, или одинарной точки и слеша ./, обозначающей текущую директорию).

Некоторые наиболее полезные и интересные методы класса File перечислены и кратко описаны в табл. 18.1.

Табл. 18.1. Некоторые методы класса File

Метод	Описание
canExecute()	Результатом метод возвращает логическое значение, определяющее, является ли файл исполняемым (может ли он быть запущен на выполнение)
canRead()	Результатом метод возвращает логическое значение, определяющее, доступен ли файл для чтения
canWrite()	Результатом метод возвращает логическое значение, определяющее, доступен ли файл для записи
createNewFile()	Методом создается новый пустой файл с названием, определяемым на основе объекта, из которого вызывается метод. Файл создается только при условии, что файла с данным именем еще не существует
delete()	Метод используется для удаления файла, соответствующего объекту, из которого вызывается метод
deleteOnExit()	Методом предназначен для удаления файла (определяемого объектом, из которого вызывается метод) перед завершением работы виртуальной машины
exists()	Метод позволяет проверить, существует ли файл
getAbsoluteFile()	Методом возвращается объект класса File, соответствующий абсолютному пути для объекта, из которого вызывается метод
getAbsolutePath()	Методом возвращается текст, соответствующий абсолютному пути для объекта, из которого вызывается метод
getCanonicalFile()	Методом возвращается объект класса File, соответствующий каноническому пути для объекта, из которого вызывается метод
getCanonicalPath()	Методом возвращается текст, соответствующий каноническому пути для объекта, из которого вызывается метод
getName()	Методом возвращается текстовое значение с названием файла
getParent()	Методом возвращается название родительской директории
getParentFile()	Методом возвращается объект класса File, соответствующий родительской директории
getPath()	Методом возвращается текстовое значение с указанием полного пути к файлу
isAbsolute()	Методом выполняется проверка, содержит ли объект, из которого вызывается метод, абсолютный путь к файлу
isDirectory()	Методом выполняется проверка, соответствует ли объект, из которого вызывается метод, директории
isFile()	Методом выполняется проверка, соответствует ли объект, из которого вызывается метод, файлу
isHidden()	Метод позволяет проверить, является ли файл скрытым
lastModified()	Метод возвращает время (в миллисекундах) последнего изменения файла
list()	Методом возвращается массив с названиями файлов и директорий, находящихся в директории, определяемой объектом, из которого вызывается метод

Метод	Описание
listFiles()	Методом возвращается массив объектов класса File, соответствующих файлам, находящимся в директории, определяемой объектом, из которого вызывается метод
listRoots()	Статический метод возвращает результатом массив объектов класса File, определяющих корневые директории системы
mkdir()	Методом создается директория, соответствующая объекту, из которого вызывается метод
mkdirs()	Методом создается директория, определяемая объектом, из которого вызывается метод. Если при этом адрес конечной директории содержит несуществующие внешние директории, они тоже создаются
renameTo()	Метод позволяет переименовать файл, определяемый объектом, из которого вызывается метод. Аргумент метода — объект класса File, определяющий конечный файл
setExecutable()	Методом устанавливается или отменяется (в зависимости от значения логического аргумента) разрешение на выполнение файла
setLastModified()	Методом определяется время последнего изменения файла
setReadable()	Методом устанавливается или отменяется (в зависимости от значения логического аргумента) разрешение на чтение файла
setReadOnly()	Метод задает для файла или директории режим, разрешающий только чтение
setWritable()	Методом устанавливается или отменяется (в зависимости от значения логического аргумента) разрешение на запись в файл
toString()	Метод для преобразования объекта класса File в текстовое значение

Небольшой пример, в котором используются объекты класса File, представлен в листинге 18.2.

Листинг 18.2. Программный код проекта GetFileInfoApplication

```
// Импорт классов:
import java.io.*;
// Главный класс:
class GetFileInfoDemo{
  // Главный метод:
  public static void main(String[] args){
    // Путь к файлу:
    String path="d:\\documents\\MyData.pdf";
    // Отображение пути к файлу:
    System.out.println(path);
    // Создание объекта класса File:
```

```
File F=new File(path);
// Название файла:
System.out.println("Имя файла: "+F.getName());
// Родительская директория:
System.out.println("Директория: "+F.getParent());
// Содержимое родительской директории:
File[] files=F.getParentFile().listFiles();
System.out.println("Содержимое папки:");
// Отображение названий файлов и директорий:
for(int k=0;k<files.length;k++){
  System.out.print("\t"+files[k].getName());
  if(files[k].isFile()){
    System.out.println(" — файл");
  }
  else{
    System.out.println(" — директория");
  }
}
System.out.println("Создание новой директории и перемещение файла");
// Создание объекта для новой директории:
File D=new File(F.getParentFile(),".\\MyFiles");
// Создание новой директории:
D.mkdir();
// Полный канонический путь к директории:
try{ // Контролируемый код
  System.out.println("Создана директория "+D.getCanonicalPath());
} // Обработка исключения:
catch(IOException e){
  System.out.println("Ошибка: "+e);
}
// Перемещение файла:
F.renameTo(new File(D,"MyCV.pdf"));
// Проверка существования файла:
System.out.print("Файл "+F.getAbsolutePath());
```

```
    if(F.exists()){ // Если файл существует
      System.out.println(" существует");
    }
    else{ // Если файл не существует
      System.out.println(" не существует");
    }
    // Путь к новой директории:
    System.out.println("В директории "+D.getPath()+" есть файлы:");
    // Массив файлов из новой директории:
    files=D.listFiles();
    // Названия файлов из новой директории:
    for(int k=0;k<files.length;k++){
      System.out.println(files[k].getName());
    }
  }
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 18.2)

```
d:\documents\MyData.pdf
Имя файла: MyData.pdf
Директория: d:\documents
Содержимое папки:
   covers — директория
   crystal.exe — файл
   docs — директория
   MyData.pdf — файл
   vasilev.jpg — файл
Создание новой директории и перемещение файла
Создана директория D:\Documents\MyFiles
Файл d:\documents\MyData.pdf не существует
В директории d:\documents\.\MyFiles есть файлы:
MyCV.pdf
```

Прокомментируем команды из программного кода и результат их выполнения. Итак, в программе объявляется текстовая переменная path, значение которой определяет путь к файлу. Значение переменной path отображается в консольном окне. Затем командой File F=new File(path) создается объект F класса File. Хотя формально объект F как бы имеет отношение к файлу, на самом деле это объект, который является «оберткой» для пути к файлу. О каком файле идет речь, можно узнать с помощью выражения F.getName(), возвращающего результатом название файла. Полный путь к директории, в которой находится файл, определяется выражением F.getParent().

НА ЗАМЕТКУ

Операции выполняются на уровне манипуляций с путем к файлу, указанному при создании объекта класса File. Из того, что возвращается путь к директории, в которой находится файл, совершенно не следует, что такой файл и такая директория существуют.

Для определения содержимого родительской папки используем выражение F.getParentFile().listFiles(). Здесь из объекта F вызывается метод getParentFile(). Результатом является ссылка на объект класса File, который «содержит» путь к родительской директории для файла, путь к которому «спрятан» в объекте F. После вызова listFiles() получаем список на массив из объектных ссылок класса File, и каждая ссылка соответствует файлу или директории в родительской директории. Ссылка на такой массив записывается в переменную массива files, после чего для каждого элемента массива отображается название файла или директории, причем с помощью метода isFile() выполняется «идентификация» файлов и директорий, о чем появляется сообщение.

В результате выполнения команды File D=new File(F.getParentFile(),".\\MyFiles") создается объект D класса File, который соответствует директории с названием MyFiles. Эта директория является внутренней директорией в текущей директории для файла, путь к которому реализуется через объект F. Здесь, во-первых, следует учесть, что точка с двойной комой чертой в выражении ".\\MyFiles" означает текущую директорию. А, во-вторых, то, создание объекта D для директории не означает создания самой директории. Соответствующая директория создается командой D.mkdir(). Определить канонический («правильно оформленный») путь к созданной директории можно с помощью команды D.getCanonicalPath(). Но поскольку метод getCanonicalPath() может вызывать контролируемое исключение

класса IOException, при вызове метода необходимо предусмотреть обработку исключений (что мы и сделали).

Командой F.renameTo(new File(D,"MyCV.pdf")) файл, для которого содержится путь в объекте File, меняет свое название на MyCV.pdf и перемещается в директорию, определяемую объектом D. Аргументом методу renameTo() передается анонимный объект класса File, определяющий конечное «состояние» (место хранения и новое название) перемещаемого файла. После этого файл, путь к которому содержится в объекте F, прекращает свое существование. Убеждаемся в этом с помощью команды F.exists().

НА ЗАМЕТКУ

Объект F продолжает существовать и после переименования файла, путь на который содержится в объекте. При этом путь к файлу, содержащийся в объекте F, не изменяется. Просто теперь такого файла нет.

Зато при проверке содержимого вновь созданной директории убеждаемся, что там появился новый файл.

НА ЗАМЕТКУ

Желающие могут сравнить разницу в способе представления методами getPath() и getCanonicalPath() пути к созданной директории.

Чтение из файла и запись в файл

В Java существуют разные способы считывания данных из файла и записи данных в файл, но все они в итоге подразумевают, что используется поток ввода или вывода данных, и этот поток связывается с файлом.

Для начала рассмотрим наиболее простой случай, когда данные из файла считываются по отдельным байтам и записываются в файл байт за байтом. В таком случае используют *байтовый поток*. Для реализации байтового файлового потока ввода используют класс FileInputStream из пакета java.io. Для создания потока ввода создается объект данного класса. Аргументом конструктору передается имя файла (полный путь к файлу), из которого считываются данные. Байтовый файловый поток вывода реализуется через объект класса FileOutputStream. Аргументом конструктору

при создании объекта потока вывода указывается полный путь к файлу, в который выполняется запись. Если указанного аргументом конструктора файла нет, то он будет создан.

НА ЗАМЕТКУ

Если мы говорим о файловом потоке ввода, то имеется в виду ввод информации в программу — то есть считывание данных из файла. Файловый поток вывода используется для вывода данных из программы, и, следовательно, о записи данных в файл.

Конструкторы классов FileInputStream и FileOutputStream могут выбрасывать контролируемое исключение класса FileNotFoundException.

После того как потоки ввода и/или вывода созданы, через соответствующие объекты можно получать данные из файла и записывать данные в файл. Так, для считывания очередного байта из файла из соответствующего потока ввода вызывается метод read(). Результатом метод возвращает считанный из файла байт.

Обычно байты считываются последовательно один за другим, вплоть до окончания файла. Признаком окончания файла является считанное значение -1.

Для записи байта в файл используют метод write(), который вызывается из объекта файлового потока вывода. Аргументом методу write() передается записываемое в файл значение. После завершения работы с файлом необходимо вызвать из объекта связанного с файлом потока метод close(). Если этого не сделать, то некоторые операции по чтению и записи данных могут быть выполнены некорректно. Методы read(), write() и close() могут вызвать контролируемое исключение класса IOException.

НА ЗАМЕТКУ

Класс IOException является суперклассом для класса FileNotFoundException. Поэтому вместо обработки каждого из этих исключений по отдельности можно, например, ограничиться обработкой исключений класса IOException.

В листинге 18.3 представлена очень простая программа, в которой с помощью байтовых файловых потоков выполняется копирование файла.

Листинг 18.3. Программный код проекта FileInputOutputApplication

```
// Импорт классов:
import java.io.*;
// Главный класс:
class FileInputOutputDemo{
  // Главный метод:
  public static void main(String[] args){
    System.out.println("Начинается копирование файла");
    // Путь к директории:
    String path="d:/documents/";
    // Переменная для записи считываемых байтов:
    int bt;
    // Контролируемый код:
    try{
      // Объект файлового потока ввода:
      FileInputStream input=new FileInputStream(path+"MyFiles/MyCV.pdf");
      // Объект файлового потока вывода:
      FileOutputStream output=new FileOutputStream(path+"MyData.pdf");
      // Считывание байта из файла:
      bt=input.read();
      // Считывание байтов из одного файла
      // и запись в другой файл:
      while(bt!=-1){ // Пока не достигнут конец файла
        // Запись байта в файл:
        output.write(bt);
        // Считывание байта из файла:
        bt=input.read();
      }
      // Потоки закрываются:
      input.close();
      output.close();
    }
```

```
    // Обработка исключений:
    catch(FileNotFoundException e){
      System.out.println("Файл не найден: "+e);
    }
    catch(IOException e){
      System.out.println("Ошибка доступа к файлу: "+e);
    }
    System.out.println("Копирование файла завершено");
  }
}
```

Если все проходит штатно, то в результате выполнения программы появляется два сообщения:

Результат выполнения программы (из листинга 18.3)

```
Начинается копирование файла
Копирование файла завершено
```

Теперь рассмотрим более детально и проанализируем программный код примера. А именно, в текстовую переменную path записывается путь к директории, в которую предполагается копировать файл. Также объявляется целочисленная переменная bt, в которую мы собираемся записывать числовые значения (байты), считанные из файла. Прочий код размещен в контролируемом блоке. Там командой FileInputStream input=new FileInputStream(path+"MyFiles/MyCV.pdf") создается объект input байтового файлового потока ввода. Аргументом конструктора указано текстовое значение, определяющее полный путь к копируемому файлу (файлу, из которого считываются значения). Аналогично, командой FileOutputStream output=new FileOutputStream(path+"MyData.pdf") создается объект output для байтового файлового потока вывода. Аргументом конструктору передается полный путь к файлу, в который выполняется копирование. В данном случае конечного файла перед запуском программы нет, но он будет создан в процессе копирования.

Копирование данных из одного файла в другой выполняется байт за байтом с помощью оператора цикла while. Сначала командой bt=input.read() из копируемого файла считывается значение (очередной байт), а командой

output.write(bt) считанный байт записывается в конечный файл. Эти операции выполняются до тех пор, пока истинно условие bt!=-1, то есть пока не считано значение -1 (что означает окончание файла). По окончании копирования командами input.close() и output.close() закрываются файловые потоки ввода и вывода.

Байтовые потоки не всегда удобны в плане прикладного использования, особенно если речь идет о работе с символьными файлами (файлами, содержащими символы). Проблема в том, что символ — это два байта. Поэтому при побайтовом считывании данных могут возникнуть проблемы. Однако есть возможность считывать данные посимвольно (то есть по два байта за один раз). В таком случае говорят о *символьных потоках* ввода и вывода.

Символьный файловый поток ввода реализуется с помощью класса FileReader, а символьный файловый поток вывода реализуется с помощью класса FileWriter. Принципы работы с классами FileReader и FileWriter во многом напоминает методы работы с классами FileInputStream и FileOutputStream, включая наличие методов read(), write() и close(). Небольшая вариация на тему предыдущего примера, но уже с использованием классов FileReader и FileWriter, представлена в листинге 18.4.

Листинг 18.4. Программный код проекта FileReaderWriterApplication

```
// Импорт классов:
import java.io.*;
// Главный класс:
class FileReaderWriterDemo{
  // Главный метод:
  public static void main(String[] args){
    System.out.println("Начинается копирование файла");
    // Путь к директории:
    String path="d:/documents/";
    // Переменная для записи кодов считываемых символов:
    int sm;
    // Контролируемый код:
    try{
      // Объект файлового потока ввода:
```

```
      FileReader input=new FileReader(path+"MyFiles/MyText.txt");
      // Объект файлового потока вывода:
      FileWriter output=new FileWriter(path+"MyNewText.txt");
      // Считывание кода символа из файла:
      sm=input.read();
      // Считывание символов из одного файла
      // и запись в другой файл:
      while(sm!=-1){ // Пока не достигнут конец файла
        // Запись символа в файл:
        output.write(Character.toUpperCase((char)sm));
        // Считывание кода символа из файла:
        sm=input.read();
      }
      // Потоки закрываются:
      input.close();
      output.close();
    }
    // Обработка исключений:
    catch(FileNotFoundException e){
      System.out.println("Файл не найден: "+e);
    }
    catch(IOException e){
      System.out.println("Ошибка доступа к файлу: "+e);
    }
    System.out.println("Копирование файла завершено");
  }
}
```

Перед началом выполнения программы в директории d:\documents\myfiles следует поместить текстовый файл MyText.txt. В результате выполнения программы, если все сделано правильно, появляется такая последовательность сообщений:

Результат выполнения программы (из листинга 18.4)

```
Начинается копирование файла
Копирование файла завершено
```

В результате выполнения программы посимвольно считывается содержимое одного текстового файла и заносится в другой текстовый файл. Причем во втором файле все символы переводятся в верхний регистр (то есть все буквы прописные). Например, допустим, что исходный файл MyText.txt имеет содержимое, как показано на рис. 18.5.

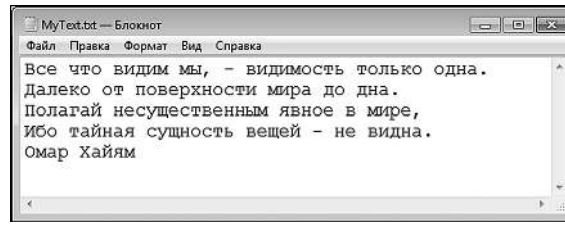


Рис. 18.5. Содержимое копируемого файла

Тогда текстовый файл MyNewText.txt, в который выполняется запись, будет иметь содержание, как показано на рис. 18.6.

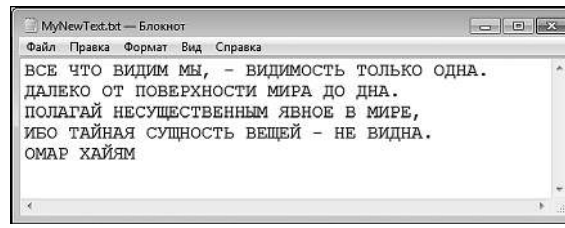


Рис. 18.6. Содержимое файла, в который выполнялась запись

Программный код организован очень схоже к предыдущему примеру. Различие лишь в том, что потоки создаются на основе файлов FileReader и FileWriter. При вызове метода read() из объекта потока ввода получаем числовое значение, которое является кодом символа.

Преобразовать код в символ можно, воспользовавшись инструкцией (char) явного приведения типа. Для преобразования символов в верхний регистр использован статический метод toUpperCase() класса-оболочки Character.

Еще один способ работы с файлами при чтении и записи данных, который мы рассмотрим, базируется на использовании *буферизированных потоков* ввода и вывода.

Особенность буферизированного потока в том, что запись и считывание данных выполняется через буфер. Благодаря этому количество обращений к файлам на диске уменьшается и скорость выполнения операций увеличивается. Есть и дополнительный «бонус», состоящий в том, что считывание и запись данных может выполняться блоками большими, чем байт или символ, — во всяком случае, на программном уровне.

Мы рассмотрим небольшой пример (похожий на два предыдущих), в котором буферизированные потоки ввода и вывода реализуются с помощью классов BufferedReader и BufferedWriter. При создании объекта буферизированного потока аргументом конструктору будет передаваться анонимный объект символьного потока. При считывании содержимого текстового файла через буферизированный поток мы используем метод readLine(), позволяющий считывать целые строки. А теперь рассмотрим программный код, представленный в листинге 18.5.

Листинг 18.5. Программный код проекта BufferedReaderWriterApplication

```
// Импорт классов:
import java.io.*;
// Главный класс:
class BufferedReaderWriterDemo{
  // Главный метод:
  public static void main(String[] args){
    System.out.println("Начинается копирование файла");
    // Путь к директории:
    String path="d:/documents/";
    // Текстовая переменная для записи
    // считываемых из файла строк:
    String str;
    // Контролируемый код:
    try{
      // Объект буферизированного потока ввода:
```

```
        BufferedReader input=new BufferedReader(new FileReader(path+"MyNewText.txt"));
        // Объект буферизированного потока вывода:
        BufferedWriter output=new BufferedWriter(new FileWriter(path+"MyFiles/MyVeryNewText.
txt"));
        // Переменная — счетчик строк:
        int k=1;
        // Считывание строк из одного файла
        // и запись в другой файл:
        while((str=input.readLine())!=null){
          // Запись строки в файл:
          output.write("["+k+"] "+str.toLowerCase());
          // Переход к новой строке:
          output.newLine();
          // Новое значение переменной-счетчика:
          k++;
        }
        // Потоки закрываются:
        input.close();
        output.close();
      }
      // Обработка исключений:
      catch(FileNotFoundException e){
        System.out.println("Файл не найден: "+e);
      }
      catch(IOException e){
        System.out.println("Ошибка доступа к файлу: "+e);
      }
      System.out.println("Копирование файла завершено");
    }
}
```

В данном случае объект input буферизированного потока ввода создается передачей конструктору класса BufferedReader анонимного

объекта класса символьного потока, который создается инструкцией new FileReader(path+"MyNewText.txt"). Таким образом, считывание текста выполняется из файла MyNewText.txt. Аналогичным образом создается объект output буферизированного потока ввода. Аргументом конструктору класса BufferedWriter передается выражение new FileWriter(path+"MyFiles/MyVeryNewText.txt"), которым создается анонимный объект символьного потока вывода. В соответствии с этими командами запись текста выполняется в файл MyVeryNewText.txt.

Для записи считываемых строк объявляется текстовая переменная str, а для нумерации строк используется целочисленная переменная k. В операторе цикла while проверяется условие (str=input.readLine())!=null. Здесь командой input.readLine() считывается текстовая строка из файла, связанного с объектом input буферизированного потока. Считанная строка присваивается значением переменной str. А поскольку оператор присваивания возвращает результат, то значение переменной str можно сравнить с пустой ссылкой null (если достигнут конец файла, то считывать больше нечего и ссылка на считанную строку пустая). В теле оператора цикла командой output.write("["+k+"] "+str.toLowerCase()) в файл, связанный с объектом output потока вывода, записывается текстовое значение, получающееся объединением квадратных скобок, номера строки и текстового значения, считанного из файла в переменную str и преобразованного в нижний регистр (для этого использован метод toLowerCase()).

Для перехода в файле, в котором выполняется запись, к новой строке, выполняется команда output.newLine(), а с помощью команды k++ переменная-счетчик получает новое значение.

Ожидаемый результат выполнения программы в плане выводимых в консоль сообщений такой же, как и в двух предыдущих случаях:

Результат выполнения программы (из листинга 18.5)

```
Начинается копирование файла
Копирование файла завершено
```

Предполагается, что файл MyNewText.txt, из которого выполняется считывание данных, получен в результате выполнения предыдущей программы (см. рис. 18.6). В таком случае в результате построчного считывания и копирования данных из одного текстового файла в другой получим файл MyVeryNewText.txt с содержимым, показанным на рис. 18.7.

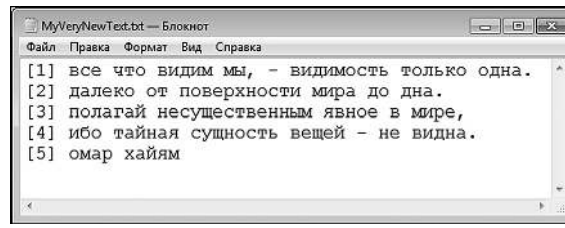


Рис. 18.7. Результат построчного копирования содержимого текстового файла

Текст в файле отображается строчными буквами, а в начале каждой строки указан номер строки (в квадратных скобках).

Средства выбора файлов

В рассмотренных ранее примерах ссылки на файлы мы указывали явно путем записи в текстовое значение пути к файлу. На практике намного удобнее искать и выбирать файл в файловой системе с помощью специального окна выбора файлов.

Мы познакомимся с классом JFileChooser из пакета javax.swing. С помощью объекта класса JFileChooser легко организовать процесс выбора файла через отображение специального диалогового окна, позволяющего выбрать нужный файл непосредственно в файловой системе. Мы рассмотрим небольшой пример, в котором пользователю предлагается выбрать файл, после чего этот файл копируется в место на диске, которое выбирает пользователь. Наибольший интерес для нас представляют методы showOpenDialog() и showSaveDialog() класса JFileChooser. С помощью этих методов собственно и открывается диалоговое окно для выбора файла. Для объяснения принципов использования данных методов сразу обратимся к примеру, представленному в листинге 18.6.

Листинг 18.6. Программный код проекта FileChooserApplication

```
// Статический импорт:
import static javax.swing.JOptionPane.*;
// Нестатический импорт:
import javax.swing.*;
import java.io.*;
// Главный класс:
public class FileChooserDemo{
```

```
// Главный метод (выбрасывает контролируемое
// исключение класса IOException):
public static void main(String[] args) throws IOException{
  // Отображение окна подтверждения:
  int res=showConfirmDialog(null,
    "Готовы выбрать файл?", // Текст в окне
    "Работа с файлами", // Название окна
    YES_NO_OPTION, // Количество и тип кнопок
    QUESTION_MESSAGE // Тип пиктограммы
  );
  // Если не нажата кнопка подтверждения:
  if(res!=YES_OPTION){
    // Завершение выполнения программы:
    System.exit(0);
  }
  // Создание объекта класса JFileChooser:
  JFileChooser fch=new JFileChooser();
  // Отображение окна для выбора файла:
  res=fch.showOpenDialog(null);
  // Если не подтвержден выбор файла:
  if(res!=JFileChooser.APPROVE_OPTION){
    // Завершение выполнения программы:
    System.exit(0);
  }
  // Объект класса File, определяющий
  // файл, выбранный в окне выбора файлов:
  File F=fch.getSelectedFile();
  // Формирование текста для отображения в сообщении:
  String txt="Вы выбрали файл\n";
  txt+=F.getAbsolutePath(); // Путь к файлу
  txt+="\nХотите создать копию?";
  // Отображение окна подтверждения:
  res=showConfirmDialog(null,
```

```
    txt, // Текст сообщения
    "Файл выбран", // Название окна
    YES_NO_OPTION, // Количество и тип кнопок
    QUESTION_MESSAGE // Тип пиктограммы
  );
  // Если не нажата кнопка подтверждения:
  if(res!=YES_OPTION){
    // Завершение выполнения программы:
    System.exit(0);
  }
  // Отображение окна для сохранения файла:
  res=fch.showSaveDialog(null);
  // Если выбор не подтвержден:
  if(res!=JFileChooser.APPROVE_OPTION){
    // Завершение выполнения программы:
    System.exit(0);
  }
  // Создание объекта для байтового потока ввода
  // на основе копируемого файла:
  FileInputStream input=new FileInputStream(F);
  // Объекта класса File, определяющий файл,
  // в который выполняется копирование:
  F=fch.getSelectedFile();
  // Создание объекта для потока вывода на основе
  // файла, в который выполняется копирование:
  FileOutputStream output=new FileOutputStream(F);
  // Побайтовое копирование:
  while((res=input.read())!=-1){
    output.write(res);
  }
  // Закрываются потоки:
  input.close();
  output.close();
```

```
    // Отображение окна с сообщением о создании
    // копии файла:
    showMessageDialog(null,
      // Текст сообщения в окне:
      "Копия сохранена как\n"+F.getAbsolutePath(),
      // Название окна:
      "Создана копия файла",
      // Тип пиктограммы:
      INFORMATION_MESSAGE
    );
  }
}
```

В первую очередь стоит отметить, что в программе использовано несколько import-инструкций. Так, мы используем статический импорт для класса JOptionPane. Это позволит нам использовать статические методы и константы данного класса без явной ссылки на класс. Также импортируются пакеты javax.swing и java.io. Первый пакет необходим, поскольку мы, как упоминалось, используем класс JFileChooser, а второй из упомянутых пакетов задействован, поскольку мы используем в программе класс File и классы для организации байтовых файловых потоков ввода и вывода.

В сигнатуре главного метода программы использована инструкция throws IOException, означающая, что главный метод может выбрасывать исключение соответствующего класса. Причина в том, что в программе, во-первых, создаются потоки классов FileInputStream и FileOutputStream, а при этом может возникнуть исключение класса FileNotFoundException, и, во-вторых, при вызове методов read(), write() и close() может возникнуть исключение класса IOException. Чтобы не выполнять обработку исключений, мы и добавили в сигнатуру главного метода throws-инструкцию. Причем мы учли, что класс FileNotFoundException является подклассом класса IOException, так что в сигнатуре метода main() достаточно указать только класс IOException.

В первую очередь в теле главного метода вызывается статический метод showConfirmDialog() из класса JOptionPane. Первым аргументом методу передается пустая ссылка null на родительское окно (то есть такого окна нет). Второй аргумент метода "Готовы выбрать файл?" определяет текст сообщения в окне. Третий аргумент метода "Работа с файлами" задает название окна.

Статическая константа YES_NO_OPTION из класса JOptionPane, переданная четвертым аргументом методу, определяет количество и название кнопок (в данном случае будут кнопки с названиями **Yes** и **No**). И, наконец, статическая константа QUESTION_MESSAGE определяет тип пиктограммы, отображаемой в окне (пиктограмма со знаком вопроса). Окно, отображаемое в результате выполнения указанной команды, будет иметь вид, как на рис. 18.8.

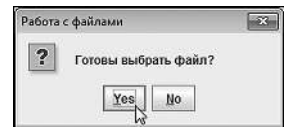


Рис. 18.8. Окно подтверждения для начала процедуры выбора файла

Метод showConfirmDialog() результатом возвращает целочисленное значение, позволяющее идентифицировать кнопку, которую пользователь нажал в окне. Мы результат вызова метода записываем в целочисленную переменную res. Для продолжения работы необходимо, чтобы пользователь щелкнул кнопку **Yes**. В таком случае метод showConfirmDialog() значением возвращает статическую константу YES_OPTION из класса JOptionPane. В условном операторе проверяется условие res!=YES_OPTION, которое истинно, если значение переменной res отлично от значения константы YES_OPTION. Если так, то командой System.exit(0) завершается выполнение программы.

НА ЗАМЕТКУ

Если пользователь закрывает представленное выше окно щелчком на кнопке No, то метод showConfirmDialog() возвращает результатом значение константы NO_OPTION. Если окно закрывается с помощью системной пиктограммы с крестиком, то результат метода совпадает со значением константы CANCEL_OPTION.

Таким образом, все дальнейшие действия выполняются, только если в первом появившемся окне пользователь щелкает кнопку **Yes**. А именно, командой JFileChooser fch=new JFileChooser() создается объект fch класса JFileChooser. Ценность объекта в том, что мы с его помощью можем отобразить диалоговое окно, в котором выбирается файл. Делается это легко: из объекта класса JFileChooser вызывается метод showOpenDialog(). Аргументом методу передается ссылка на родительское окно. Поскольку такого в данном случае нет, мы используем в качестве аргумента пустую ссылку null.

Команда, которой отображается окно, имеет вид res=fch.showOpenDialog(null). Окно, которое при этом появляется, показано на рис. 18.9.

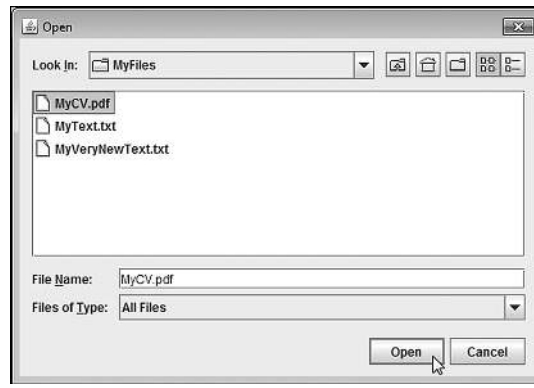


Рис. 18.9. Окно открытия файла

Это, в общем-то, стандартное окно, в котором пользователь может выбрать файл и подтвердить свой выбор, щелкнув кнопку **Open**. Но дело в том, что пользователь может и передумать, щелкнув кнопку **Cancel** или закрыв окно с помощью системной пиктограммы. Узнать, какова была реакция пользователя, можно по результату, который возвращает метод showOpenDialog(). Результат у метода целочисленный, и если пользователь в окне выбрал файл и подтвердил свой выбор щелчком на кнопке **Open**, то результатом метод showOpenDialog() возвращает значение статической константы APPROVE_OPTION из класса JFileChooser. Поскольку результат вызова метода showOpenDialog() мы записываем в переменную res, то в условном операторе проверяется условие res!=JFileChooser.APPROVE_OPTION, и при его истинности выполняется команда System.exit(0) (завершается выполнение программы). Поэтому все, что описано дальше, имеет место, только если пользователь выбрал файл.

То, что в окне выбран файл, ни к каким явным действиям не приводит — сам по себе файл открываться не будет. Просто если файл выбран, то объект fch, из которого вызывался метод showOpenDialog(), содержит сведения об этом файле. В частности, с помощью метода getSelectedFile() можно получить ссылку на объект класса File, через который реализуется доступ к выбранному файлу на диске. Мы с помощью команды File F=fch.getSelectedFile() записываем ссылку на объект для выбранного пользователем файла в объектную переменную F класса File. После этого формируется текстовая строка txt, содержащая информацию о выбранном файле

(полный путь к файлу получаем с помощью команды F.getAbsolutePath()). Далее отображается еще одно окно с информацией о выбранном файле и вопросом о том, желает ли пользователь создать копию файла. Как может выглядеть такое окно, показано на рис. 18.10.

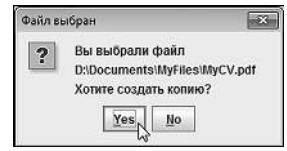


Рис. 18.10. Окно с информацией о выбранном файле

Если пользователь желает создать копию файла, то командой res=fch.showSaveDialog(null) отображается диалоговое окно, предназначенное для определения места и названия сохраняемого файла. Окно показано на рис. 18.11.

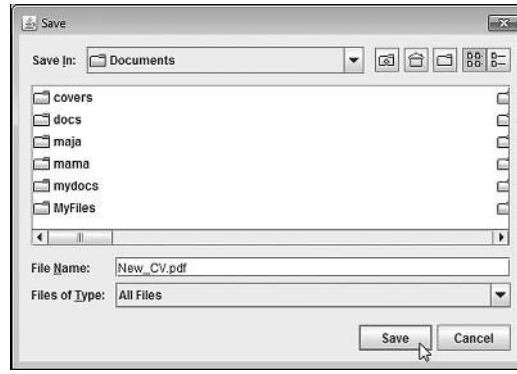


Рис. 18.11. Окно сохранения файла

Аргументом методу showSaveDialog() передается пустая ссылка null, свидетельствующая о том, что родительского окна у открываемого диалогового окна нет. Результат метода showSaveDialog() записывается в переменную res. Если в открывшемся окне пользователь укажет в поле **File Name** имя для сохраняемого файла (это может быть имя несуществующего файла) и подтвердит свой выбор щелчком на кнопке **Save**, то результатом метод вернет значение APPROVE_OPTION. Поэтому в условном операторе проверяется значение переменной res, и если пользователь решил сохранить копию файла, то собственно начинается процесс копирования.

НА ЗАМЕТКУ

Щелчок на кнопке **Save** в окне сохранения файла не приводит к автоматическому сохранению файла. Результатом выбора места для сохранения и имени файла является то, что сведения о выбранном пути и файле заносятся в объект, из которого вызывался метод showSaveDialog().

Если принято решение на сохранение копии, то командой FileInputStrea m input=new FileInputStream(F) создается объект input для байтового файлового потока ввода. Данный поток связан с файлом, который был выбран при вызове метода showOpenDialog() (копируемый файл). После этого командой F=fch.getSelectedFile() в переменную F записывается ссылка на объект для файла, который был определен при вызове метода showSaveDialog(). На основе этого объекта командой FileOutputStream output=new FileOutputStream(F) создается объект output для потока байтового файлового вывода. Данный поток связан, таким образом, с файлом-копией.

Побайтовое копирование выполняется с помощью оператора цикла while. Условием в операторе указано выражение (res=input.read())!=-1. Здесь сначала командой res=input.read() считывается байт из копируемого файла, и это значение записывается в переменную res.

Если такое значение отлично от -1 (конец файла не достигнут), то командой output.write(res) в теле оператора цикла считанный байт из копируемого файла записывается в файл-копию.

По окончании копирования командами input.close() и output.close() закрываются потоки ввода и вывода. После этого отображается окно с информацией о файле-копии. На рис. 18.12 показано, как такое окно может выглядеть.

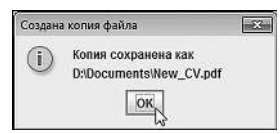


Рис. 18.12. Окно с информацией о сохраненном файле

В данном случае мы фактически приводим только полный путь к сохраненному файлу, который можно получить, вызвав из объекта F метод getAbsolutePath().

Резюме

Сам не пойму, что со мной происходит. Как начну речь произносить, не те слова идут!

Из к/ф «Безумный день инженера Баркасова»

- При запуске программе можно передавать некоторые параметры или аргументы, которые затем используются при выполнении программы. Параметры в программу передаются в текстовом формате и реализуются в виде текстового массива, объявленного аргументом метода main().
- Для работы с файлами в Java предусмотрены специальные классы. Так, с помощью объектов класса File из пакета java.io можно получить основную информацию о файле и выполнять с файлами некоторые базовые операции.
- Для чтения данных из файла и записи данных в файл создают потоки ввода и вывода, связанные с соответствующими файлами. Байтовые потоки создаются на основе классов FileInputStream и FileOutputStream (пакет java.io). При создании символьных потоков используют классы FileReader и FileWriter. Для создания буферизированных потоков можно использовать классы BufferedReader и BufferedWriter.
- С помощью объекта класса JFileChooser (пакет javax.swing) можно организовать процедуру выбора файла на диске.

Заключение
ЕЩЕ НЕМНОГО О JAVA

Официально заявляю, что за всё, что здесь сегодня было, я лично никакой ответственности не несу!

Из к/ф «Карнавальная ночь»

Изучение языка программирования — процесс интересный, иногда сложный, иногда приятный, но он практически никогда не бывает полностью завершенным. Всегда есть вопросы, ответы на которые не очевидны. Это замечание особенно справедливо, если речь идет о языке Java. Поэтому наивно было бы полагать, что в одной книге можно описать и объяснить все тонкие моменты, связанные с программированием в Java. Откровенно говоря, такая задача и не ставилась. Важно было другое — показать красоту, эффективность, «гибкость» и универсальность этого языка программирования. Хотя по отношению к Java термин «язык программирования» является слишком узким. На самом деле Java — это целая технология, которая постоянно развивается и совершенствуется. Важно быть частью этого процесса, тоже развиваться и совершенствоваться. Только в этом случае можно рассчитывать на успех в прикладном применении своих навыков программирования в Java.

Хочется верить, что книга послужит путеводителем, другом и помощником для тех, кто решил приобщиться к сообществу Java-программистов. А главный совет состоит в том, что лучший способ научиться программированию — программировать. Ну и еще читать книги.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ