Васильев А. Н.

Java

с примерами и программами

САМОУЧИТЕЛЬ

Васильев А. Н.

САМОУЧИТЕЛЬ

Java

с примерами и программами

3-е издание

Наука и Техника
Санкт-Петербург
2016

Васильев А. Н.

Самоучитель Java с примерами и программами. 3-е издание. — СПб.: Наука и Техника, 2016. — 368 с.: ил.

Серия «Самоучитель»

Данная книга является превосходным и эффективным учебным пособием для изучения языка программирования Java с нуля. Книга задумывалась, с одной стороны, как пособие для тех, кто самостоятельно изучает язык программирования Java, а с другой, она может восприниматься как лекционный курс с проведением практических занятий. Книга содержит полный набор сведений о синтаксисе и концепции языка Java, необходимый для успешного анализа и составления эффективных программных кодов. Материал книги излагается последовательно и сопровождается большим количеством наглядных примеров, разноплановых практических задач и детальным разбором их решений.

Книга отличается предельной ясностью, четкостью и доступностью изложения, что вкупе с обширной наглядной практикой (примерами и программами) позволяет ее рекомендовать как отличный выбор для изучения Java.

Внимание! Все дополнительные материалы к книге (программные коды, примеры и проч.) доступны для скачивания на сайте издательства: www.nit.com.ru.

Контактные телефоны издательства:
(812) 412 70 25, (812) 412 70 26, (044) 516 38 66
Официальный сайт: www.nit.com.ru

Содержание


Java™

Введение

О книге, языке и программном обеспечении

Эта книга о том, как быстро научиться писать более-менее приличные программы на языке Java. Задача сложная, но вполне реальная. Во всяком случае, в книге сделана попытка упростить все до предела. Поэтому не стоит удивляться, что нередко в книге сначала описывается, что и как нужно сделать, и только впоследствии объясняется, почему все это работает.

Технология Java

Пикантность ситуации придает то обстоятельство, что Java – это не только язык программирования, но и технология, которую называют тем же словом. Обратимся к фактам.

В 1991 году с подачи *Патрика Ноутона*, инженера компании *Sun Microsystems*, и при участии *Джеймса Гослинга* (члена Совета директоров компании, кстати) был запущен проект по разработке средств компьютерной поддержки электронных компонентов всевозможных приборов (в основном бытовых). Проект базировался на разработке специального языка программирования – простого, удобного, универсального и, очень важно, экономного с точки зрения расходуемой памяти. Проект поначалу оказался не очень удачным, несколько раз менял название и концепцию, пока, наконец, в 1995 году увидел свет под именем Java. В конечной версии проекта получился язык программирования, ориентированный для использования в сети Интернет с сопутствующими средствами поддержки. На войне, как на войне: начинали с бытовых приборов, закончили универсальным языком программирования.

В основу новой технологии была положена модель *виртуальной машины*. Эта же идея, кстати, использовалась при создании первых версий языка Pascal. Она проста и элегантна и позволяет решить главную, фундаментальную проблему – проблему универсальности программного кода. Все очень просто: как говорится, ловкость рук и никакого мошенничества. Дело в том, что программа, написанная на каком-нибудь языке программирования, понятном для программиста (в том числе и на Java), должна быть переведена в набор инструкций, понятных для компьютера, или в *машинный код*. Эту почетную миссию берут на себя программы-компиляторы. Проблема в том, что компьютеры у разных программистов разные. Машинный код,

понятный для одного компьютера, в принципе не обязательно должен быть понятен для другого. Поэтому компилятор Java-программы переводит программный код не в машинный, а в так называемый *промежуточный код*, или *байт-код*. Он один для всех типов компьютеров. Особенности того или иного компьютера учитывает *виртуальная Java-машина* – программа, которая предварительно устанавливается на компьютер и под управлением которой выполняется байт-код. Ситуация примерно такая, как если бы нужно было перевести для большого количества иностранных туристов какой-то текст (например, русский). Проблема в том, что иностранцы разные, говорят на разных языках, и переводчиков на всех не хватает. Поэтому текст переводится на английский, а иностранцев предварительно обучают этому языку. Английский в данном случае играет роль байт-кода, а обучение иностранцев английскому языку сродни установке на компьютер виртуальной Java-машины. Упреждая естественный вопрос о том, почему бы иностранцам вместо английского сразу не выучить русский, заметим, что русский язык по сравнению с английским намного сложнее. Есть сомнения – возьмите вместо русского китайский.

Помимо модели виртуальной машины, в языке программирования Java реализована концепция *объектно-ориентированного программирования* (сокращенно *ООП*). Другими словами, язык Java – объектно-ориентированный язык программирования. Читатель морально должен быть готов к тому, что сплошь и рядом по тексту встречаются такие ласкающие слух слова, как *класс* и *объект*. В книге встречаются и другие не менее красивые слова: *интерфейс*, *пакет*, *метод*, *поле*, *поток* – список достаточно большой. Читатель в этом вскоре убедится.

В технологии Java существует несколько основных направлений:

- *Java Enterprise Edition* (сокращенно Java EE или J2EE) – технология для создания программного обеспечения уровня больших корпораций.
- *Java Standard Edition* (сокращенно Java SE или J2SE) – технология создания пользовательских приложений.
- *Java Micro Edition* (сокращенно Java ME или J2ME) – технология создания приложений для мобильных телефонов, карманных персональных компьютеров и других маломощных вычислительных систем.
- *Java Card* – технология для устройств типа смарт-карт.

В книге будет использована технология *Java Standard Edition*, то есть стандартный дистрибутив Java.

Программное обеспечение

Приятно, что практически все необходимое для работы с Java программное обеспечение имеется в открытом (то есть бесплатном) доступе и может быть загружено через Интернет. Оптимальный джентльменский набор состоит из трех сайтов. Первые два – сайты поддержки языка программирования Java `www.java.com` и `www.java.sun.com`. Еще один — сайт `www.netbeans.org`. Здесь есть очень полезные утилиты, но самое главное, бесплатная и эффективная программа *NetBeans* – интегрированная среда разработки, благодаря которой работа по написанию Java-кодов и их компиляции становится исключительно простой.

Условно процесс создания программы можно разбить на несколько этапов. Первый, самый важный, связан с написанием программного кода. Программный код пишется на определенном языке программирования – например, на Java. В принципе, набирать код можно хоть в текстовом редакторе. Вопрос в том, что потом с таким кодом делать. Точнее, как его потом компилировать. Кроме того, при наборе кода желательно иметь возможность проверять в автоматическом режиме корректность синтаксиса команд и инструкций программы. Такие возможности предоставляют специальные редакторы программных кодов, а еще лучше – интегрированные среды разработки (аббревиатура *IDE* от *Integrated Development Environment*). Это такой очень продвинутый редактор кодов, который позволяет в удобном режиме набрать программный код, проверить его (на уровне синтаксических конструкций), откомпилировать и запустить программу на выполнение. Программные средства, необходимые для компиляции и запуска программы, могут поставляться вместе с интегрированной средой разработки, а могут устанавливаться отдельно. Последний вариант хуже, поскольку обычно приходится выполнять некоторые настройки вручную.

Чтобы начать работу с Java, необходимо установить на компьютер систему JDK (аббревиатура от *Java Development Kit*). Это комплект для разработки приложений на языке Java, разработанный компанией *Sun Microsystems* и распространяемый бесплатно. В состав пакета входит стандартный компилятор (файл `javac.exe`), стандартные библиотеки классов, примеры программы, документация и исполняющая система Java (комплект JRE – аббревиатура от *Java Runtime Environment*). В свою очередь, исполняющая система Java, реализованная в комплекте программ и программных пакетов JRE, содержит утилиты, необходимые для выполнения байт-кодов. Среди этих утилит и интерпретатор `java.exe`. Комплект JRE, хотя и входит в состав JDK, распространяется также и отдельно. Это удобно, поскольку позволяет отдельно обновлять исполняющую систему Java. Далее показано,

как обновляется JRE, устанавливается JDK и интегрированная среда разработки NetBeans. Начнем с установки системы JDK.

Выходим на сайт `www.java.sun.com` и в разделе Downloads выбираем что-нибудь приличное – например, Java SE (рис. В.1).

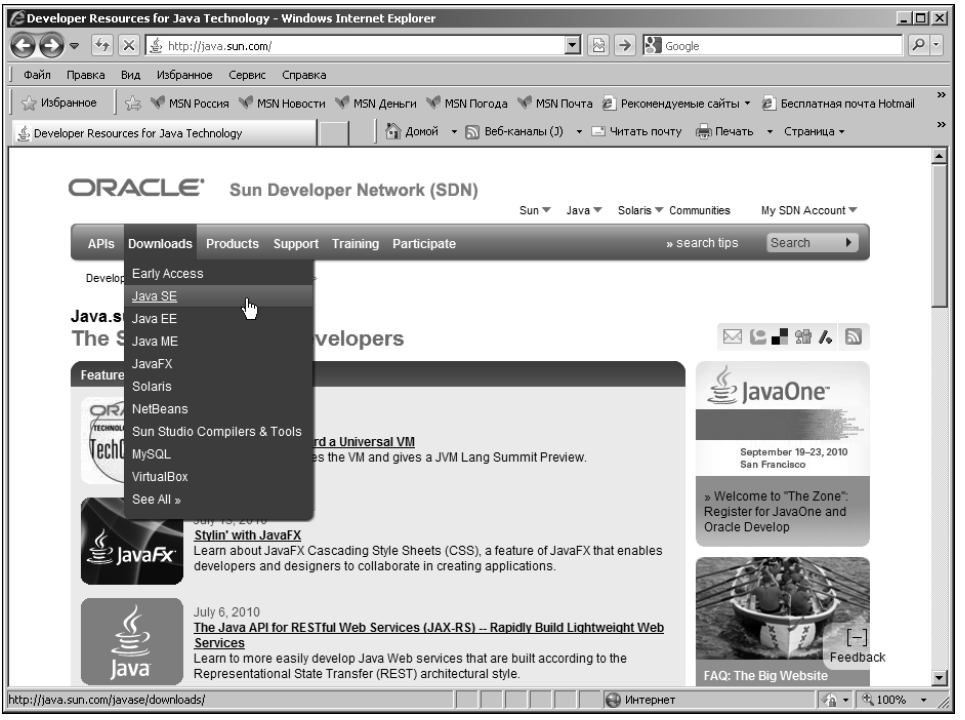


Рис. В.1. Страница www.java.sun.com

Присутствие в большом количестве на сайте следов компании *Oracle* пугать не должно – эта *Sun Microsystems* была поглощена корпорацией *Oracle*, отсюда и ее логотипы повсеместно на сайтах поддержки Java. Но вернемся к загрузке JDK. В новом открывшемся окне необходимо выбрать тип загружаемого программного обеспечения. Окно показано на рис. В.2.

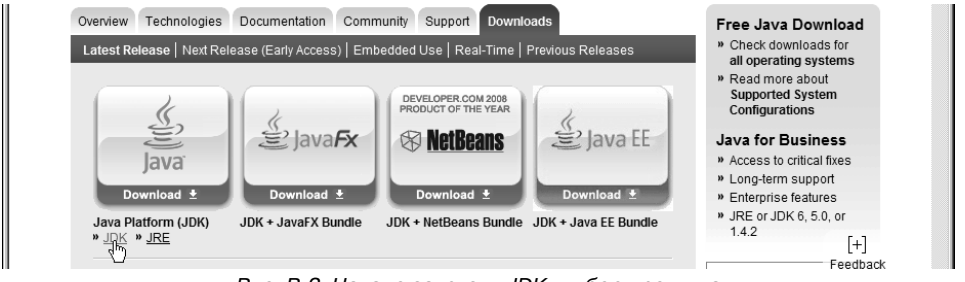


Рис. В.2. Начало загрузки JDK: выбор продукта

Обращаем внимание читателя, что там достаточно много полезных утилит, включая ссылку на страницу загрузки интегрированной среды NetBeans (вместе с JDK). Мы простым путем не пойдем и будем грузить и устанавливать все в отдельности. Поэтому для загрузки выбираем просто JDK. В следующем окне нужно указать платформу (тип операционной системы) и щелкнуть кнопку Download (рис. В.3).

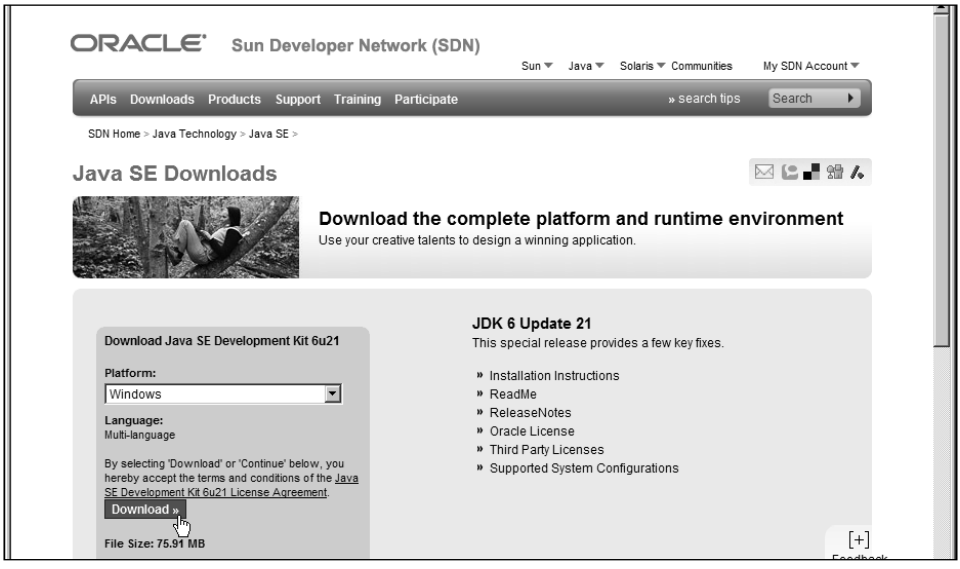


Рис. В.3. Выбор типа загрузки

Следующий неприятный момент может несколько охладить пыл любителей бесплатного программного обеспечения: появляется окно регистрации (рис. В.4).

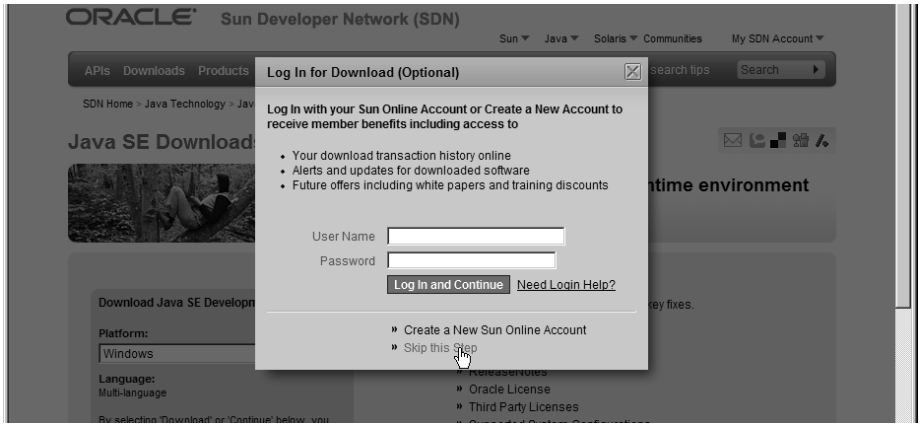


Рис. В.4. Отказ от создания учетной записи

Можно, конечно, зарегистрироваться – денег за это не возьмут и даже не будут их просить. Но есть простой военный маневр, который экономит время и нервы: щелкаем на ссылке Skip this Step, и вопрос с регистрацией снимается сам собой. После этого, собственно, получаем доступ к файлу установки JDK для загрузки (рис. В.5).

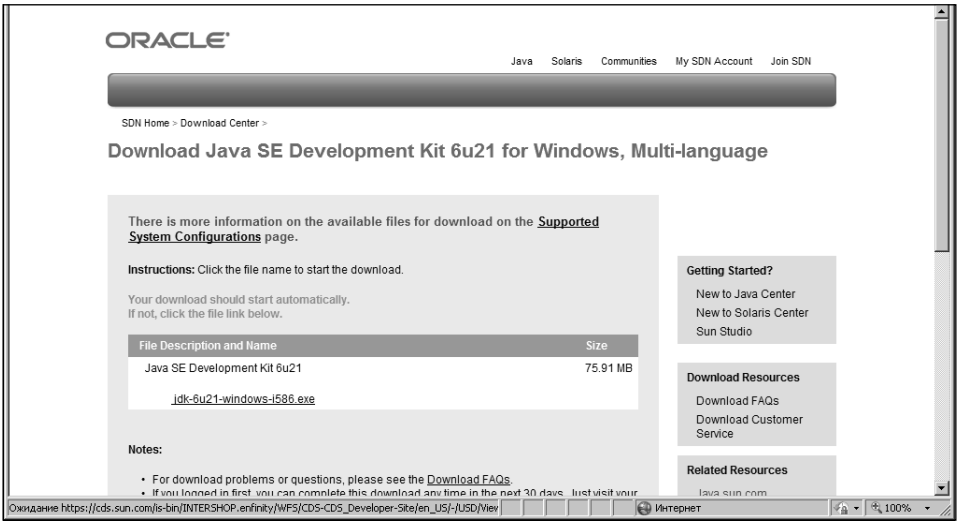


Рис. В.5. Файл для загрузки

На всякий случай для тех, кто никогда не видел таких окон, на рис. В.6 показано окно запроса на загрузку файла.

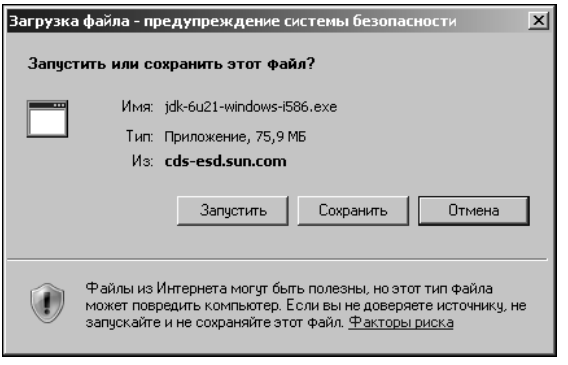


Рис. В.6. Загрузка JDK: диалоговое окно Загрузка файла

На всякий случай файл лучше сохранить на диск – вдруг еще пригодится. Другими словами, щелкаем кнопку Сохранить. После этого необходимо подождать окончания загрузки файла. Загруженный файл запускаем на выполнение. Откроется окно, как на рис. В.7.

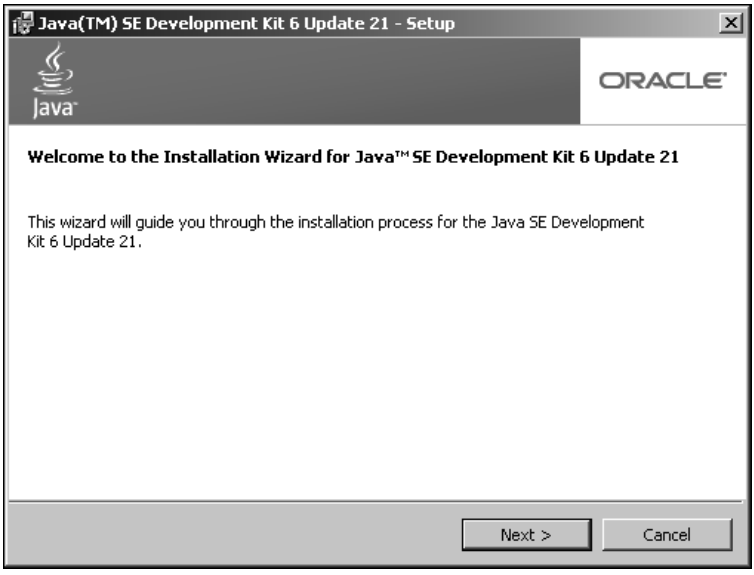


Рис. В.7. Начало установки JDK

Выбор в данном случае небольшой, поэтому щелкаем на кнопке Next и начинаем установку JDK. На рис. В.8 показано окно, в котором выполняются настройки устанавливаемого комплекта JDK. Если крайней необходимости нет, там лучше ничего не менять.

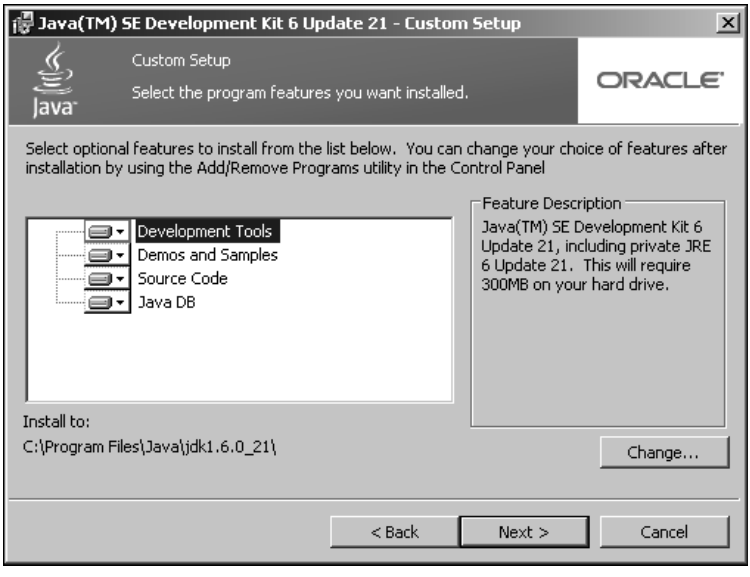


Рис. В.8. Настройка параметров установки JDK

Еще раз щелкаем кнопку Next и ждем, пока не появится окно, как на рис. В.9.

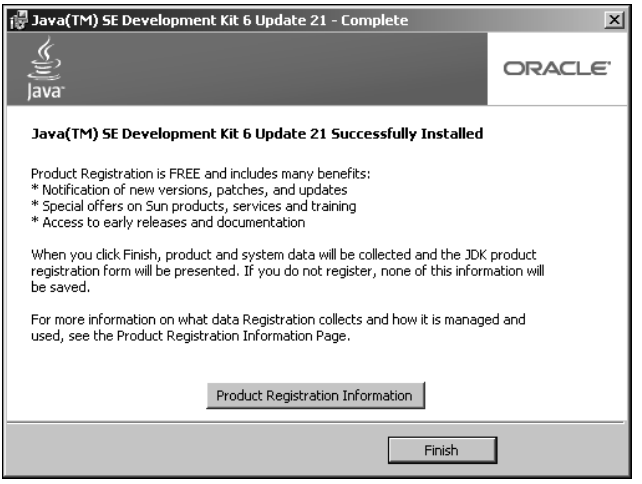


Рис. В.9. Установка JDK завершена

После этого можно считать, что дорога к программированию на Java открыта. Если вдруг понадобится ее еще расширить путем обновления исполнительной системы Java, в отчаяние не впадаем, а сразу выходим на сайт `www.java.com` (рис. В.10). Выходим и начинаем загружать, как показано на рис. 11. После загрузки запускаем обновление.

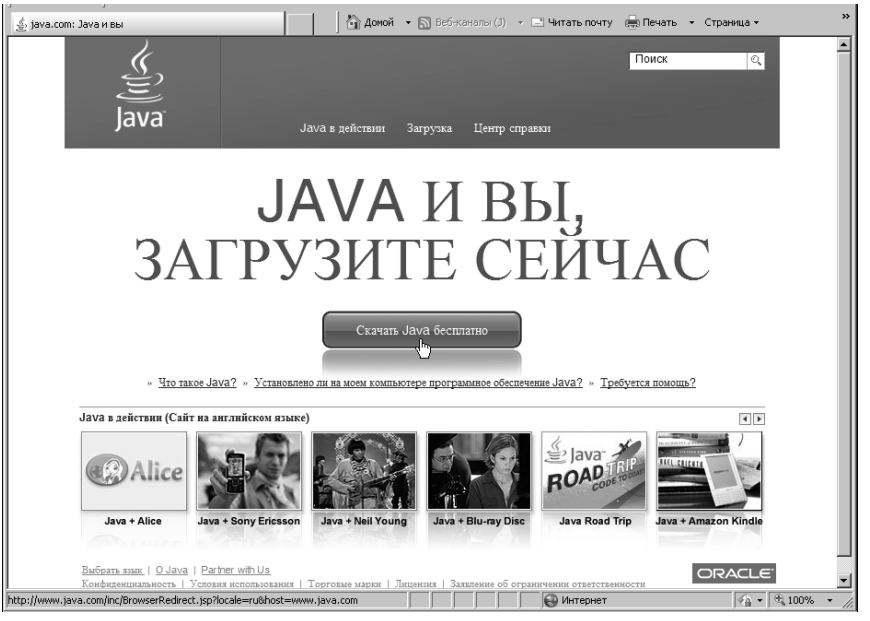


Рис. В.10. Страница www.java.com

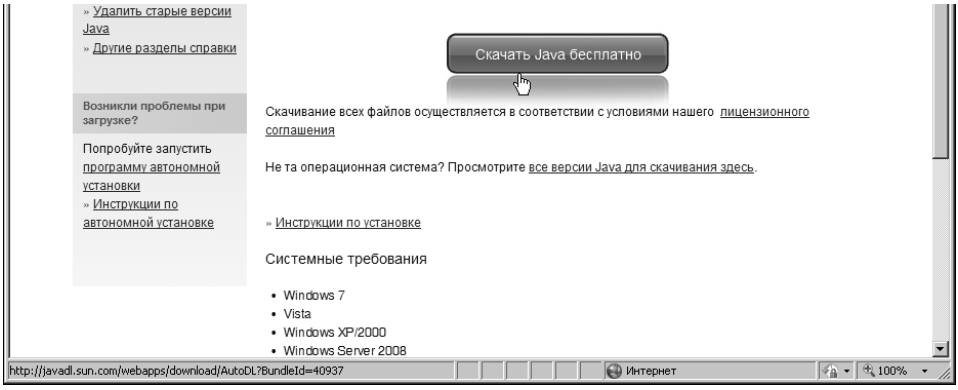


Рис. В.11. Обновление JRE: загрузка файла

Когда этот комплект наконец установлен (обновлен), появляется окно, как на рис. В.12.

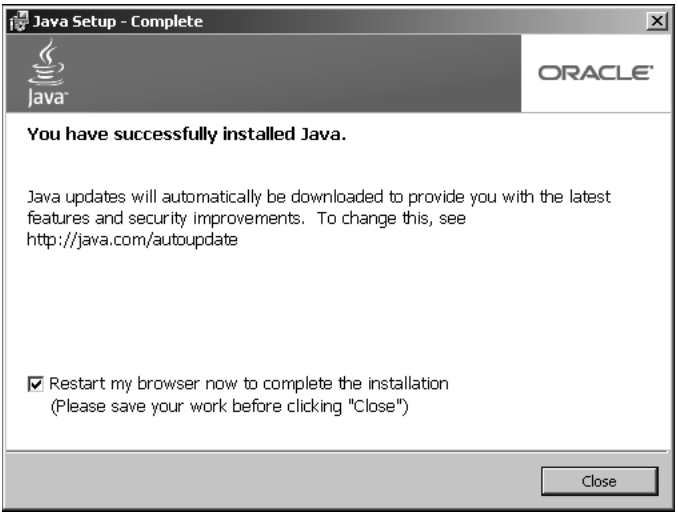


Рис. В.12. Установка обновлений JRE завершена

Собственно говоря, и это отмечалось, наличия JDK достаточно для компиляции и выполнения программ на Java. Достаточно только настроить операционную систему в той ее части, чтобы она знала, где какие файлы искать. Но все это даже описывать скучно, не то что читать. Поэтому любителей ручной компиляции программ отсылаем к другим книгам или фирменной справке. Здесь же поступим просто и мудро: опишем, где загрузить и как установить приличную среду разработки. Ею и будем пользоваться в дальнейшем. Скачать ее можно либо с сайта `www.netbeans.org`.

Запускаем установочный файл на выполнение. В результате появится окно, как на рис. В.13.

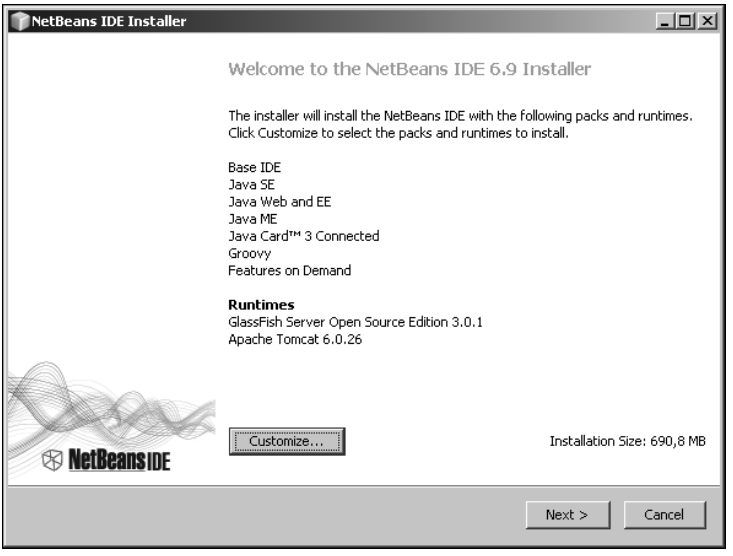


Рис. В.13. Запуск установки среды NetBeans

Хотя программное обеспечение бесплатное, лицензия на него есть, и лицензию нужно принять – иначе никто ничего не установит. В специальном диалоговом окне необходимо указать место для интегрированной среды и каталог, в который установлен комплект JDK (рис. В.14).

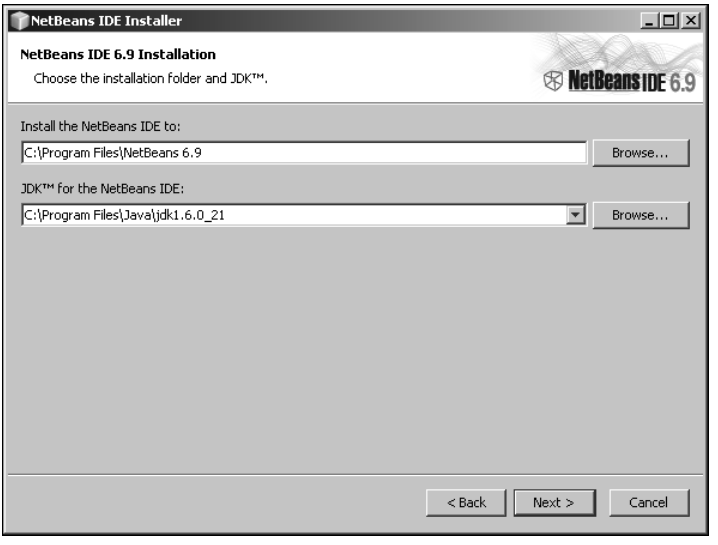


Рис. В.14. Место установки среды NetBeans

Для установки среды каталог предлагается по умолчанию в папке с программными файлами. Что касается JDK, то если он на компьютере установлен, то, скорее всего, место установки JDK будет определено автоматически и указано в соответствующем поле. В крайнем случае, можно место, куда установлен JDK, определить вручную. В процессе установки также может понадобиться указать место для установки пакетов интегрированной среды (рис. В.15).

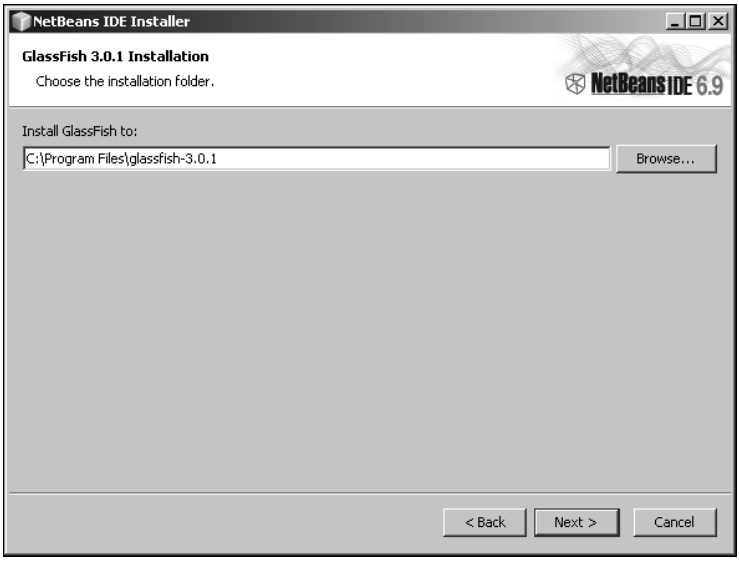


Рис. В.15. Место установки утилит среды NetBeans

Когда все формальности выполнены, начинается непосредственно процесс установки. Процедура может быть небыстрой. Окно приложения NetBeans показано на рис. В.16.

Приемы работы с приложением NetBeans (в минимальном необходимом объеме) описываются по ходу книги в контексте создаваемых программ. Здесь только отметим, что флажок опции Show On Startup лучше убрать, если, конечно, читатель не хочет каждый раз закрывать окно-заставку вручную.

Еще одно замечание хочется сделать по поводу описанных выше красивых окон и веб-страниц. Они имеют нехорошее свойство со временем становиться еще красивее, а то и вообще радикально меняют внешний вид. Если к моменту, как книга попала к читателю, такое действительно произошло, не теряйте оптимизма. Доверяйте своей интуиции и помните, что лучше один раз самому найти нужный файл для закачки, чем сто раз прочитать об этом в книге.

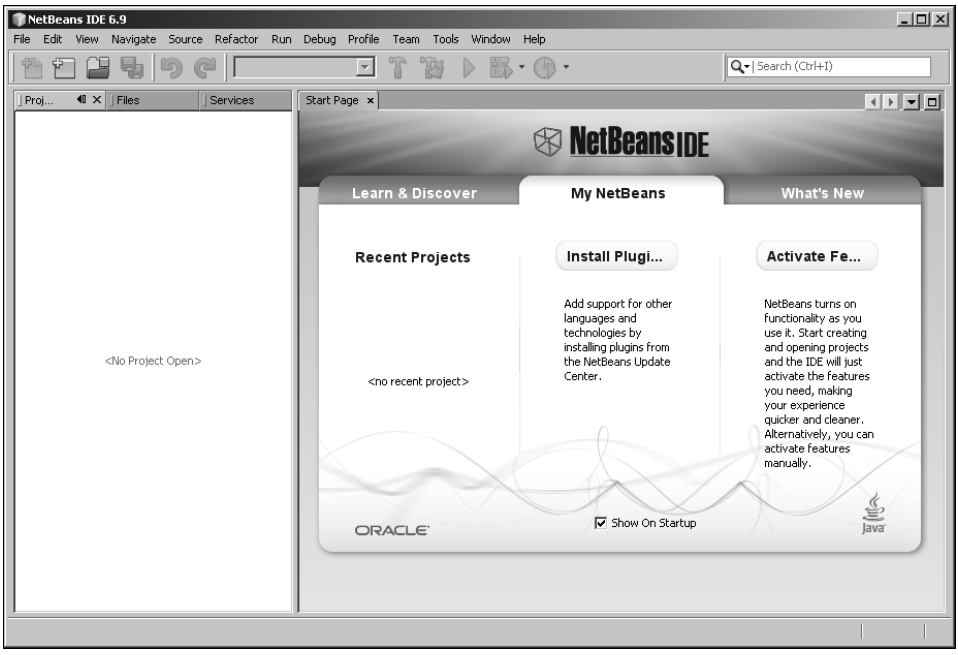


Рис. В.16. Окно приложения NetBeans

Про ООП вообще и Java в частности

Есть такая штука, как парадигма программирования. Это совокупность правил, идей и подходов, которые в совокупности определяют стиль программирования, то есть стиль написания программных кодов. Важными показателями при анализе парадигмы программирования являются синтаксис используемого языка программирования и набор стандартных библиотек. С одной стороны, синтаксис языка подчиняется определенной парадигме, которая закладывается еще при создании языка программирования. С другой стороны, парадигма реализуется средствами синтаксиса языка программирования. Здесь проявляется принцип "выше головы не прыгнешь" — если в синтаксисе языка что-то не предусмотрено, значит этого нет.

В рамках парадигмы *структурного программирования* программа представляет собой последовательность инструкций, которые выполняются одна за другой. При этом широко используются *операторы цикла*, позволяющие многократно повторять однотипные действия, а также *условные операторы*, позволяющие одноразово выполнять то или иное действие в зависимости от истинности определенных условий. Также допускается использование

именованных блоков программного кода — *процедур* и *функций*. Это своеобразные подпрограммы, которые предназначены упростить составление программного кода.

Нас интересует парадигма *объектно-ориентированного программирования*. Именно она реализуется в Java. В рамках этой парадигмы программа представляет собой набор взаимодействующих *объектов*. Обычно (например, в Java, C++ и C#) объекты создаются на основе *классов*.

Класс описывает определенный тип, объединяющий не только данные, но и программный код для обработки этих данных. Класс является тем шаблоном, на основе которого собственно и создаются объекты. Таким образом, концепция классов и объектов является краеугольным камнем для парадигмы объектно-ориентированного программирования. И это не случайно.

Дело в том, что с помощью классов и их конкретных экземпляров — объектов, изменяется способ структурирования программы. В ООП данные и предназначенный для их обработки код "находят друг друга" на уровне классов/объектов, а уже из объектов формируется структура программы. В отличие от ООП, в структурном программировании данные и код для их обработки существуют сами по себе и собираются вместе непосредственно в программе. Для нас сравнение ООП со структурным программированием является важным, потому что именно ООП пришло на смену структурному программированию. Причина такой радикальной перемены связана в первую очередь с тем, что при написании больших программ подходы в рамках парадигмы структурного программирования на практике малопригодны для использования: большие массивы данных приходится связывать с большим объемом программного кода.

Принято считать, что ООП проявилось в результате естественного развития *процедурного программирования* (с использованием процедур и функций, то есть подпрограмм). Первым языком программирования, в котором появились классы и объекты, был язык *Simula*. Язык появился в конце 60-х прошлого века и, по мнению многих специалистов в области программирования, опередил время. В языке Simula была предложена встроенная поддержка базовых механизмов ООП. Язык Simula в силу разных причин не прижился, а вот объектно-ориентированный подход оставил след в умах разработчиков. В качестве примера следующего после Simula языка, поддерживающего ООП, можно привести язык Smalltalk. И эта цепочка не оборвалась. Как результат — ООП процветает и поныне. Доказательство тому - язык программирования Java.

Обычно, когда обсуждают ООП, вспоминают три базовых механизма, реализуемых в рамках объектно-ориентированного подхода. Это *инкапсуляция*, *полиморфизм* и *наследование*.

- Под **инкапсуляцией** подразумевают механизм, который позволяет объединить в объекте данные и код для обработки данных, скрыв и защитив при этом от пользователя внутреннюю реализацию объекта. Взаимодействие пользователя с объектом ограничивается предопределенным набором действий и правил. Эти правила, вместе с механизмом их реализации, формируют *интерфейс* объекта или его *спецификацию*. В языке Java инкапсуляция реализуется на уровне концепции классов и объектов.

- Благодаря **полиморфизму** объекты с одинаковой спецификацией/интерфейсом могут иметь разное содержимое или реализацию. Девиз полиморфизма можно сформулировать так: "Один интерфейс – много реализаций!". Главным практическим проявлением полиморфизма является возможность создавать методы с одинаковыми именами или даже сигнатурами (тип результата, имя метода, список аргументов), но выполняющихся по-разному. В Java полиморфизм реализуется путем *переопределения* и *перегрузки* методов (смысл этих терминов описан в книге) с учетом *наследования* классов.

- Такой механизм, как **наследование**, позволяет создавать новые классы на основе уже существующих классов. При этом новый класс автоматически получает (или *наследует)* свойства исходного (или родительского) класса. Согласно принятой в Java терминологии, родительский класс называется *суперклассом*, а созданный на его основе класс – *подклассом*. Кроме унаследованных из суперкласса свойств, в подклассе могут добавляться новые свойства. Благодаря наследованию, формируется своеобразная структура или иерархия классов. В вершине иерархии находятся классы наиболее общего характера, а дальше по цепочке наследования располагаются классы более узкого "профиля". При этом класс-наследник реализует спецификацию родительского класса, в результате чего методы работы с объектами производного класса аналогичны методам работы с объектами родительского класса.

Справедливости ради следует отметить, что далеко не всегда объектно-ориентированный подход базируется на использовании классов. В некоторых случаях вместо классов применяются *прототипы*. В этом случае роль шаблона, на основе которого создается объект, играет объект-прототип. В рамках данного подхода новый объект создается путем модификации существующего объекта или его клонированием. Поскольку к Java все это не имеет никакого отношения, останавливаться на этом и не будем.

Язык программирования Java является полностью объектно-ориентированным. Это означает, что даже самая маленькая программа в Java реализуется в рамках концепции ООП и содержит, по крайней мере, один класс. В этом отношении все, с чем мы будем иметь дело в книге, – это ООП. С другой стороны, базовые синтаксические конструкции языка имеют достаточно универсальный характер. И именно с их изучения начинается в книге путешествие в мир Java. Поэтому особо заострять внимание на особенностях объектно-ориентрованного подхода в начальных главах книги мы не будем. Способ создания единственного необходимого класса в начальных примерах книги фактически постулируется. Тем не менее, это не меняет сути вопроса – ООП оно и есть ООП. От того, что мы о нем постоянно не говорим, не значит, что мы его не используем. Практическое знакомство с ООП в Java начинается с *главы 6*, посвященной созданию классов и объектов. Методы и подходы ООП в Java реализуются в полной мере. Все они нашли отражение в книге.

Полезные ресурсы

Кроме книг по Java и сопутствующим продуктам, полезную информацию читатель сможет почерпнуть через Интернет. Сайт разработчика уже упоминался, повторяться не будем. Некоторые (точнее, совсем некоторые) из полезных интернет-ресурсов приведены в табл. 1.

Табл. 1. Некоторые полезные интернет-ресурсы в области Java-технологий

Адрес	Описание
http://developers.sun.ru	Российский портал разработчиков Sun Microsystems
http://www.javaworld.com	Электронный журнал, посвященный Java-технологии
http://www.juga.ru	Сайт разработчиков и потребителей Java
http://www.jars.com	Служба информационной поддержки Java
http://www.javable.com	Сайт, посвященный Java
http://www.ibm.com/developerworks/ru/java/	Сайт компании IBM, посвященный Java-технологии
http://java.dzone.com	Сайт сообщества разработчиков Java
http://www.microsoft.com/java	Сайт компании Microsoft с информацией об использовании Java
http://www.erudite-center.com	Центр информационной поддержки Java-технологий
http://www.embarcadero.com/products/jbuilder	Сайт компании-разработчика интегрированной среды JBuilder

Разумеется, приведенным списком (а он достаточно субъективный) возможности по получению полезной и бесплатной информации в области Java-технологий не ограничиваются. Нужно всегда искать новые источники — они постоянно появляются!

Структура книги

Книга состоит из *Вступления*, *15*-ти *глав* и *Заключения*. Для удобства восприятия материала каждая глава заканчивается кратким *резюме*. Первая *глава* является скорее иллюстративной, поскольку в ней содержится несколько простых примеров программы, написанных на языке Java. С одной стороны, это позволяет читателю уже с первых строк окунуться в мир реального программирования и стать соавтором хотя и простой, но все же полноценной программы на Java, а с другой стороны — не перегружает раньше времени специфической терминологией ООП и дотошным анализом синтаксических конструкций. Сами синтаксические конструкции и методы их построения описываются, шаг за шагом, в *главах* со *2*-й по *5*-ю включительно. А именно, в *главе 2* характеризуются основные, базовые типы Java а также методы использования переменных. *Глава 3* посвящена базовым операторам (арифметическим, логическим, побитовым и операторам сравнения). Условные операторы и операторы цикла описываются в *главе 4*. Наконец, в *главе 5* читатель познакомится с массивами — в Java они особенные!

Как отмечалось выше, фактически свое знакомство с ООП читатель начнет с самой первой главы, но подозрения у него появятся только где-то к *главе 6*. Именно начиная с этой главы обсуждаются вопросы, связанные с основными механизмами ООП — *инкапсуляцией*, *полиморфизмом* и *наследованием*. *Главы 6* и *7* посвящены непосредственно классам и объектам и методам работы с ними. В *главе 8* обсуждаются интерфейсы и наследование. В *главе 9* описываются методы работы с текстом и ряд других утилит. *Глава 10* посвящена обработке исключительных ситуаций. В *главе 11* даются основы многопоточного программирования в Java, а *глава 12* содержит полезную информацию о системе ввода/вывода в Java, в том числе и консольного.

Три последние главы книги посвящены созданию приложений с графическим интерфейсом. При этом за основу в книге принята библиотека Swing. В *главе 13* представлены основные сведения по этой библиотеке и приведены простые примеры приложений с графическим интерфейсом. Расширить свой кругозор по этому вопросу читатель сможет с помощью примеров из *главы 14*. Наконец, *глава 15* позволит читателю составить представление об апплетах. В этой, последней, главе, решается двойная задача. Первая непосредственно связана с апплетами, а вторая — проиллюстрировать методы работы с некоторыми классами библиотеки Swing.

Хотя "основательное" (в разумных пределах) описание библиотеки Swing начинается с *главы 13*, с методами создания приложений с графическим интерфейсом читатель познакомится уже в первой главе. Там для вывода информации создаются специальные диалоговые окна, реализуемые на основе одного из классов библиотеки Swing.

Терминология и принятые договоренности

Хотя книга предназначена для всех, кто интересуется программированием в Java, адаптирована она в первую очередь в расчете на тех, кто делает в программировании первые шаги. Отсюда и некоторая специфика, связанная со стилем изложения, способом и последовательностью подачи материала. Некоторые вещи новичкам могут показаться необычными и непривычными.

Что касается базовой терминологии — то в книге она стандартная, общепринятая в сообществе Java-разработчиков. Есть, правда, некий "тонкий" момент, связанный с тем, что базовая терминология — англоязычная. Поэтому здесь мы имеем дело с переводом тех или иных терминов на русский язык. А перевод не всегда однозначен. Важное значение также имеет, прижился тот или иной термин в русском варианте перевода или нет. Примером может быть ситуация с термином "*поток*". Он обычно используется и при описании многопоточного программирования, и для обозначения потоков данных в системе ввода/вывода. В тексте книги по этому поводу даются пояснения. Во всем остальном особых проблем с терминологией не наблюдается.

Коды всех примеров, рассматриваемых в книге, полностью приведены в тексте. Так что читатель получает доступ к готовым программам. Все программы рабочие — в том смысле, что работают. По традиции имена методов в тексте книги приводятся с пустыми круглыми скобками. Все остальные специфические обозначения поясняются непосредственно в тексте книги в том месте, где они появляются.

> 🕮 **На заметку:**
>
> Некоторые наиболее важные места в книге или те моменты, на которые стоит обратить внимание, выделены в специальные блоки.

Обратная связь

Связаться с автором можно по адресу электронной почты `alex@vasilev.kiev.ua`. По этому адресу можно сообщить свои замечания по поводу книги или пожелания на будущее. Принимаются и вопросы, но автор вынужден честно предупредить, что физической возможности ответить на *все* во-

просы у него нет. Зато не исключено, что полезную информацию читатель сможет для себя найти на сайте `www.vasilev.kiev.ua`. Как говорится, милости просим.

Благодарности

Автор выражает искреннюю благодарность редакторам издательства "Наука и Техника" и лично *Марку Финкову*, а в его лице и всему издательству — за эффективную работу и плодотворное сотрудничество.

Глава 1
Первая программа

Как вы яхту назовете, так она и поплывет!
(Из м/ф "Приключения капитана Врунгеля")

Хорошее дело лучше начинать с хорошего примера. Изучение языка программирования Java начнем с простой программы, которая выводит текстовое сообщение. Куда она его выводит – это вопрос отдельный.

Очень простая программа

Запускаем приложение NetBeans. В окне приложения выбираем команду **File ▶ New Project**, как показано на рис. 1.1.

С таким же успехом можно воспользоваться комбинацией клавиш <Ctrl>+<Shift>+<N> или специальной пиктограммой на панели инструментов приложения NetBeans (рис. 1.2).

Откроется окно с названием **New Project**, в котором следует выбрать тип создаваемого приложения (рис. 1.3). В зависимости от версии и "комплектации" среды разработки NetBeans окно может выглядеть по-разному, но идея остается неизменной: необходимо указать язык разработки (если соответствующая версия NetBeans допускает использование нескольких языков) и непосредственно тип приложения. В данном случае выбираем в категории языков (список **Categories** в левой части окна) пункт **Java** (что соответствует языку Java), а в правой части окна в списке **Projects** выбираем пункт **Java Application**, и с чувством выполненного долга щелкаем кнопку **Next** (см. рис. 1.3).

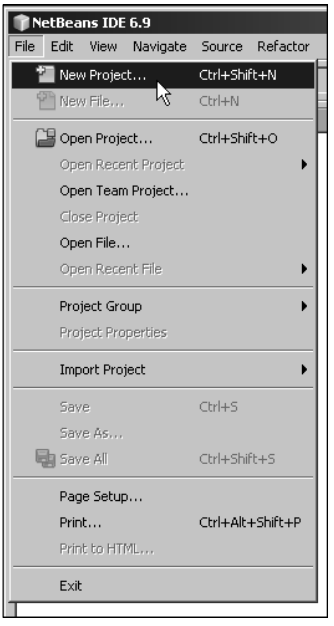


Рис. 1.1. Создание нового проекта в NetBeans

Рис. 1.2. Еще один способ создания нового проекта в NetBeans

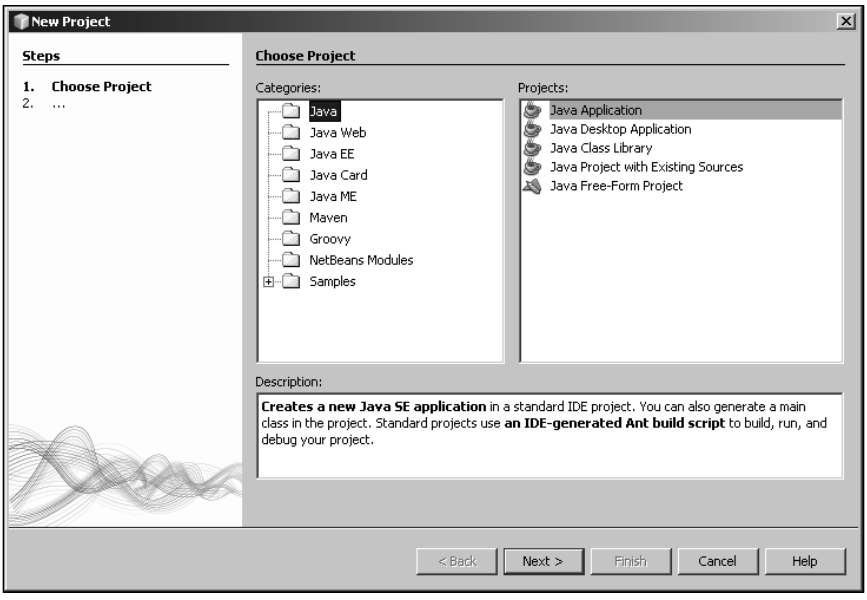


Рис. 1.3. Выбор типа создаваемого приложения

Надежды на то, что мучения закончились, исчезают, когда появляется следующее окно с интригующим названием **New Java Application** (рис. 1.4).

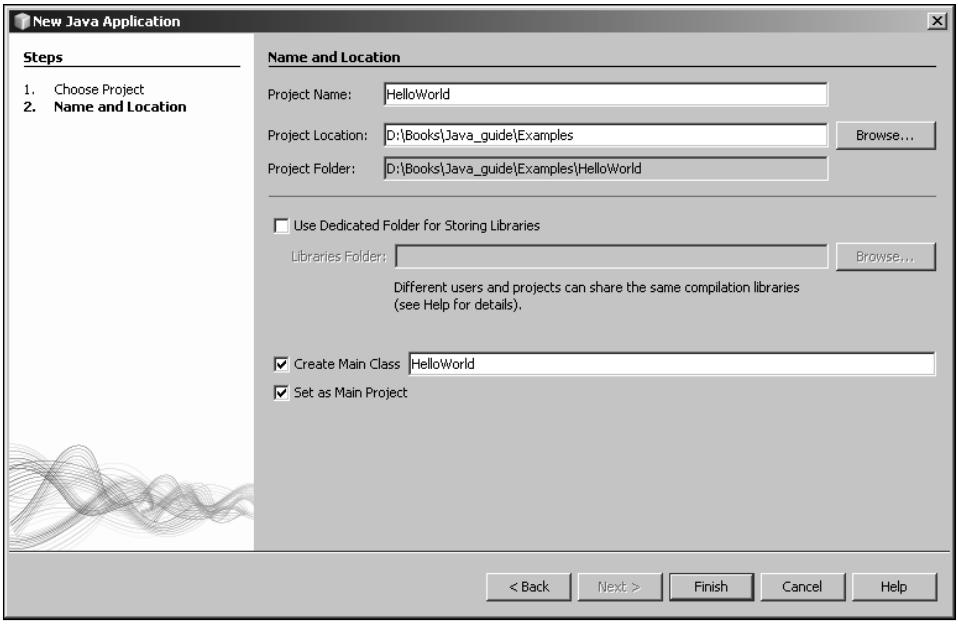


Рис. 1.4. Название и место расположения нового проекта

Это чудесное окно содержит несколько полей, в которых придется что-то писать. В первую очередь в поле **Project name** вводим название для создаваемого проекта. По возможности оно должно быть кратким и информативным. Но если приходится выбирать между краткостью и информативностью, последняя все же предпочтительнее. В данном случае все просто. Под давлением силы традиции первую программу (или проект) называем `HelloWorld`. Не менее важно место, где будет сохранен проект. Оно указывается в поле **Project Location**. Вместо того, чтобы вводить путь к соответствующей папке, разумнее и проще воспользоваться кнопкой **Browse**, размещенной справа от поля. Флажки опций **Create Main Class** и **Set as Main Project** оставляем установленными. В поле возле опции **Create Main Class** дублируем название проекта `HelloWorld`. Вообще в этом поле вводится название для главного класса программы. Но поскольку мы пока не знаем, что такое класс, а тем более главный класс программы, то просто дублируем имя проекта. Венцом проделанной работы становится щелчок на кнопке **Finish**.

Создается проект с шаблонным программным кодом – то есть, чтобы упростить жизнь программисту, автоматически генерируется начальный код, в который затем вносится правка. Окно редактора NetBeans с этим шаблонным кодом нового созданного проекта показано на рис. 1.5.

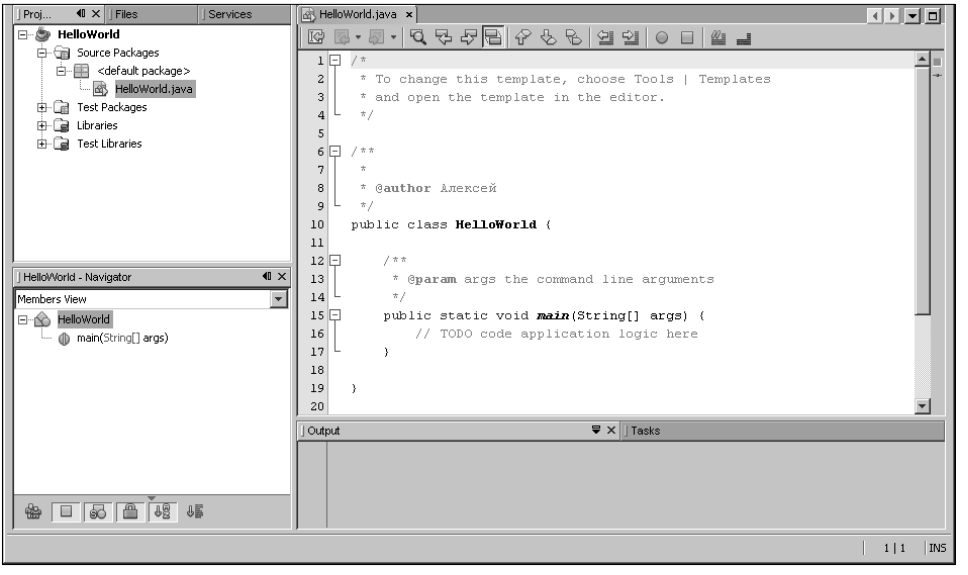


Рис. 1.5. Программный код шаблона для нового проекта

При создании однотипных проектов шаблон бывает полезен. Причем шаблон изменяется, его можно настроить по собственному усмотрению. Но это уже вопрос работы со средой NetBeans, и он не входит в круг вопросов,

освещаемых в книге. Желающие, конечно, могут покопаться в настройках NetBeans, но на данном этапе это не главное. Главное – написать работающую программу. Плюс не последнее значение имеет возможность поупражняться в наборе программных кодов. Поэтому разумно код шаблона удалить, а вместо него ввести другой, представленный в листинге 1.1.

Листинг 1.1. Простая программа

```
import javax.swing.*;
public class HelloWorld{
   public static void main(String[] args){
   JOptionPane.showMessageDialog(null, "Всем большой привет!");
   }
}
```

Окно редактора NetBeans с кодом программы показано на рис. 1.6.

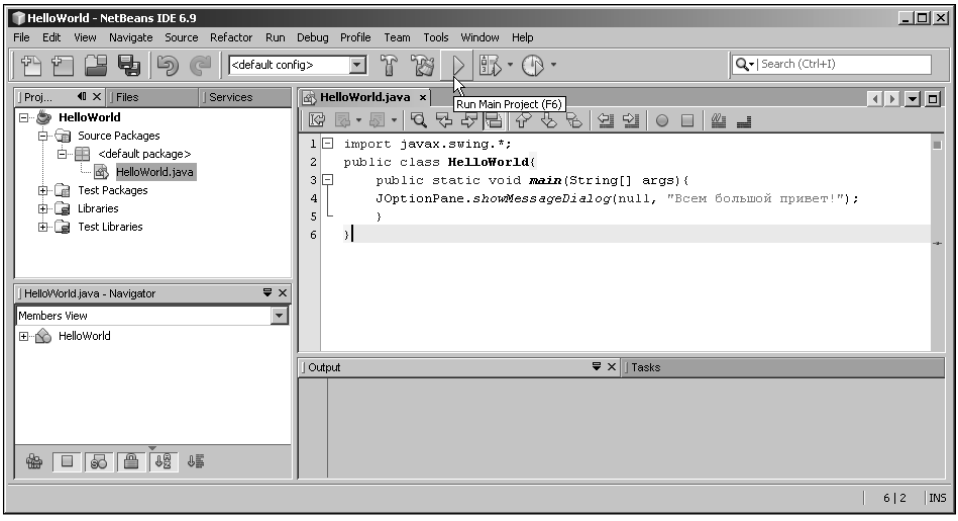


Рис. 1.6. Программный код проекта в окне редактора NetBeans и запуск проекта на выполнение

Назначение команд кода описывается далее. А на нынешнем этапе проверим, как код работает. Для выполнения этой почетной миссии на панели инструментов NetBeans находим пиктограмму с изображением зеленой стрелки и смело щелкаем ее. В результате появляется диалоговое окно, показанное на рис. 1.7.

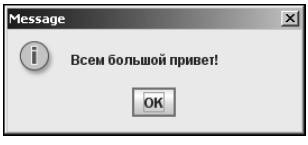


Рис. 1.7. Результат выполнения программы

Это стандартное окно с текстовым сообщением "Всем большой привет!". И вот теперь самое время разобраться, откуда это окно взялось и почему оно имеет именно такой вид, а не какой-то другой. Обратимся к программному коду в листинге 1.1. Но прежде сделаем несколько общих замечаний.

Код достаточно простой, и главная идея состоит в том, чтобы воспользоваться уже готовой библиотекой для того, чтобы вывести стандартное диалоговое окно. Особенность или уникальность этого окна связана лишь с тем текстом, который в нем отображается. Этот текст (фраза "Всем большой привет!") задается в программе. Сама программа состоит из инструкции подключения библиотеки графических компонентов и класса (который называется HelloWorld). Класс, в свою очередь, состоит из всего одного метода (который называется main()). Это особенный метод. Он называется *главным методом программы*. По большому счету выполнение программы означает выполнение этого метода. В методе, в свою очередь, всего одна команда, которой на экране отображается диалоговое окно.

Итак, первой командой программы является инструкция import javax.swing.*, которая заканчивается точкой с запятой (; - так заканчиваются все команды в Java) и состоит из двух частей: ключевого слова import и инструкции javax.swing.*. Ключевое слово import является инструкцией подключения классов и пакетов. Фактически наличие этой инструкции означает, что подключается какой-то ресурс, или, нормальным языком, библиотека. Просто в Java библиотеки реализуются через *классы*, объединенные в *пакеты*. Какой именно ресурс подключается, позволяет определить вторая часть команды. В данном случае диалоговое окно, которое отображается при выполнении программы, описано в библиотеке графических компонентов, которая называется Swing. Это название присутствует в инструкции javax.swing.*. В такого рода инструкциях указывается полный путь к подключаемым средствам. Путь отражает структуру пакетов Java (пакет может содержать классы, *интерфейсы* и другие пакеты). В качестве разделителя при указании структуры пакетов используется точка, а звездочка означает все классы в пакете. Поэтому инструкция import javax.swing.* означает не что иное, как подключение библиотеки Swing. На данном этапе достаточно осознать этот простой факт.

Вся остальная часть программного кода представляет собой описание класса HelloWorld – основного и единственного класса программы. Начинается описание класса инструкцией public class HelloWorld с последующей открывающей фигурной скобкой {. Чтобы у читателя не зародились смутные сомнения в каком-то подвохе, для порядка опишем назначение отдельных ключевых слов в этой и следующих инструкциях. Если поначалу

описание покажется сложным — не страшно. Важно помнить, что на самом деле все намного проще.

Ключевое слово `public` означает, что класс доступен за пределами того пакета, в котором он описан. Здесь, строго говоря, можно было бы обойтись и без этого ключевого слова, но обычно главный класс программы описывается с ключевым словом `public`. Ключевое слово `class` свидетельствует о том, что речь идет о классе, а не о чем-то другом. После ключевого слова `class` указывается название класса – в данном случае класс называется `HelloWorld`. Далее описывается программный код класса, определяющий его свойства в широком смысле этого слова. В Java блоки программного кода выделяются фигурными скобками.

Открывающая фигурная скобка `{` означает начало программного блока, закрывающая фигурная скобка `}` означает окончание программного блока. Скобки используются только парами. Внутри одного программного блока могут быть и другие программные блоки. Первый, внешний блок, выделенный фигурными скобками, является кодом класса. Внутри этого блока описан метод `main()`. Инструкция `public static void main(String[] args)` называется *сигнатурой* метода, и каждое ключевое слово в ней имеет определенное назначение. Слово `public` означает, что метод доступен за пределами класса. В данном случае это важно. Как отмечалось, выполнение программы означает выполнение метода `main()`. Очевидно, что метод должен вызываться извне, ведь не может программа запускать сама себя. Именно поэтому метод объявляется как открытый, то есть с ключевым словом `public`. Следующее ключевое слово `static` означает, что метод *статический*. То, что метод статический, означает, в свою очередь, что его можно вызывать без создания экземпляра класса, то есть объекта. Стандартная схема вызова обычного, нестатического метода, состоит в том, что на основе класса создается объект, и из объекта вызывается метод. С методом `main()` такой номер не пройдет в принципе. Ведь для того, чтобы создать объект класса `HelloWorld`, необходимо сначала запустить программу на выполнение. Это означает, что нужно выполнить метод `main()`. А если метод нестатический, то для его выполнения необходимо создать экземпляр класса, и так далее – круг замыкается. Если вся эта чехарда со статическими методами сейчас не очень понятна – не страшно. Достаточно запомнить, что метод `main()` должен быть статическим.

Ключевое слово `void` означает, что метод не возвращает результат. То есть метод что-то делает и завершает работу. Все вычисления происходят только внутри метода. После завершения метода от всех вычисленных значений в методе не остается и следа. Такой метод разумно было бы назвать процедурой, но в Java данный термин не употребляется. Существуют методы,

которые возвращают значения. В этом случае при выполнении метода выполняются команды и вычисляются значения, и одно из них не удаляется по завершении работы метода. Это значение называется возвращаемым результатом. О возврате результата методом имеет смысл говорить, если один метод вызывается внутри другого. Поскольку метод `main()` отождествляется с программой, то ни о каком внешнем для него методе речь идти не может, и возвращать результат незачем, да и не для кого.

Слово `main` означает название метода. Вообще методы могут называться самыми чудными именами. Но если речь не идет об *апплетах*, то в программе всегда должен быть один и только один главный метод, который называется `main()`.

На заметку:

Согласно общепринятым правилам названия методов для удобства указываются с пустыми круглыми скобками.

У метода могут быть *параметры*, или *аргументы*. Это те значения, которые передаются методом и которые обычно используются при выполнении метода. Аргументы при описании метода перечисляются в круглых скобках через запятую. Для каждого аргумента, кроме формального имени, указывается еще и тип. У метода `main()` один аргумент, который называется `args` (но здесь допускается и другое имя). Этот аргумент текстовый, о чем свидетельствует ключевое слово `String`. Наличие пустых квадратных скобок `[]` после ключевого слова `String` свидетельствует о том, что аргумент метода `main()` не просто текст, а текстовый массив, то есть набор текстовых значений. Фигурная скобка `{` означает начало программного блока метода `main()`. У нее есть напарница – фигурная скобка `}`, которая закрывает блок.

Специально для тех, кто уже отчаялся, предназначен следующий утешительный приз: в этой и нескольких следующих главах все (или почти все) программы соответствуют следующему шаблону:

```
import javax.swing.*;
public class HelloWorld{
   public static void main(String[] args){
   // здесь основной код
   }
}
```

Запомним его и, не мудрствуя лукаво, просто будем использовать. Все интересующие нас команды размещаются в том месте шаблона, который обозначен комментарием *здесь основной код*. Кстати, двойная косая черта в Java означает комментарий – все, что слева от двойной косой черты, при компиляции кода игнорируется.

На заметку:

Кроме однострочного комментария, выделяемого двойной косой чертой, существуют еще два способа комментирования. Первый подразумевает использование набора символов `/*` и `*/`. Все, что находится между этими инструкциями, считается комментарием. Можно также использовать открывающую инструкцию комментария `/**` и закрывающую инструкцию `*/`.

Остался финальный аккорд – команда `JOptionPane.showMessageDialog(null, "Всем большой привет!")`. В этой команде ссылка `JOptionPane` есть не что иное, как класс библиотеки Swing. Класс `JOptionPane` отвечает за работу с диалоговыми окнами. В частности с его помощью можно создавать и отображать диалоговые окна четырех типов:

- Диалоговые окна для вывода информационных сообщений.
- Диалоговые окна получения подтверждения на определенные действия от пользователя.
- Диалоговые окна получения (ввода) данных от пользователя.
- Диалоговые окна для выполнения настроек.

Именно для того, чтобы можно было использовать класс `JOptionPane` со всеми его утилитами, в начале программы и размещалась команда `import javax.swing.*`. Если бы мы не использовали класс `JOptionPane`, без указанной инструкции можно было обойтись.

У класса `JOptionPane` имеется статический метод `showMessageDialog()`, которым отображается диалоговое окно информационного типа, то есть окно для вывода информационного сообщения. Если вызывается статический метод класса, то правило его вызова такое: сначала указывается имя класса, и через точку имя метода с аргументами (если нужно). Метод `showMessageDialog()` допускает разный способ передачи аргументов. В данном случае методу передается два аргумента. Первым аргументом указано ключевое слово `null`. Вообще первый аргумент определяет, какое окно порождает отображаемое на экране диалоговое окно с сообщением. Ключевое слово `null` означает, что такого окна нет – диалоговое окно порождается не окном, а вызывается из метода `main()`. Второй аргумент метода `showMessageDialog()` представляет собой текст сообщения в окне. Текст заключается в двойные кавычки. На этом, собственно, все!

Сообщение можно выводить не в диалоговое окно, а на консоль. Выглядит все это не так эффектно, как в случае с диалоговым окном, зато более естественно. По сравнению с предыдущим примером в код вносятся минимальные изменения. Во-первых, избавляемся от инструкции `import javax.`

swing.* подключения библиотеки Swing - ведь диалоговое окно использоваться не будет. Во-вторых, вместо команды JOptionPane.showMessageDialog(null, "Всем большой привет!") используем команду System.out.println("Всем большой привет!"). Здесь для вывода текста на консоль используется метод println(). Выводимый на консоль текст указывается аргументом метода. Ключевое слово System – это имя класса. Класс доступен автоматически и имеет поле out, которое связано с консолью.

На заметку:

Более подробно класс System и ряд других вопросов, которые связаны с вводом/выводом данных, обсуждаются в *главе 12*.

Короче говоря, рецепт простой: если нужно вывести сообщение на консоль, используем команду с шаблоном System.out.println(), в которой аргументом метода println() указывается то, что выводится на консоль (обычно это текст, хотя и не факт). В листинге 1.2 представлен код программы, которой выводится сообщение на консоль.

Листинг 1.2. Еще одна простая программа

```
public class HelloWorldAgain{
   public static void main(String[] args){
   System.out.println("Всем большой привет!");
   }
}
```

Окно среды разработки NetBeans с программным кодом показано на рис. 1.8.

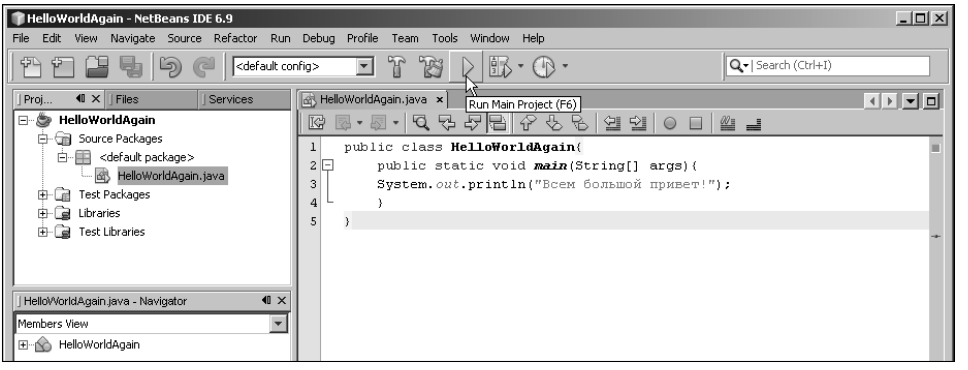


Рис. 1.8. Программа выводит сообщение на консоль

В результате выполнения программы сообщение выводится в нижнем подокне приложения NetBeans с названием **Output** (рис. 1.9).

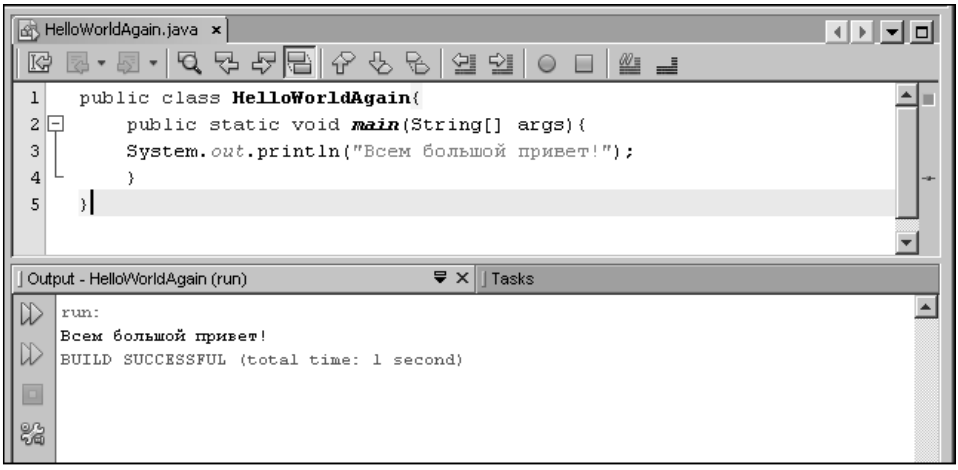


Рис. 1.9. Результат выполнения программы

Конечно, не так эффектно, как с диалоговым окном, но тоже вполне приемлемо. Далее мы будем пользоваться, с переменным успехом, для вывода данных как диалоговыми окнами, так и консолью.

На заметку:

Для того чтобы закрыть проект в окне интегрированной среды NetBeans, в подокне **Projects** (по умолчанию окно в левом верхнем углу окна приложения) в контекстном меню выбираем команду **Close**.

Еще одна простая программа

Если в первых двух примерах текстовые сообщения выводились через диалоговое окно или на консоль, то в следующем примере информация (текст) будет еще и считываться.

Алгоритм программы достаточно прост: программой выводится диалоговое окно с приветствием и просьбой указать свое имя. Для ввода имени в диалоговом окне есть специальное поле. После того, как пользователь вводит имя в поле диалогового окна, выводится новое окно с приветствием. В этом новом окне приветствия содержится и имя пользователя. Вот такая простая программа.

Несложно сообразить, что в новой программе появляется диалоговое окно нового типа – окно с полем для ввода информации. Как и в предыдущем случае, поможет нам класс `JOptionPane`. Это значит, что придется подключать библиотеку Swing. Рассмотрим программный код, представленный в листинге 1.3.

Листинг 1.3. Вопрос-ответ

```
import javax.swing.*;
public class WhatIsYourName{
   public static void main(String[] args){
   String name;
   name=JOptionPane.showInputDialog("Добрый день! Как Вас зовут?");
   JOptionPane.showMessageDialog(null, "Приятно познакомиться,\n"+name+"!");
   }
}
```

Изменилась начинка метода `main()`. Он теперь содержит три команды. Первая команда `String name` объявляет текстовую *переменную* с названием `name`. Переменная – это именованная ячейка памяти, к которой в программном коде обращаются через имя. В программном коде переменная впервые появляется при *объявлении*. Для этого указывают тип переменной и ее имя. Объявлять переменную можно практически в любом месте программного кода метода, а использовать только после объявления (и присваивания значения).

На заметку:

Переменная напоминает банковскую ячейку. В нее можно что-то положить, а потом это что-то оттуда забрать. Точнее, посмотреть, что там есть, или положить туда что-то другое взамен уже существующего. Но сначала ячейку нужно открыть. Роль номера счета или номера ячейки играет имя переменной. Далее, ячейки могут быть разного размера. Размер ячейки определяет ее назначение. Точно так же, при создании переменной важно знать, какого типа данные предполагается в ней хранить.

О том, что переменная `name` является текстовой, свидетельствует ключевое слово `String`. Строго говоря, `String` – это класс, через который реализуются текстовые значения. Но здесь нам достаточно знать, что инструкция `String name` означает текстовую переменную с именем `name`.

Мало объявить переменную. Необходимо ей присвоить значение. Значения присваиваются с помощью оператора, который, как ни странно, называется *оператором присваивания*. В Java оператор присваивания обозначается знаком равенства (то есть =). Переменная, которой присваивается значение, указывается слева от оператора присваивания. Значение, которое присваивается, указывается справа от оператора присваивания.

В переменную `name` записывается имя пользователя, которое вводится в диалоговое окно при запросе. Таким образом, необходимо отобразить окно, считать введенное пользователем значение (текст) и записать это значение в переменную `name`. Все три задачи решаются одной командой: `name=JOptionPane.showInputDialog("Добрый день!`

Как Вас зовут?"). Слева от оператора присваивания указана переменная name. Следовательно, этой переменной присваивается то, что указано справа от оператора присваивания. А справа указана инструкция, основу которой составляет метод showInputDialog() класса JOptionPane. Метод статический, поэтому при его вызове сначала указывается имя класса. Аргумент у метода один, и это тот текст, который отображается в диалоговом окне. Это окно отображается методом showInputDialog(), и оно отличается от окна, которое отображается методом showMessageDialog(), рассматривавшимся ранее. Окно, которое выводится на экран методом showInputDialog(), имеет поле для ввода текста. Текст, который пользователь вводит в поле, возвращается методом в качестве результата. Все это выглядит так: пользователь в поле диалогового окна вводит текст и щелкает кнопку подтверждения. Окно после этого закрывается, но текст запоминается. Этот текст может быть присвоен какой-нибудь переменой. В данном случае значение (текст) записывается в переменную name.

Наконец, командой JOptionPane.showMessageDialog(null, "Приятно познакомиться,\n"+name+"!") отображается информационное окно. Такого типа команда уже рассматривалась выше. Особенность этой команды связана со вторым текстовым аргументом метода showMessageDialog(). Он представляет собой *объединение* трех текстовых фрагментов. Для объединения используется оператор сложения +. Если складываются текстовые фрагменты, то происходит их последовательное объединение. В данном случае к тексту "Приятно познакомиться,\n" добавляется значение текстовой переменной name и еще восклицательный знак (текст "!"). В первом текстовом фрагменте использована инструкция

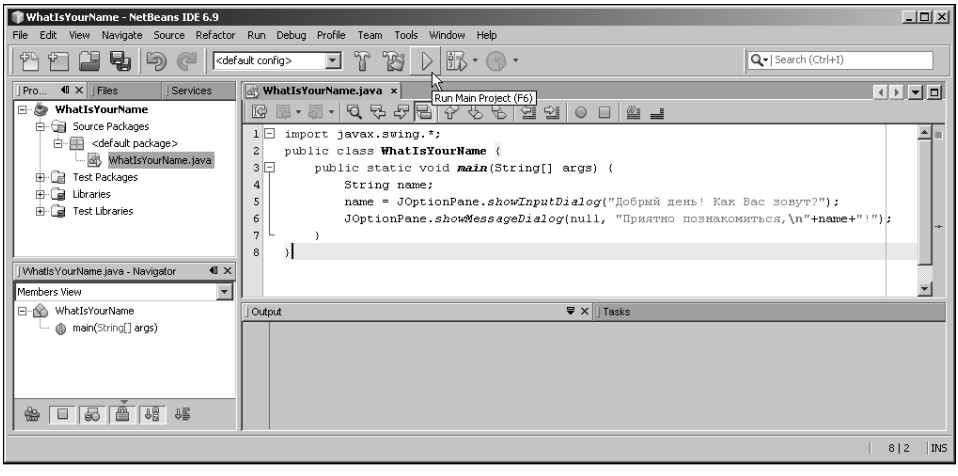


Рис. 1.10. Окно среды NetBeans с программным кодом

\n. Она означает переход к новой строке. Поэтому полученное в результате объединения трех текстовых фрагментов сообщение будет отображаться в диалоговом окне в две строки.

На рис. 1.10 показано диалоговое окно среды NetBeans с кодом программы.

После запуска программы на выполнение появляется диалоговое окно, как на рис. 1.11.

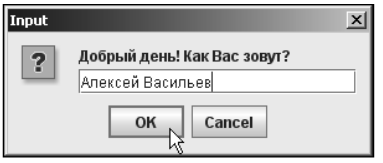


Рис. 1.11. Первое диалоговое окно

Окно содержит поле ввода, над которым отображается текст `Добрый день! Как Вас зовут?`. Если в поле ввести текст и щелкнуть кнопку **OK**, появляется еще одно окно, представленное на рис. 1.12.

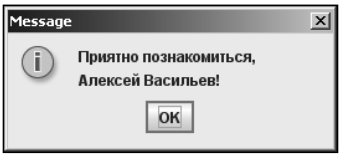


Рис. 1.12. Второе диалоговое окно

Как и предполагалось, текст в диалоговом окне отображается в две строки. На этом мы заканчиваем рассмотрение простых программ.

Резюме

Таким образом, можем подвести некоторые скромные итоги.

1. Любая программа на Java содержит хотя бы один класс. В этом хотя бы одном классе должен быть метод с названием `main()`, который называется главным методом программы. Выполнение программы отождествляется с выполнением этого метода.
2. Если для ввода или вывода информации используются диалоговые окна, необходимо подключить библиотеку Swing, для чего в начале программы размещается `import`-инструкция. В этой библиотеке есть класс `JOptionPane`, через который реализуются окна для ввода и вывода сообщений.
3. Программы, которые мы в ближайшее время будем писать, имеют следующий шаблон (его следует запомнить):

```
import javax.swing.*;
public class HelloWorld{
   public static void main(String[] args){
   // здесь основной код
   }
}
```

4. В программе (пока что программа для нас - метод `main()` и содержащий его класс) могут и обычно используются переменные. Переменная – это ячейка памяти, которая имеет имя и в которую можно записывать значения и считывать значение из нее. Обращение к переменной в программе осуществляется по имени.

5. Перед использованием переменной в программе ее нужно объявить. При объявлении переменной указывается ее *тип* и имя. Мы пока знаем только один тип переменной – текстовый `String`.

6. Переменной присваивается значение. Оператором присваивания в Java служит знак равенства =. Переменная, которой присваивается значение, указывается слева от оператора присваивания. Присваиваемое значение указывается справа от оператора присваивания.

Глава 2
Переменные и основные типы данных

Это не ерунда, это совсем даже не ерунда...
Особенно в таком деле!
(Из к/ф "Семнадцать мгновений весны")

С переменными (точнее, одной переменной) и операторами (присваивания и сложения) мы уже имели дело в предыдущей главе. Там же было показано, как переменная текстового типа объявляется и используется в программе. Но в Java, кроме текстового, много других типов переменных. В этой главе мы более детально рассмотрим эти типы, а также основные операции (и соответствующие им операторы), которые применимы по отношению к переменным разных типов.

Переменные базовых типов

Повторимся: **переменная представляет собой ячейку памяти с именем. По этому имени к ячейке можно обращаться в программе. Две базовых операции, которые выполняются с ячейками, – это** *запись* **в ячейку значения и** *считывание* **из ячейки значения**. **Для работы с переменной ее нужно создать.** То есть в памяти необходимо как-то пометить то место, в которое будут заноситься значения и из которого будут считываться значения. В зависимости от того, какие данные предполагается хранить в ячейке, должен выбираться размер ячейки. С другой стороны, в Java данные строго типизированы. Поэтому неудивительно, что при создании переменной (то есть при выделении ячейки памяти), кроме имени переменной, указывается еще и ее *тип*.

Выделяют *базовые* типы переменных (их еще называют *простыми* или *примитивными*) и переменные *классовых* типов (переменные типа класса - их еще называют *ссылочными* переменными).

📖 На заметку:

Наличие примитивных или базовых типов в Java – дань традиции. Была идея вообще от них отказаться и все типы реализовать через классы. Компромиссом стало наличие для каждого из базовых типов класса-оболочки. Мы эти классы-оболочки будем использовать в основном при преобразовании значений: текстовых представлений чисел в числа.

Кроме текста, который, кстати, к базовым типам не относится, в Java существует несколько типов данных для чисел, символьный тип и логический. Рассмотрим их.

Числовые значения делятся на целочисленные и действительные. В Java четыре целочисленных типа: `byte`, `short`, `int` и `long`. Разница между этими типами с формальной точки зрения лишь в диапазоне значений, который поддерживает тот или иной тип. Выше целочисленные типы перечислены в порядке увеличения "мощности": самый маленький диапазон значений поддерживается типом `byte`, а самый большой – типом `long`. Числовые значения для типа `byte` лежат в диапазоне от `-128` до `127`. Значения типа `short` могут принимать значения от `-32768` до `32767`. Типом `int` поддерживаются значения в диапазоне от `-2147483648` до `2147483647`. Наконец, типом `long` поддерживаются значения в диапазоне от `-9223372036854775808` до `9223372036854775807`. На практике использование того или иного типа определяется потребностями в системных ресурсах. Тем не менее, среди перечисленных четырех типов тип `int` имеет особое значение, и использовать мы будем именно его. Причина заключается в правилах автоматического приведения типов, о которых речь пойдет несколько позже. Пока же запомним, что если в переменную предполагается заносить (присваивать в качестве значения) целые числа, то переменная эта будет иметь один из упомянутых четырех типов и, скорее всего, это тип `int`.

Для работы с действительными числами предназначены два типа: `float` и `double`. Как и в случае с целочисленными типами, разница между типами `float` и `double` заключается в ширине диапазона хранения данных. Для типа `double` памяти для хранения данных (в битах) выделяется в два раза больше, поэтому и диапазон шире. Переменная типа `float` позволяет работать с числами со значениями (по модулю) до $3{,}4\times10^{38}$. Чувствительность переменной (минимальный шаг дискретности) составляет величину порядка $3{,}4\times10^{-38}$. Точность значений составляет при этом до 8 цифр в представлении числа. В то же время максимальное значение и минимальный шаг дискретности для переменной типа `double` составляют соответственно $1{,}7\times10^{308}$ и $1{,}7\times10^{-308}$. Точность в представлении числа обеспечивается на уровне 17 цифр. Совершенно очевидно, что тип `double` намного предпочтительней. При работе с действительными числами мы будем использовать тип `double`. Причина не столько в том, что этот тип позволяет хранить числа и производить вычисления с большей точностью, по сравнению с типом `float`, но скорее связана с теми же правилами автоматического преобразования типов, на которые мы ссылались выше.

В некотором смысле близок к числовым типам и **символьный тип** `char`. Значениями переменных этого типа являются буквы (или символы). Бли-

зок тип char к целочисленным типам потому, что технически символы реализуются в виде целочисленных кодов кодовой таблицы. Диапазон значений лежит в пределах от 0 до 65535.

Еще один базовый тип данных – **логический**, который в Java называется boolean. Переменные типа boolean принимают всего два значения: true (*истина*) и false (*ложь*). Несмотря на такую кажущуюся примитивность, тип boolean в Java играет достаточно важную роль, а переменные этого типа (или выражения, возвращающие значение логического типа) используются очень часто. Дело в том, что в Java, в отличие, например, от языка C++, в условных операторах используются, увы, только логические значения.

Как отмечалось, кроме переменных базовых типов, существуют переменные классовых, или ссылочных типов. Эти переменные в качестве значений содержат ссылки на объекты (экземпляры) классов. С такой переменной мы уже сталкивались в предыдущей главе (напомним, это была тестовая переменная, в которую записывалось имя пользователя). Методы работы с ссылочными переменными рассмотрим после того, как поближе познакомимся с классами и объектами.

Как объявлять переменные

С процедурой объявления переменной мы имели дело в первой главе. Здесь лишь обобщим приобретенные навыки. Итак, для объявления переменной (пока речь идет только о переменных базовых типов) необходимо указать тип переменной (ключевое слово, обозначающее тип) и имя переменной. Если объявляется несколько переменных одного типа, их имена можно перечислить через запятую, указав идентификатор типа лишь один раз. Например:

```
int number;
char symbol;
double x,y;
```

В данном случае объявляется целочисленная переменная number типа int, символьная переменная symbol (переменная типа char) и две переменные x и y типа double.

Объявление переменной означает, что для нее в памяти выделяется место. При этом объем выделяемой памяти определяется типом переменной. Например, для переменной типа double это 8 байт (64 бита). Переменная типа float получает в свое распоряжение 4 байта (32 бита). Под переменные целочисленных типов byte, short, int и long выделяется соответственно 1, 2, 4 и 8 байтов (8, 16, 32 и 64 бита). Переменная типа char занимает в памяти 2 байта (16 бит).

Что касается места объявления переменной, то здесь демократия полная. Правда, важное значение имеет категория переменной: это может быть локальная (внутренняя) переменная метода или поле класса. Мы до этого имели дело только с внутренними переменными метода `main()`. С ними же будем иметь дело и в ближайшем будущем. Поэтому пока речь идет только о таких переменных – они объявляются внутри метода (в пределах программного блока из фигурных скобок, которые выделяют тело метода). **В пределах метода переменная может объявляться где угодно – главное, чтобы до того, как она начинает использоваться.** Более того, даже если переменная объявлена, она еще не получает значения. Чтобы использовать переменную в каком-то выражении, предварительно ей необходимо присвоить значение. **Первое присваивание значения переменной называется** *инициализацией*. В Java инициализацию переменной можно совмещать с ее объявлением. В предыдущей главе мы узнали, что оператором присваивания в Java служит знак равенства =. Чтобы присвоить переменной значение, после ее имени указывают знак равенства (оператор присваивания) и значение, которое присваивается переменной. Например:

```
int number;
number=12;
char symbol='a';
double x=3.4,y;
```

Целочисленной переменной `number` после объявления командой `number=12` присваивается значение `12`. Символьная переменная `symbol` инициализируется со значением `'a'` при объявлении. Обратите внимание, что значение символьной переменной указывается в одинарных кавычках. Напомним, для текста мы использовали двойные кавычки.

При объявлении нескольких однотипных переменных часть из них или даже все могут инициализироваться. Примером тому служит последняя команда из приведенного выше кода, в которой объявляется две переменные `x` и `y` типа `double`, причем переменной `x` сразу при объявлении присваивается значение `3.4`.

В листинге 2.1 приведен программный код, в котором использовано несколько переменных.

Листинг 2.1. Использование переменных

```
import javax.swing.*;
public class UsingVar{
   public static void main(String[] args){
   String name="Иванов Иван Иванович";
   int age=60;
```

```
   char category='B';
   String text="Водитель: "+name;
   text=text+"\n"+"Возраст: "+age+" лет";
   text=text+'\n'+"Права категории: "+category;
   JOptionPane.showMessageDialog(null,text);
   }
}
```

В результате выполнения программы отображается диалоговое окно с информацией о некоем водителе: указывается его полное имя, возраст и категория водительских прав (рис. 2.1).

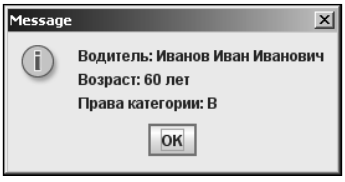


Рис. 2.1. Результат выполнения программы из листинга 2.1

Исследуем, почему это диалоговое окно выглядит именно так, а не как-нибудь иначе. Думается, что недюжинные усилия, приложенные для создания антуража кода (шапка с импортом библиотеки графических утилит заголовки класса и главного метода программы), можно оставить без внимания – там ничего нового нет. Интерес представляет то, что внутри метода `main()`. А именно объявления нескольких переменных, формирование отображаемого в диалоговом окне текста и команда вывода диалогового окна. Теперь по порядку.

Командой `String name="Иванов Иван Иванович"` создается текстовая переменная `name` со значением `"Иванов Иван Иванович"`. Это просто: в переменную записывается имя нашего виртуального водителя. Еще проще дела обстоят с целочисленной переменной `age`. Она объявляется (и инициализируется) командой `int age=60` и содержит, таким образом, в качестве значения возраст водителя. Категория водительских прав – это всего одна буква. Для ее хранения можно использовать символьную переменную, что и было сделано. Соответствующая переменная называется `category`, имеет тип `char` и получает значение `'B'`. Все это реализуется командой `char category='B'`.

Текстовая (тип переменной `String`) переменная `text` нужна для того, чтобы записать в нее тот текст, который выводится в диалоговом окне. Значение переменной `text` определяется в несколько этапов. Сначала командой `String text="Водитель: "+name` эта переменная объявляется и инициализируется со значением, которое получается объединением текста `"Водитель: "` и текстового значения переменной `name`. Обращаем вни-

мание, что в данном случае при объявлении и инициализации переменной text ее значение определяется выражением, в которое входит другая переменная (текстовая переменная name). Такая ситуация (когда при объявлении переменной инициализация выполняется выражением, содержащим другие переменные) называется *динамическим объявлением переменной*. При динамическом объявлении переменной важно, чтобы другие используемые при инициализации переменные были предварительно объявлены и инициализированы. В данном случае это условие выполнено: ранее переменной name было присвоено текстовое значение "Иванов Иван Иванович". Поэтому в результате переменная text получает значение "Водитель: Иванов Иван Иванович". Здесь при объединении текста в первой фразе "Водитель: " специально в конце добавлен пробел, чтобы при слиянии текстов пробел был между соседними словами.

На заметку:

Не забывайте, что пробел, с точки зрения синтаксиса языка Java, такой же символ, как и все прочие буквы, и пренебрегать им не стоит.

Значение, которое получила переменная text, не является ее конечным значением. Туда нужно еще кое-что дописать. Строго говоря, это *кое-что* можно было вписать сразу, но поскольку *кое-что* достаточно длинное, сразу мы его не вписывали, а растянули удовольствие на несколько команд. В частности, командой text=text+"\n"+"Возраст: "+age+" лет" к текущему значению переменной text дописывается символ перехода к новой строке "\n", фраза "Возраст: ", затем значение переменной age (хотя это целочисленная переменная, ее значение автоматически преобразуется к текстовому представлению) и, наконец, фраза " лет". Таким образом, переменная text содержит уже две строки из трех, которые выводятся программой в диалоговом окне. Стоит заметить, что, хотя инструкция перехода к новой строке "\n" реализована в данном случае в виде текста и состоит, формально, из двух символов (косой черты \ и буквы n), инструкция эта все равно считается одним символом и поэтому вместо двойных кавычек можно использовать и одинарные, как в следующей команде text=text+'\n'+"Права категории: "+category, которой к тексту в переменной text добавляется еще одна инструкция перехода к новой строке, фраза "Права категории: " и значение переменной category. После этого переменная text содержит в качестве значения тот текст (с учетом построчной разбивки), который требуется отобразить в диалоговом окне. Само окно отображается командой JOptionPane.showMessageDialog(null,text). Вторым аргументом в метод showMessageDialog() передана текстовая переменная text.

В рассмотренном примере несколько раз использовались команды, в которых к тексту, кроме другого текста, прибавлялись значения типа int

и char. В таких случаях имеет место *автоматическое приведение типов*, которое описывается в следующей главе. Суть его состоит в том, что операнды разных типов приводятся к одному типу. В данном случае все значения преобразуются в текст.

Классы-оболочки

Как отмечалось ранее, для базовых типов данных существуют так называемые *классы-оболочки*. Классы-оболочки используют в тех случаях, когда необходимо реализовать числовое, логическое или символьное значение в виде объекта. Достаточно часто такая необходимость связана с преобразованием значений. Простой и с практической точки зрения важный пример – когда текстовое представление числа необходимо преобразовать в числовое значение. Рассмотрим следующую ситуацию. Имеется целочисленная переменная Number со значением 321 и текстовая переменная NotANumber, со значением "123":

```
int Number=321;
String NotANumber="123";
```

С переменной Number все достаточно просто: переменная как переменная. Хитрее обстоят дела с переменной NotANumber. Хотя вроде бы понятно, о каком значении (числовом) идет речь, переменная NotANumber является текстовой, и только текстовой. То, что мы догадались о числе, которое "спрятано" в эту переменную – ничего не значит с точки зрения синтаксиса языка Java. Чтобы воплотить нашу догадку в жизнь, необходимо "вытащить" числовое значение из текста. Вот здесь как раз и пригодится класс-оболочка для целых чисел. Поскольку о классах мы знаем пока не очень много, в тонкости ситуации вдаваться не будем. Освоим лишь несколько практических приемов – они нам понадобятся в дальнейшем.

Класс-оболочка для значений типа int называется Integer. У класса есть статический метод parseInt(), который в качестве результата возвращает целое число (значение типа int), "спрятанное" в текст, как и в нашем случае. Этот текст передается аргументом методу parseInt(). Другими словами, чтобы извлечь число из переменной NotANumber, используем инструкцию вида Integer.parseInt(NotANumber). Пример использования класса-оболочки Integer и его метода parseInt() приведен в листинге 2.2.

Листинг 2.2. Использование класса-оболочки

```
import javax.swing.*;
public class UsingInteger{
   public static void main(String[] args){
   int Number=321;
   String NotANumber="123";
```

```
    int ANumber=Integer.parseInt(NotANumber);
    String title1="Команда Number+NotANumber";
    String title2="Команда Number+ANumber";
    int type1=JOptionPane.ERROR_MESSAGE;
    int type2=JOptionPane.WARNING_MESSAGE;
    JOptionPane.showMessageDialog(null,Number+NotANumber,title1,type1);
    JOptionPane.showMessageDialog(null,Number+ANumber,title2,type2);
    }
}
```

Первые две команды `int Number=321` и `String NotANumber="123"` нам уже знакомы и особых комментариев не требуют. Более интересной представляется команда `int ANumber=Integer.parseInt(NotANumber)`. Здесь, во-первых, объявляется целочисленная переменная `ANumber`. Этой переменной также присваивается значение. Для этого текстовое представление числа из переменной `NotANumber` "извлекается" и записывается в переменную `ANumber`. Следовательно, значением переменной `ANumber` будет число `123`. Весь остальной код носит декоративный характер и служит исключительно цели достижения максимальной наглядности и эффектности. А именно, в данном примере мы несколько освежим знакомые нам уже диалоговые окна, изменив их заголовки (отображаются в полосе названия окна) и пиктограммы в области окна.

В программе последовательно отображаются два диалоговых окна. Текстовые названия для этих окон определяются командами `String title1="Команда Number+NotANumber"` и `String title2="Команда Number+ANumber"`. Также надо будет определить тип диалогового окна. Для этого в классе `JOptionPane` есть набор статических целочисленных полей-констант. Одну из этих констант надо будет указать аргументом метода `showMessageDialog()`. **Обращение к статическим полям класса осуществляется практически так же, как к статическим методам – указывается имя класса и через точку имя поля.** Синтаксис обращения в таком формате достаточно громоздкий. Поэтому для удобства определим две числовые переменные `type1` и `type2`, в которые запишем значения нужных статических полей. Реализуется все это через команды `int type1=JOptionPane.ERROR_MESSAGE` и `int type2=JOptionPane.WARNING_MESSAGE`.

На заметку:

Константа от переменной отличается тем, что значение константы изменить нельзя. Объявляется константа так же, как переменная, только с ключевым словом `final`. Инициализируются константы одновременно с объявлением. Например, командой `final double PI=3.141592` объявляется константа `PI` типа `double` со значением `3.141592`. Хотя такого жесткого правила нет, но в Java *принято* (но не обязательно) названия констант писать из прописных букв.

Значение ERROR_MESSAGE используется для отображения окна из серии *"произошла ошибка"*. Значение WARNING_MESSAGE используется для отображения *"окна предупреждения"*. На практике разница между окнами разных видов проявляется в различии пиктограмм, отображаемых в области окна.

Командой `JOptionPane.showMessageDialog(null,Number+NotANumber, title1,type1)` отображается окно, показанное на рис. 2.2.

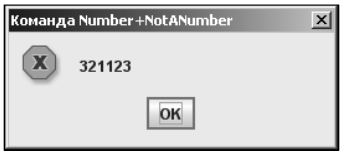


Рис. 2.2. Результат сложения числа и текста

В данном случае метод `showMessageDialog()` вызывается с четырьмя аргументами. Назначение первых двух нам знакомо. Третий аргумент определяет заголовок диалогового окна, а четвертый – его стиль (пиктограмма в рабочей области окна).

Интерес представляет значение, которое отображается в области окна. Значение вычисляется командой `Number+NotANumber`. Поскольку в данном случае вычисляется сумма целого числа и текста, то число автоматически преобразуется в текст и происходит объединение текстовых значений. Отсюда результат: `321123`.

Несколько иная ситуация при вычислении результата выражения `Number+ANumber`. Поскольку теперь складываются два числовых значения (`123` и `321`), то получаем результат `444`. В результате выполнения команды `JOptionPane.showMessageDialog(null,Number+ANumber, title2,type2)` открывается окно, показанное на рис. 2.3.

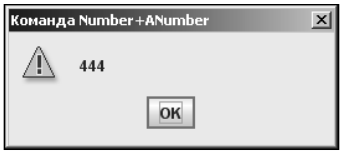


Рис. 2.3. Результат сложения числа и числа

Классы-оболочки существуют для всех базовых типов. В табл. 2.1 показано соответствие между базовыми типами и классами-оболочками.

За несколькими исключениями названия классов-оболочек совпадают (с точностью до первой заглавной буквы) с названиями базовых типов. Полезные утилиты, реализуемые через классы-оболочки, будем изучать по мере необходимости. Но и сейчас несложно догадаться, что у класса

Double, например, есть метод parseDouble() для преобразования текстового представления действительного числа в значение типа double.

Табл. 2.1. Классы-оболочки для базовых типов

Базовый тип	Класс-оболочка
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Резюме

1. В Java переменные имеют тип. Тип переменной определяет, какие данные могут записываться в переменную, какой объем памяти выделяется под переменную и какие операции могут выполняться с переменной.
2. В Java существует несколько базовых, или простых, типов данных. Это целочисленные типы (byte, short, int и long), действительные числовые типы (float и double), символьный тип char (значение – буква или символ) и логический тип boolean (два значения – true и false).
3. При объявлении переменной указывается тип ее значения и имя. Одновременно с объявлением можно переменной присвоить значение (инициализировать переменную). Несколько переменных можно объявлять одновременно. Объявление и инициализация переменной производится в любом месте метода, но до ее первого использования.
4. Константа от переменной отличается тем, что значение константы после инициализации (которая выполняется вместе с объявлением) изменить нельзя. Константы описываются так же, как переменные, но для них указывается еще ключевое слово final.
5. Для базовых типов существуют классы-оболочки: Byte, Short, Integer, Long, Float, Double, Character и Boolean соответственно для типов byte, short, int, long, float, double, char и boolean. Класс-оболочка предназначен для реализации того или иного значения в виде объекта. У классов-оболочек, кроме прочего, имеется ряд полезных методов. Например, у класса Integer есть статический метод parseInt(), с помощью которого текст, содержащий представление целого числа, преобразуется в число.

Глава 3
Базовые операторы и приведение типов

У меня есть мысль, и я ее думаю.
(Из м/ф "Тридцать восемь попугаев")

В этой главе речь пойдет об основных операторах Java. Их условно делят на четыре группы в зависимости от типа операндов и выполняемых над этими операндами действий. Это *арифметические* операторы, *логические* операторы, операторы *сравнения* и *побитовые* (поразрядные) операторы. Обсудим некоторые особенности оператора присваивания и алгоритмы автоматического *приведения типов*. В этой же главе кратко обсуждаются *литералы*. Вообще-то это тема предыдущей главы, но здесь ее уместнее обсудить в контексте процедуры приведения типов. Начнем с базовых операторов.

Арифметические операторы

К арифметическим относят операторы, предназначенные для работы с числовыми значениями и действие которых состоит в вычислении основных алгебраических операций, таких как сложение, вычитание, умножение и деление. Для этих операций операторы формально совпадают с соответствующими математическими обозначениями: `+` (*плюс*) для сложения, `-` (*минус*) для вычитания, `*` (*звездочка*) для умножения и `/` (*косая черта*) для деления. Думается, особых пояснений эти операции не требуют. Особенности имеет только *оператор деления*: если операндами являются целые числа, то деление выполняется нацело, то есть результат есть целая часть от деления одного целого числа на другое. Как обойти это ограничение в случае необходимости, будет рассказано далее.

Кроме перечисленных, в Java есть оператор `%` (*процент*), которым вычисляется остаток от деления одного числа на другое. Все эти операторы бинарные, то есть такие, у которых два операнда. Операторы инкремента `++` и декремента `--` являются унарными. У них один операнд. Действие оператора инкремента `++` на операнд состоит в том, что значение операнда увеличивается на единицу. Оператор инкремента `--` уменьшает на единицу значение операнда. Например, в результате выполнения команды `a++` значение переменной `a` увеличится на единицу. Чтобы уменьшить значение переменной `a` на единицу, используем команду `a--`. У операторов инкремента и декремента есть *префиксная* и *постфиксная* формы. В префиксной форме оператор указывается перед операндом (например, `++a` или `--a`). В пост-

фиксной форме оператор указывается после операнда (например, `a++` или `a--`). С точки зрения действия на операнд разницы в том, какую форму оператора использовать, нет. Различие между префиксной и постфиксной формами проявляется в случае, если инструкция с оператором инкремента или декремента сама является частью некоторого выражения. В этом случае для префиксной формы сначала вычисляется инкремент или декремент, а уже затем с учетом нового значения операнда вычисляется выражение. Для постфиксной формы сначала вычисляется выражение, а уже затем выполняется операция инкремента или декремента. Позднее эта особенность операторов инкремента и декремента иллюстрируется на примере.

📖 На заметку:

В Java нет оператора для операции *возведения в степень*. Для этих целей имеется специальная функция, но это далеко не то же, что оператор.

Важное значение при выполнении арифметических операций имеет приведение типов. Этому вопросу посвящен отдельный раздел главы. Кроме того, некоторые операции с использованием базовых операторов, в том числе и арифметических, могут быть записаны в упрощенном виде с использованием *составных операторов*. Составные операторы описываются в разделе, посвященном оператору присваивания.

Логические операторы

Логические операторы в качестве результата возвращают логические значения, то есть значения типа `boolean`. **Операнды у логических операторов также логические**. Логических операторов не много. Вот они: `&` (*амперсанд*) - оператор *логического И* (*конъюнкция*), `|` (*вертикальная черта*) - оператор *логического ИЛИ* (*дизъюнкция*), `^` (*каре*) – оператор *исключающего ИЛИ*, `!` (*восклицательный знак*) – логическое *отрицание*. Все операторы, кроме последнего (оператора отрицания), являются бинарными (оператор отрицания унарный). Результатом логических операций может быть значение `true` или `false`. Рассмотрим (кратко) каждую из операций.

Табл. 3.1 дает представление об операции *логическое И*. Она содержит значения результата операции `A&B` в зависимости от значений операндов `A` и `B`.

Табл. 3.1. Операция логическое И: `A&B`

A / B	**true**	**false**
true	true	false
false	false	false

В качестве значения выражения A&B возвращается значение true, только если оба операнда равны true. Если хотя бы один операнд равен false, результатом выражения A&B также является false.

Результатом выражения вида A|B является значение true, если хотя бы один из операндов равен true. Если оба операнда равны false, то результатом выражения A|B также является false (табл. 3.2).

Табл. 3.2. Операция логическое ИЛИ: A|B

A / B	**true**	**false**
true	true	true
false	true	false

В табл. 3.3 описан результат выполнения операции *исключающее ИЛИ*.

Табл. 3.3. Операция логическое исключающее ИЛИ: A^B

A / B	**true**	**false**
true	false	true
false	true	false

В качестве результата выражения A^B возвращается значение true, если один и только один из операндов равен true. В противном случае результатом является значение false.

Как отмечалось, операция логического отрицания является унарной. Результатом выражения вида !A является значение true, если значение операнда A равно false. Если значение операнда A равно true, то результатом операции !A является false.

Для операций *логического ИЛИ* и *логического И* есть сокращенные формы операторов – соответственно || (две вертикальных черты) и && (двойной амперсанд). В чем их особенность?

Несложно заметить, что если первый операнд при вычислении *логического ИЛИ* равен true, то второй операнд можно не вычислять – результат в любом случае будет true. Аналогично при вычислении операции *логического И,* если первый операнд равен false, то значение всего выражения также будет false независимо от значения второго операнда. Отличие между полными и сокращенными формами логических операторов состоит в способе вычисления результата. При вычислении выражений вида A|B и A&B определяются значения обоих операндов A и B. При вычислении выражений вида A||B и A&&B второй операнд B вычисляется, только если конечный результат невозможно определить по первому операнду A.

Операторы сравнения

Операторы сравнения все бинарные и применяются к числовым операндам. Результатом выражения с оператором сравнения является логическое значение: `true`, если соответствующее соотношение между значениями числовых операндов истинно, и `false` в противном случае. Всего в Java шесть операторов сравнения:

- **Оператор** *меньше* <. Результатом выражения `A<B` является `true`, если значение переменной `A` *меньше* значения переменной `B`. В противном случае значение выражения `false`.
- **Оператор** *меньше или равно* <=. Результатом выражения `A<=B` является `true`, если значение переменной `A` *не больше* значения переменной `B`. В противном случае значение выражения `false`.
- **Оператор** *больше* >. Результатом выражения `A>B` является `true`, если значение переменной `A` *больше* значения переменной `B`. В противном случае значение выражения `false`.
- **Оператор** *больше или равно* >=. Результатом выражения `A>=B` является `true`, если значение переменной `A` *не меньше* значения переменной `B`. В противном случае значение выражения `false`.
- **Оператор** *равно* ==. Результатом выражения `A==B` является `true`, если значение переменной `A` *равно* значению переменной `B`. В противном случае значение выражения `false`.
- **Оператор** *не равно* !=. Результатом выражения `A!=B` является `true`, если значение переменной `A` *не равно* значению переменной `B`. В противном случае значение выражения `false`.

Побитовые операторы

Побитовые операторы предназначены для выполнения операций с числами на уровне их побитового представления. Результатом операции с побитовыми операторами являются числа. Как известно, любое целое число может быть представлено в двоичном коде, когда позиционная запись числа состоит сплошь лишь из нулей и единиц. Побитовые операторы и по форме, и по сути очень напоминают логические операторы. Для понимания принципа, заложенного в основу вычисления побитовых операций, необходимо знать как минимум двоичный код чисел-операндов соответствующего выражения.

Например, действие оператора *побитового И* или *побитовой конъюнкции* `&` состоит в том, что в двоичном представлении операндов сравниваются со-

ответствующие биты по принципу *логического И*. Единичное значение бита отождествляется со значением `true`, а нулевое – со значением `false`. Для лучшего понимания в табл. 3.4 проиллюстрирован принцип действия этого оператора при сопоставлении двух битов.

Табл. 3.4. Операция побитового И: `a&b`

b \ a	**1**	**0**
1	1	0
0	0	0

В качестве иллюстрации рассмотрим выражение `9&5`, результатом которого является `1`. Поясним это. Учтем, что число `9` в двоичном представлении выглядит как `1001` (старшие нулевые биты можно проигнорировать). Число `5` в двоичном представлении можно записать (в представлении четырех битов) как `0101`. Легко проверить, что

$$\begin{array}{r} 1001 \\ \& \quad 0101 \\ \hline 0001 \end{array}$$

Двоичному представлению `0001` соответствует число `1`.

Побитовое ИЛИ (*побитовая дизъюнкция*) вычисляется оператором `|`. Идея та же, что и в предыдущем случае: операнды приводятся к двоичному представлению и для каждой пары битов выполняется сравнение по принципу *логического ИЛИ*. Табл. 3.5 содержит информацию относительно результата сравнения битов при помощи оператора `|`.

Табл. 3.5. Операция побитовое ИЛИ: `a|b`

b \ a	**1**	**0**
1	1	1
0	1	0

Рассмотрим выражение `9|5`. Имеем следующее:

$$\begin{array}{r} 1001 \\ | \quad 0101 \\ \hline 1101 \end{array}$$

Двоичному числу `1101` соответствует десятичное число `13`. Это и есть значение выражения `9|5`.

На заметку:

Если в двоичном представлении число имеет вид $a_n a_{n-1}...a_2 a_1 a_0$ (величины a_0, a_1,..., a_n могут принимать значение 0 или 1), то к десятичному виду это число приводится вычислением выражения $2^0 a_0 + 2^1 a_1 + 2^2 a_2 + ... + 2^{n-1} a_{n-1} + 2^n a_n$. Поэтому, например, $1101 = 2^0 \cdot 1 + 2^1 \cdot 0 + 2^2 \cdot 1 + 2^3 \cdot 1 = 13$.

Побитовое исключающее ИЛИ вычисляется оператором ^. Принцип - тот же, информация – в табл. 3.6.

Табл. 3.6. Операция побитовое исключающее ИЛИ: `a^b`

b \ a	1	0
1	0	1
0	1	0

Вычислим на всякий случай выражение `9^5`. Без особых проблем получаем такое:

$$\begin{array}{r} 1001 \\ \wedge\ 0101 \\ \hline 1100 \end{array}$$

Двоичное значение `1100` в десятичном представлении соответствует числу `12`.

Еще один побитовый оператор – оператор *дополнения* или *побитовой инверсии* ~ (тильда). Это унарный оператор. Его действие сводится к тому, что в побитовом представлении нуле меняются на единицы и единицы на нули. Эта операция имеет одно интересное свойство: результатом выражения `~a` является значение `-1-a`.

На заметку:

Для понимания этого чудесного факта необходимо учесть некоторые особенности представления отрицательных чисел в двоичном коде. В Java при представлении отрицательных чисел используется принцип *дополнения до нуля*. Главная идея состоит в том, что знак числа определяется самым старшим битом. Ноль соответствует положительному числу, а единица – отрицательному. До этого мы имели дело только с положительными числами. Если необходимо перевести из двоичной системы в десятичную отрицательное число, то алгоритм следующий. Во-первых, производится побитовая инверсия в двоичном представлении числа. Полученный результат переводим в десятичное представление (получится положительное число). Во-вторых, к полученному числу добавляется единица. В третьих, необходимо перед результатом добавить знак минус. Все!

Что касается приведенного выше алгоритма, то получить его несложно. Так, если мы сложим числа `~a` и `a`, то получим результат - число, в двоичном представлении которого все единицы. Другими словами, если переменная занимает, например, 32 бита, то все 32 бита заполнены единицами. Прибавим к этому числу `1`. Получим число из нулей и теоретически нового единичного старшего бита. Пикантность ситуации в том, что этот новый старший бит теряется, поскольку он должен занять 33-й бит, которого физически нет (а в таких случаях бит теряется). Следовательно, получаем число, состоящее в двоичном представлении из всех нулей. А это есть самый настоящий ноль! Таким образом, получили, что выражение `~a+a+1==0` истинно. А из него напрямую следует то, что число `~a` равняется `-1-a`, что и требовалось доказать.

Еще одна группа побитовых операторов (их три) позволяет выполнять сдвиги. Рассмотрим их. Это оператор сдвига влево <<, оператор сдвига вправо >> и оператор сдвига вправо без смещения старшего бита >>>. Все операторы бинарные.

Результатом вычисления выражения `a<<n` является число, которое получается сдвигом двоичного представления числа `a` на `n` позиций влево. Старшие биты теряются, младшие заполняются нулями. Фактически в данном случае идет речь об умножении числа `a` на `2` в степени `n` (так рассчитывается результат, само число `a` при этом не меняется). Аналогично вычисляется выражение `a>>n`, только сдвиг выполняется вправо. Биты слева заполняются значением старшего бита, а биты справа теряются. Отличие оператора >>> от оператора >> состоит в том, что биты слева заполняются нулями.

Тернарный оператор

В Java существует один единственный тернарный оператор, то есть оператор, у которого три операнда. Обозначается оператор как `?:`. Синтаксис вызова этого оператора с операндами `a`, `b` и `c` такой: `a?b:c`. Операнд `a` является логическим, то есть переменной типа `boolean` или выражением, которое возвращает значение типа `boolean`. Результат вычисляется следующим образом: если первый операнд равен `true`, то в качестве результата тернарного оператора возвращается значение второго операнда `b`. В противном случае (если первый операнд равен `false`) в качестве результата тернарного оператора возвращается значение третьего операнда `c`. Пример использования тернарного оператора приведен в листинге 3.1.

Листинг 3.1. Использование тернарного оператора

```
import javax.swing.*;
public class MakeChoice{
   public static void main(String[] args){
```

```
    int number=Integer.parseInt(JOptionPane.showInputDialog("Введите
                                                          число:"));
    String text=number%2==0?"четное":"нечетное";
    JOptionPane.showMessageDialog(null,"Вы ввели "+text+" число!");
    }
}
```

Программа выполняется так. Сначала отображается окно с полем ввода и просьбой указать число. Число считывается. Если оно четное, отображается окно с сообщением "Вы ввели четное число!". Если число нечетное, отображается сообщение "Вы ввели нечетное число!".

В программе командой `int number=Integer.parseInt(JOptionPane.showInputDialog("Введите число:"))` объявляется целочисленная переменная `number` и сразу инициализируется со значением, которое пользователь вводит в поле диалогового окна. Для отображения диалогового окна с полем ввода использована инструкция `JOptionPane.showInputDialog("Введите число:")`. Напомним, в качестве результата методом `showInputDialog()` возвращается введенный в поле пользователем *текст*. Мы предполагаем, что это будет текстовое представление целого числа – во всяком случае, об этом просили пользователя. Такое текстовое представление числа необходимо преобразовать в число. Для этого результат считывания содержимого поля ввода передаем аргументом методу `parseInt()`, который вызывается из класса-оболочки `Integer`.

Командой `String text=number%2==0?"четное":"нечетное"` объявляется текстовая переменная `text`. Ее значение определяется в зависимости от того, что ввел пользователь в поле ввода. Используем здесь тернарный оператор. В тернарном операторе проверяется условие `number%2==0` (*остаток от деления числа* `number` *на* `2` *равен нулю*). Для четных чисел данное условие выполняется, для нечетных – нет. В случае если условие выполнено, тернарным оператором возвращается текстовое значение `"четное"`. Если условие не выполнено, оператором возвращается текст `"нечетное"`. Наконец, командой `JOptionPane.showMessageDialog(null,"Вы ввели "+text+" число!")` отображается диалоговое окно с информацией о том, четное или нечетное число ввел пользователь.

Проверим работу программы. Запускаем. В отображаемом диалоговом окне в поле вводим нечетное число, как на рис. 3.1.

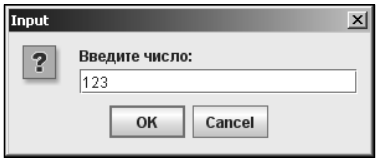


Рис. 3.1. В поле вводится нечетное число

После подтверждения ввода отображается еще одно диалоговое окно с сообщением о том, что веденное число нечетное (рис. 3.2).

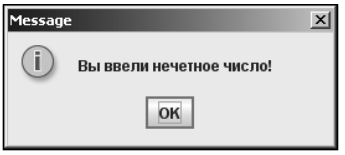


Рис. 3.2. Сообщение о том, что введено нечетное число

Снова запускаем программу. Но теперь в поле ввода указываем четное число (рис. 3.3).

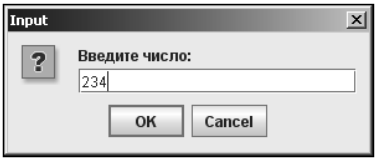


Рис. 3.3. В поле вводится нечетное число

Подтверждаем ввод и наблюдаем следующее окно с сообщением, что пользователь ввел четное число (рис. 3.4).

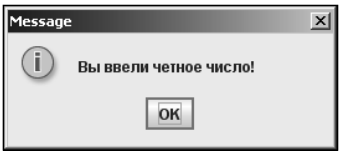


Рис. 3.4. Сообщение о том, что введено четное число

Фактически тернарный оператор представляет собой миниатюрный условный оператор. У полноценного условного оператора возможностей больше, да и синтаксис демократичнее. Убедимся в этом в следующей главе, которая посвящена *управляющим инструкциям* (к которым относится и условный оператор).

Оператор присваивания

Оператор присваивания мы уже использовали много и с большой пользой для программного кода. Однако тема далеко не исчерпана. Здесь сделаем некоторые уточнения.

Итак, напомним, что в качестве оператора присваивания в Java используется знак равенства =. Слева от оператора присваивания указывается переменная, которой присваивается значение. Справа от оператора присваивания указывается выражение. П**рисваивание выполняется по следующей**

схеме. Вычисляется выражение в правой части. Вычисленное значение записывается в переменную слева от оператора присваивания. Тип результата выражения должен быть совместимым с типом переменной в левой части оператора присваивания. Речь не идет о том, что типы должны совпадать. Они лишь должны быть *совместимы*. Совместимы типы или нет, несложно догадаться, даже не читая справочной информации. Обычно речь идет о числовых типах. В этом случае условие совместимости типов сводится к тому, чтобы числовой тип выражения справа от оператора присваивания был не "шире", чем тип переменной, которой присваивается значение. Другими словами, тип результата выражения должен быть таким, чтобы его можно было "втиснуть" в переменную слева от оператора присваивания без потери значения. Например, тип `short` совместим с типом `int`, а тип `int` совместим с типом `double`.

Обращаем внимание, что даже если переменная слева от оператора присваивания имеет значение, после присваивания оно теряется. То есть присваивание значения переменной означает, что новое значение "замещает" старое.

В Java в выражении может быть несколько операторов присваивания. Например, законной является команда вида `x=y=z=10`. В этом случае все три переменные `x`, `y` и `z` в качестве значения получат `10`. Дело в том, что оператор присваивания возвращает результат – это значение выражения в правой части от оператора присваивания. В данном случае сначала вычисляется выражение `z=10`. Результатом является значение `10`, и при этом переменная `z` получает такое же значение. Далее вычисляется выражение `y=z=10`, в котором значение присваивается переменной `y`, и так далее.

Явное приведение и автоматическое расширение типов

Достаточно часто в выражениях используются переменные разных типов. С точки зрения здравого смысла это вполне оправдано, поскольку, например, было бы странно, если нельзя было сложить значения типа `int` и `short`, ведь оба они являются целыми числами. Еще с одной аналогичной ситуацией мы встречались раньше, когда складывали текст и число. В этом случае получали текстовое значение. Возможны такие экзотические комбинации благодаря автоматическому *расширению* и *приведению* типов. Рассмотрим базовые принципы автоматического расширения и приведения числовых типов.

- Если в выражении имеется операнд типа `double`, другой операнд приводится к типу `double`.

- Если в выражении нет операндов типа `double`, но есть операнд типа `float`, то второй операнд также приводится к типу `float`.
- Если в выражении действительных операндов нет, но есть операнд типа `long`, то другой операнд также приводится к этому типу.
- Если в выражении операндов типа `long` нет, то операнды типа `byte`, `short` и `char` расширяются до типа `int`. Результат вычисления выражения также имеет тип `int`.

Последнее обстоятельство особенно важно и нередко является причиной серьезных недоразумений. Например, объявляются переменные `x` и `y` типа `byte`:

```
byte x=100, y=50;
```

Затем объявляем еще одну переменную `z` того же типа и инициализируем ее на основе значений переменных `x` и `y`:

```
byte z=x+y; // ОШИБКА!!!
```

Как бы это странно ни звучало, но последняя инструкция приводит к ошибке, то есть она неправильная. Причина в том, что при вычислении выражения `x+y`, несмотря на то, что оба операнда имеют тип `byte`, они автоматически расширяются до типа `int` и такого же типа будет результат выражения `x+y`. Поэтому ситуация компилятором воспринимается так: переменной типа `byte` присваивается значение типа `int`. А это криминал! Дело в том, что в Java присваивание с потерей значения запрещено, поэтому ситуация, подобная описанной выше, является ошибочной.

На заметку:

Все, что расположено справа от двойной косой черты (то есть справа от `//`) , является *комментарием* и компилятором игнорируется.

Из этой трагической ситуации есть достаточно простой, но очень неэстетичный выход – воспользоваться явным приведением типов. "Правильная" команда объявления и инициализации переменной `z` может быть такой:

```
byte z=(byte)(x+y); // ОШИБКИ НЕТ!!!
```

Здесь использована инструкция явного приведения типов – перед выражением в круглых скобках указывается тип, к которому необходимо привести результат выражения. Это насильственный акт. Он может привести к потере значения. Следить, чтобы ничего не потерялось, приходится самому. Выражение `x+y` заключено в круглые скобки для того, чтобы сначала вычислялась сумма, а уже затем результат этой операции приводился к типу `byte`.

Явное приведение типов позволяет решить и мини-проблему с оператором деления. Напомним, если операнды – оба целые числа, то деление выполняется нацело. Другими словами, результатом выражения `9/4` является значение `2`. Чтобы получить полагающиеся в таких случаях `2.25`, необходимо либо один из литералов задать в формате числа с плавающей точкой (например, использовать команду `9.0/4`), либо использовать команду `(double)9/4`.

Типы литералов

До этого мы как-то не обращали особого внимания на литералы. **Литералы – это числа, буквы, текст, который мы присваивали в качестве значений переменным.** Пикантность ситуации состоит в том, что литералы *имеют тип*. Если с текстом и буквами (символами) все более-менее ясно (там, как говорится, без вариантов), то с целыми и действительными числами интрига сохраняется. Поэтому сразу расставим все точки над "i": целочисленные литералы относятся к типу `int`. Сразу возникает вопрос: а как же эти литералы присваиваются переменным типа `byte` или `short`? Ответ прост: а просто так. Если переменной типа `byte` или `short` присваивается *литерал* (это важно!), и значение этого литерала не выходит за допустимые пределы типа переменной, то литерал автоматически урезается до нужного типа. Технически это выглядит так. Литерал записан в ячейку памяти из 32 бит. Если переменная занимает, например, 8 бит, то лишние 24 бита отбрасываются, но так, чтобы не было потери значения.

📖 На заметку:

Кроме десятичного представления, целочисленные литералы могут задаваться в восьмеричном и шестнадцатеричном представлении. В восьмеричном представлении запись литерала начинается с нуля. Например, литерал `011` означает десятичное число `9`. В шестнадцатеричном представлении запись литерала начинается с комбинации `0x` или `0X`. Например, литерал `0x11` означает десятичное число `17`.

В некоторых случаях может понадобиться, чтобы литерал относился не к типу `int`, а к типу `long`. Проблема решается добавлением литеры `L` (или `l`) в конец литерала. Например, `123L` является литералом (число `123`) типа `long`, то есть на этот литерал выделяется не 32, а 64 бита.

Далее, литералы-действительные числа относятся к типу `double`. В качестве десятичного разделителя используется точка. Также литералы для действительных чисел записывают в научной нотации, с мантиссой и показателем степени. Разделителем между мантиссой и показателем степе-

ни служит символ E или e. Например, литералы 123.4567, 1.234567E2 и 1234567e-4 означают одно и то же число.

На заметку:

Чтобы создать float-литерал, в конец литерала добавляется литера F или f. Например, 1.23F или 0.123E+2f.

Учитывая все сказанное и правила приведения (расширения) типов, можно дать одну рекомендацию: если нет крайней необходимости, целочисленные переменные объявляются с типом int, а действительные – с типом double. Это простое правило поможет в будущем избежать многих неприятностей.

Сокращенные формы операторов

В Java существуют так называемые *сокращенные формы операторов*. Речь идет об инструкциях вида x=x ☺ y, где x и y – переменные, а оператором ☺ является один из рассмотренных ранее бинарных арифметических и побитовых операторов. Такого вида инструкцию можно записать в сокращенной форме, а именно x ☺= y. Например, команду x=x+y можно записать как x+=y. Другими словами, команды x ☺= y и x=x ☺ y практически эквивалентны. Если точнее, то команда x ☺= y эквивалентна команде x=(x) (x ☺ y), то есть вычисляется результат выражения x ☺ y, результат приводится к типу переменной x и записывается в эту переменную. Эта на первый взгляд незначительная разница может приводить к некоторым проблемам при составлении программных кодов, особенно у начинающих программистов. Рассмотрим небольшой пример.

Определяем переменную a типа byte и присваиваем ей значение:

```
byte a=10;
```

Следующая команда является некорректной:

```
a=a+1;
```

Причина проста: при вычислении выражения a+1 результат автоматически приводится к типу int. Переменной типа byte значение типа int просто так не присвоить. Придется, как минимум, использовать явное приведение типов:

```
a=(byte)(a+1);
```

Нечто подобное уже описывалось выше. Обращаем внимание на наличие круглых скобок, в которые заключено выражение a+1. Если их не указать, то операция приведения типа будет применяться только к первому операнду в этом выражении.

Вместе с тем команда a+=1 ошибки не вызывает, хотя формально, как и в команде a=a+1, речь идет об увеличении значения переменной a на единицу. Именно здесь проявляется та особенность сокращенной формы оператора присваивания, о которой упоминалось выше. Не вызывают никаких проблем и команды вида a++, ++a, a-- и --a (то есть команды с операторами инкремента и декремента в префиксной и постфиксной формах). Уделим немного внимания этим замечательным операторам.

Инкремент и декремент

Разберем, в чем различия между префиксной и постфиксной формами операторов инкремента и декремента. Правила такие:

- На операнд действие оператора инкремента или декремента от формы (префиксной или постфиксной) не зависит. Инкремент в любой форме на единицу увеличивает значение операнда, декремент в любой форме – на единицу уменьшает.
- Если инструкция с оператором инкремента или декремента сама является частью некоторого выражения, то для префиксной формы сначала изменяется на единицу операнд, а затем вычисляется выражение. Для постфиксной формы сначала вычисляется выражение, а уже затем на единицу изменяется операнд. В обоих случаях (то есть для префиксной и постфиксной формы) операнд в конечном итоге изменяется одинаково.

Поясним сказанное на простом примере. Объявляем две целочисленные переменные:

```
int a=100,b;
b=2*++a;
```

После выполнения второй команды переменная b получает значение 202. Поскольку оператор инкремента использован в префиксной форме, то сначала на единицу увеличивается значение переменной a (новое значение 101), и уже это новое значение умножается на 2 (в результате получаем 202). Значение переменной a, напомним, равно 101.

Если вместо команды b=2*++a воспользоваться командой b=2*a++, переменная b получит значение 200. Теперь оператор инкремента указан в постфиксной форме. Это означает, что при вычислении выражения в правой части инструкции b=2*a++ в расчетах используется старое (неизмененное) значение переменной a. Именно оно умножается на два, и это значение записывается в переменную b. После этого значение переменной a увеличивается на единицу и становится равным 101, как и в предыдущем случае. Аналогичная ситуация и с оператором декремента.

Вычисление выражений и приоритет операций

Существует несколько правил, которые нужно иметь в виду при работе со сложными выражениями. Перечислим их.

- Каждый базовый оператор имеет свой *приоритет*. Приоритет определяет порядок выполнения операций в выражении. Операторы с более высоким приоритетом выполняются раньше. Разбивка операторов по приоритетам приведена далее.
- Для изменения порядка выполнения операторов используют круглые скобки.
- Если несколько операторов в выражении имеют одинаковый приоритет, то они выполняются слева направо.
- Операции присваивания выполняются справа налево.
- Операнды бинарных операторов вычисляются слева направо (сначала левый операнд, потом правый).
- При использовании сокращенных форм операторов для вычисления правой части выражения сохраняется копия текущего значения переменной в левой части.

Следующий список содержит разбивку базовых операторов по приоритетам (от наивысшего приоритета до низшего):

1. Наивысший приоритет имеют круглые скобки `( )`, квадратные скобки `[ ]` и оператор "точка" `.`.
2. Операторы инкремента `++`, декремента `--`, побитового `~` и логического отрицания `!`.
3. Оператор умножения `*`, деления `/` и вычисления остатка `%`.
4. Оператор сложения `+` и вычитания `-`.
5. Операторы побитовых сдвигов `>>`, `<<` и `>>>`.
6. Операторы сравнения больше `>`, больше или равно `>=`, меньше или равно `<=` и меньше `<`.
7. Операторы сравнения равно `==` и неравно `!=`.
8. Оператор `&`.
9. Оператор `^`.
10. Оператор `|`.
11. Оператор `&&`.

12. Оператор `||`.
13. Тернарный оператор `?:`.
14. Оператор присваивания `=`, и сокращенные формы операторов: `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, `>>=`, `>>>=`.

В заключение хочется дать один совет: команды должны быть наиболее простыми для понимания, а алгоритм их выполнения - предсказуемым. В случае необходимости, разбивайте сложные команды на несколько простых. Если есть сомнение в последовательности вычисления операторов, используйте круглые скобки. Также хорошим тоном считается использование круглых скобок. Даже если необходимости изменять естественный порядок вычисления операторов нет, наличие круглых скобок повышает читабельность программного кода выражений.

Резюме

1. В Java существует четыре группы операторов: арифметические, логические, операторы сравнения и побитовые операторы.
2. Оператор деления имеет особенность - если операнды целочисленные, деление выполняется нацело.
3. У операторов инкремента и декремента есть префиксная и постфиксная формы. Для префиксной формы сначала изменяется операнд, а затем вычисляется выражение, частью которого является операнд. Для постфиксной формы сначала вычисляется выражение, а затем изменяется операнд.
4. Тернарный оператор представляет собой упрощенную форму условного оператора.
5. В одном выражении может быть несколько операторов присваивания.
6. Существуют сокращенные формы операторов, которые позволяют оптимизировать синтаксис команды по изменению значения переменной.
7. При несовпадении типов переменных в выражении выполняется автоматическое приведение (расширение) типов. Приведение типов можно выполнять и в явном виде.
8. По умолчанию целочисленные литералы относятся к типу `int`, а действительные литералы относятся к типу `double`.
9. Значения выражений вычисляются с учетом приоритета операторов. Для изменения порядка вычисления выражения и повышения читабельности кода рекомендуется использовать круглые скобки.

Глава 4
Управляющие инструкции

Нормальные герои всегда идут в обход!
(Из к/ф "Айболит 66")

В этой главе речь пойдет об управляющих инструкциях: операторах цикла и условных операторах. Обсуждается также и ряд смежных вопросов.

Важно отметить, что операторы или инструкции, обсуждаемые далее, используются на практике достаточно часто и нередко составляют каркас программы. Поэтому важность обсуждаемых вопросов не стоит недооценивать.

Что касается предмета обсуждения, то условные операторы используются и предназначены для создания точек ветвления в программе, когда тот или иной код выполняется в зависимости от того, выполняется или нет некоторое условие. Операторы цикла нужны для многократного выполнения однотипных действий. Такие действия выполняются предопределенное количество раз или выполняются до выполнения некоторого условия. Рассмотрим основные управляющие инструкции Java подробнее.

Условный оператор `if`

Есть несколько вариантов или способов вызова условного оператора `if`. Начнем со стандартного. В этом случае используется два ключевых слова: `if` и `else`. Синтаксис вызова условного оператора такой:

```
if(условие){команды_1}
   else{команды_2}
```

После ключевого слова `if` в круглых скобках указывается *условие*. Это выражение, которое возвращает логический результат. Если значение *условия* равно `true` (условие выполнено), то выполняются команды, размещенные в блоке из фигурных скобок сразу после `if`-инструкции (*команды_1*). Если же значением условия является `false` (условие не выполнено), то выполняются команды в блоке после ключевого слова `else` (*команды_2*). Таким образом, условный оператор объявляется так:

- ключевое слово `if` с условием в круглых скобках;
- блок команд, которые выполняются при истинности условия;
- ключевое слово `else`;
- блок команд, которые выполняются, если условие ложно.

Принцип работы условного оператора проиллюстрирован на рис. 4.1.

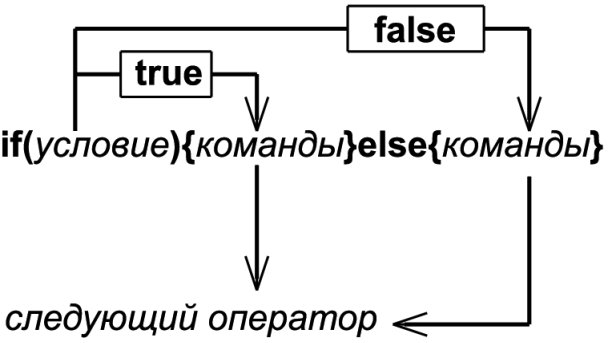


Рис. 4.1. Принцип работы условного оператора: стандартная форма

После выполнения оператора цикла управление передается следующему после него оператору.

На заметку:

Если в блоке после `if`-инструкции или `else`-инструкции всего одна команда, заключать ее в блок из фигурных скобок необязательно.

У условного оператора есть упрощенная форма. В этом случае `else`-блок не указывается вообще. Синтаксис вызова условного оператора в упрощенной форме такой:

```
if(условие){команды}
```

Если проверяемое условие истинно, выполняется блок команд в фигурных скобках после `if`-блока. Если условие не является истинным, управление передается оператору, следующему после условного. Работу условного оператора в упрощенной форме иллюстрирует рис. 4.2.

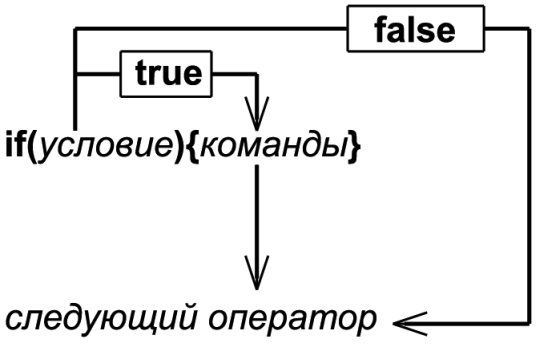


Рис. 4.2. Принцип работы условного оператора: упрощенная форма

Достаточно популярной является конструкция из вложенных условных операторов. Идея достаточно простая: в одном условном операторе вызывается другой условный оператор, в котором может вызываться третий условный оператор, и так далее. Синтаксис команды с таким иерархическим вызовом условных операторов следующий:

```
if(условие_1){команды_1}
   else if(условие_2){команды_2}
   else if(условие_3){команды_3}
   ...
   else if(условие_N){команды_N}
   else{команды_N+1}
```

На рис. 4.3 представлена схема, в соответствии с которой выполняется последовательность инструкций из вложенных операторов цикла.

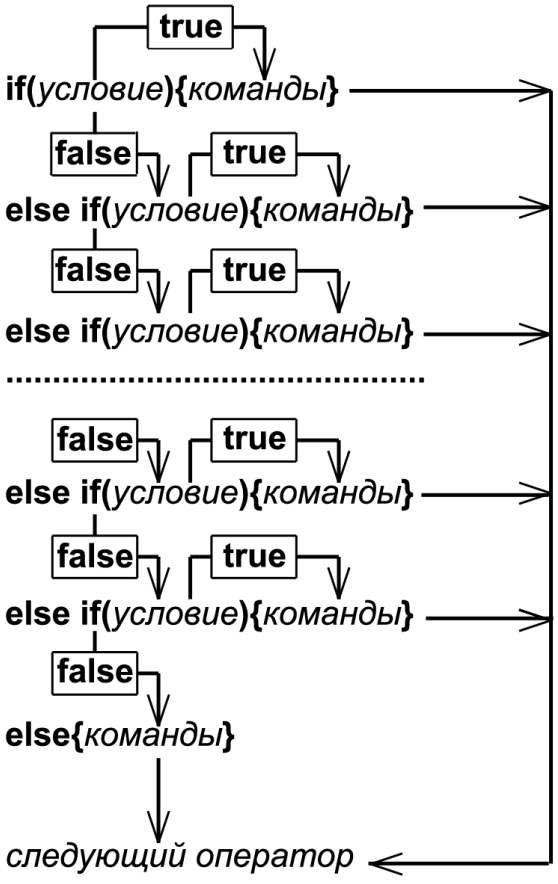


Рис. 4.3. Вложенные условные операторы

Самый последний `else`-блок обязательным не является. Пример использования условного оператора в программе приведен в листинге 4.1.

Листинг 4.1. Условный оператор

```
import javax.swing.*;
public class UsingIf{
   public static void main(String[] args){
   // Числа (числитель и знаменатель):
   double x,y;
   // Заголовок окна:
   String title="Деление чисел";
   // Текст сообщения (начальное значение):
   String text="Результат деления: ";
   // Переменная определяет тип сообщения:
   int type;
   // Считывание числителя:
   x=Double.parseDouble(JOptionPane.showInputDialog("Числитель:"));
   // Считывание знаменателя:
   y=Double.parseDouble(JOptionPane.showInputDialog("Знаменатель:"));
   // Условный оператор: равен ли нулю знаменатель?
   if(y!=0){type=JOptionPane.PLAIN_MESSAGE; // Знаменатель не равен нулю
         text=text+x+"/"+y+"="+x/y;}
   else{type=JOptionPane.ERROR_MESSAGE;     // Знаменатель равен нулю
         text=text+"деление на ноль!";}
   // Отображение окна с сообщением:
   JOptionPane.showMessageDialog(null,text,title,type);
   }
}
```

Программа достаточно простая. Пользователем последовательно вводится два действительных числа, после чего вычисляется результат деления одного числа на другое. Числа вводятся в полях диалоговых окон, которые отображаются одно за другим. После того, как числа введены, проверяется, с помощью условного оператора, не равен ли нулю знаменатель (число, на которое выполняется деление). Если знаменатель отличен от нуля, одно число делится на другое и результат с помощью окна сообщения выводится на экран. Если знаменатель нулевой, то выводится сообщение о попытке деления на ноль (хотя такая попытка на самом деле не делается). Для удобства программный код пестрит множественными комментариями.

На заметку:

Напомним: однострочный комментарий начинается двойной косой чертой `//`. Также можно закомментировать блок текста, если заключить его между инструкциями `/*` и `*/` (или между инструкциями `/**` и `*/`).

Проанализируем код. В программе объявляются две переменные x и y – обе типа double. Впоследствии мы попытаемся поделить одну переменную на другую. Пока же они просто объявлены. Текстовая переменная (тип String) title со значением "Деление чисел" нужна для того, чтобы задать заголовок окна сообщения, что выводится на экран и представляет результат выполнения программы. Еще одна текстовая переменная text с начальным значением "Результат деления: " используется для записи текста, который выводится в окне сообщения. Текущее значение переменной будет дополнено в процессе выполнения условного оператора после считывания чисел. Тип выводимого сообщения (отсутствие или наличие пиктограммы в области сообщения) зависит от того, какие пользователь введет числа. Поэтому объявляем целочисленную переменную type (пока без значения), которой затем в качестве значения присвоим одну из констант класса JOptionPane.

После того, как все переменные объявлены, производится считывание чисел. Значение переменной x определяется командой x=Double.parseDouble(JOptionPane.showInputDialog("Числитель:")). Здесь для преобразования текстового представления числа в десятичное число типа double из класса-оболочки Double вызывается метод parseDouble().

На заметку:

Метод parseDouble() похож на метод parseInt() из класса Integer, с той лишь разницей, что преобразование выполняется в формат числа с плавающей точкой (разумеется, на основе текстового представления этого числа).

Затем точно так же считываем второе число (переменная y). Используем для этого команду y=Double.parseDouble(JOptionPane.showInputDialog("Знаменатель:")). После того, как числа считаны, выполняется условный оператор. В условном операторе проверяется условие y!=0. Значение этого выражения равно true, если второе число (переменная y) отлично от нуля. В этом случае выполняется две команды. Командой type=JOptionPane.PLAIN_MESSAGE задаем значение переменной type. Константа JOptionPane.PLAIN_MESSAGE означает, что в области окна сообщения пиктограммы не будет вообще. Командой text=text+x+"/"+y+"="+x/y формируется полный текст сообщения. К той фразе, которая была записана в переменную text, добавляется текстовое представление операции деления двух чисел и вычисленный (инструкция x/y) результат деления чисел.

Нечто похожее происходит и в случае, когда пользователь ввел второе число, равное нулю. Точнее, операции выполняются с теми же переменными,

но по-другому. А именно, в блоке `else` командой `type=JOptionPane.ERROR_MESSAGE` задается значение переменной `type`. Текст, который будет отображаться в окне сообщения, формируется командой `text=text+"деление на ноль!"`.

На заметку:

Константа `JOptionPane.ERROR_MESSAGE` означает, что окно сообщения будет содержать пиктограмму из серии "произошла ошибка". С этой константой и таким окном мы уже сталкивались.

На этом выполнение условного оператора заканчивается. Какие бы числа ни ввел пользователь (если это, конечно же, числа), выполнение условного оператора приведет к тому, что переменные `type` и `text` получат свои окончательные значения. Поэтому есть все основания для выполнения команды `JOptionPane.showMessageDialog(null,text,title,type)`, которой на экран выводится окно с сообщением.

Проиллюстрируем работу программы. После запуска откроется окно, в поле которого необходимо ввести значение для первого числа (числителя). Окно показано на рис. 4.4.

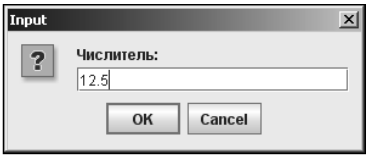


Рис. 4.4. Ввод первого числа

Практически так же, без интриги, вводится второе число, или знаменатель (рис. 4.5).

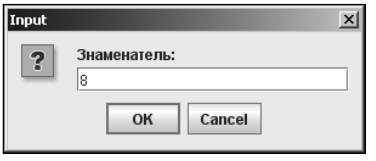


Рис. 4.5. Ввод второго числа

После этого появляется диалоговое окно с информацией о том, каков результат деления одного числа на другое (рис. 4.6).

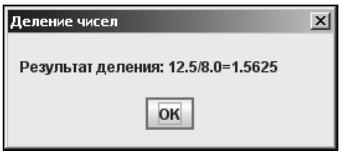


Рис. 4.6. Результат деления чисел

Но это в случае, если знаменатель (второе число) отличен от нуля. Если второе число равно нулю, сообщение появляется несколько иное, как на рис. 4.7.

На заметку:

Если при вводе чисел вместо того, чтобы ввести число и для подтверждения щелкнуть кнопку **OK**, щелкнуть кнопку отмены **Cancel**, программа экстренно завершит работу с ошибкой `NullPointerException` (ошибка при работе с указателем). Такие ситуации называются исключительными, и их можно (и нужно) обрабатывать. Исключениям посвящена отдельная глава книги. Пока же лучше таких ситуаций не допускать и давить именно те кнопки, которые нужно давить.

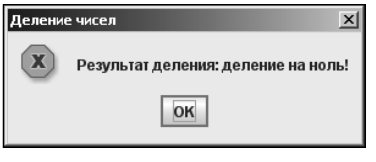


Рис. 4.7. Деление на ноль

Пример программы с вложенными условными операторами приведен в листинге 4.2.

Листинг 4.2. Вложенные условные операторы

```
import javax.swing.*;
public class UsingMoreIf{
   public static void main(String[] args){
   // Текстовые переменные для считывания ввода пользователя
   // и записи текста, выводимого в диалоговом окне:
   String text,str;
   // Считывание текста (числа), введенного пользователем:
   text=JOptionPane.showInputDialog("Введите целое число:");
   // Проверка корректности ввода:
   if(text==null){System.exit(0);}
   // Определение числа (преобразование текста в число):
   int num=Integer.parseInt(text);
   // Вложенные условные операторы:
   if(num<0){str="Вы ввели отрицательное число!";}
   else if(num>100){str="Это очень большое число!";}
   else if(num>10){str="Число, большее десяти!";}
   else{str="Число в пределах от 0 до 10!";}
   // Отображение диалогового окна:
   JOptionPane.showMessageDialog(null,str);
   }
}
```

Основу программы составляют:

- Условный оператор в упрощенной форме, которым проверяется, ввел ли пользователь число.
- Система вложенных условных операторов, в которой проверяется принадлежность введенного числа тому или иному диапазону значений.

Программа выполняется так. Выводится окно с просьбой ввести целое число. Введенное пользователем число считывается, и в зависимости от его значения выводится информационное сообщение. Различаются такие случаи:

- введенное число меньше нуля (отрицательное число);
- введенное число не меньше нуля и не больше десяти;
- введенное число больше десяти, но не больше ста;
- введенное число больше ста.

В программе командой `String text,str` объявляются две текстовые переменные (пока без значения). В переменную `text` будет считано текстовое представление введенного пользователем числа. В переменную `str` запишем текст, который отобразим в диалоговом окне, отображаемом после того, как пользователь введет число. Текст (текстовое представление числа) считыва-

ется командой `text=JOptionPane.showInputDialog("Введите целое число:")`. При этом может возникнуть одна неприятная ситуация. Если у пользователя сдадут нервы, и он не рискнет щелкнуть кнопку **OK**, а вместо этого малодушно щелкнет кнопку **Cancel**, никакой текст считан не будет. В таких кульминационных случаях методом `showInputDialog()` возвращается так называемая *пустая ссылка*. На нормальном языке это означает, что текстовая переменная `text` никакого текста не содержит. На языке программирования это означает, что текстовая переменная равна значению `null` (так называемая *пустая ссылка*).

На заметку:

С инструкцией `null` мы уже сталкивались, когда передавали ее первым аргументом методу `showMessageDialog()`. Там инструкция `null` означала, что отображаемое диалоговое окно не имеет *никакого* родительского окна (то есть не имеет окна, которое порождает данное окно). В данной программе инструкция `null` при проверке значения переменной `text` означает, что эта переменная не содержит *никакого* текста.

Проверка того обстоятельства, что пользователем введен текст, выполняется командой `text==null`. Это логическое выражение. Если его значение равно `true`, это означает, что переменная `text` не содержит значения. Именно это обстоятельство отслеживается с помощью условного оператора, который использован в упрощенной форме. Если значением выражения `text==null` является `true`, выполняется команда `System.exit(0)`. Методом `exit()` класса `System` завершается работа программы. Аргументом методу передается код завершения программы. Нулевой код означает, что работа программы завершена в нормальном режиме.

Если команда `System.exit(0)` выполнена, никакие другие команды уже выполняться не будут. Таким образом, если вместо того, чтобы ввести число в поле ввода и щелкнуть кнопку **OK**, пользователь сделает отмену (щелкнет кнопку **Cancel**), работа программы будет завершена. Если этого не произошло, мы смело делаем вывод, что пользователь что-то ввел в поле ввода. Более того, мы предполагаем, что это текстовое представление числа. Поэтому командой `int num=Integer.parseInt(text)` объявляем целочисленную переменную `num` и в качестве значения присваиваем ей результат преобразования текста `text` в число.

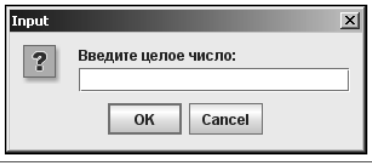
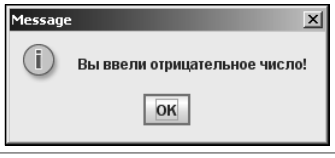
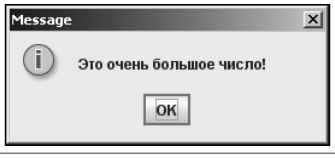
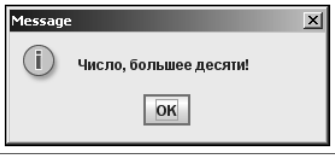
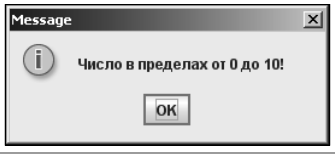
После этого выполняется блок из вложенных уловных операторов, в котором проверяется, какому диапазону значений принадлежит введенное пользователем число. В соответствии с этим определяется значение текстовой переменной `str`.

Сначала проверяем условие `num<0`. Если условие выполнено, переменная `str` получает значение `"Вы ввели отрицательное число!"`.

Если условие num<0 не выполнено, проверяется условие num>100. Для значений переменной num, больших 100, переменная str содержит текст "Это очень большое число!". Если и это условие не выполнено, проверяется условие num>10. Для случая истинности данного выражения выполняется команда str="Число, большее десяти!". На случай, когда ни одно из проверяемых условий не является истинным, предусмотрена команда str="Число в пределах от 0 до 10!". Последней в программном коде размещена команда JOptionPane.showMessageDialog(null,str), которой отображается окно с сообщением, в котором указано, какого типа число ввел пользователь.

Окна, которые появляются или могут появляться в процессе выполнения программы, представлены в табл. 4.1. Там же дано краткое описание ситуаций, при которых эти окна появляются.

Табл. 4.1. Диалоговые окна программы

Диалоговое окно	Описание
	Окно для ввода числа пользователем. Окно отображается в начале работы программы
	Окно отображается в том случае, если введенное число меньше нуля
	Окно отображается в том случае, если введенное число больше 100
	Окно отображается, если введенное число больше 10, но не больше 100
	Окно отображается, если введено целое число от 0 до 10 включительно

Дальше рассмотрим оператор выбора, или оператор switch-case.

Оператор выбора switch-case

Оператор switch-case **позволяет сделать выбор из нескольких вариантов.** Синтаксис у оператора следующий:

```
switch(выражение){
   case значение_1:
           команды_1
           break;
   case значение_2:
           команды_2
           break;
   ...
   case значение_N:
           команды_N
           break;
   default:
           команды_N+1
   }
```

Сначала указывается ключевое слово switch. В круглых скобках после этого слова указывается *выражение*, значение которого проверяется в результате выполнения оператора. Результатом выражения может быть целое число или символ. Затем идет блок из фигурных скобок. Блок состоит из case-инструкций. В case-инструкции после ключевого слова case указываем возможное значение *выражения*, и завершается все это двоеточием. Далее следуют команды. Это именно те команды, которые выполняются, если выражение принимает значение, указанное после ключевого слова case.

Оператор switch-case выполняется так. Вычисляется значение выражения, указанного после ключевого слова switch в круглых скобках. Затем по очереди выполняется проверка на предмет совпадения результата выражения со значениями, указанными в case-блоках. Если совпадение найдено, начинают выполняться команды соответствующего case-блока. Если совпадения нет, выполняются команды в последнем блоке, обозначенном ключевым словом default. Этот блок, кстати, не является обязательным и может не указываться в операторе switch-case. Собственно, все.

На заметку:

Обычно каждый case-блок заканчивается командой break. Причина кроется в способе выполнения команд case-блока. Если уж они начали выполняться, то будут последовательно выполняться все команды не только данного case-блока, но и все последующие – вплоть до окончания оператора switch-case или пока не встретится инструкция break. Чтобы инструкция break встретилась, ее нужно указать.

Алгоритм выполнения оператора `switch-case` проиллюстрирован на рис. 4.8.

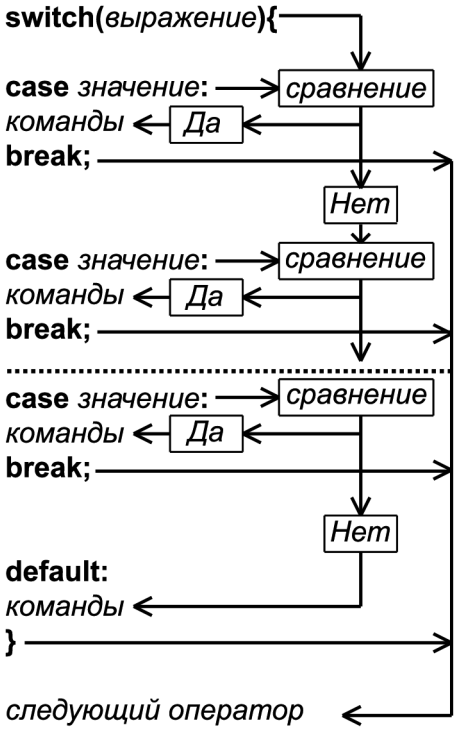


Рис. 4.8. Алгоритм выполнения оператора `switch-case`

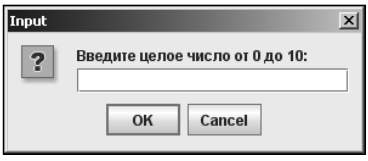
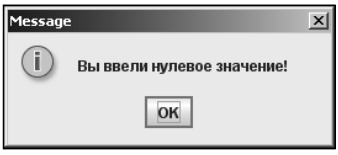
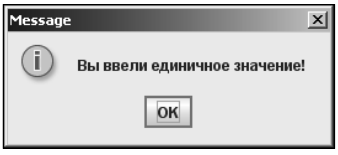
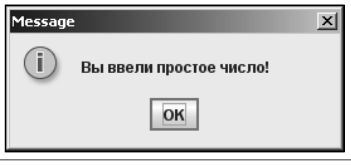
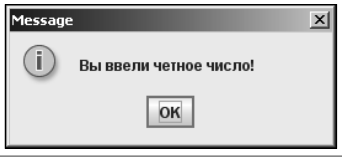
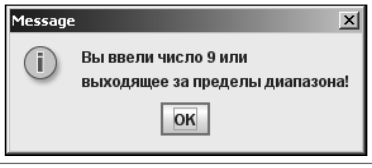
Пример использования `switch-case` оператора в программе приведен в листинге 4.3.

Листинг 4.3. Оператор выбора

```
import javax.swing.*;
public class UsingSwitch{
   public static void main(String[] args){
   // Вводимое пользователем число:
   int num;
   // Текст из поля ввода и текст сообщения:
   String text,str;
   // Отображение окна и считывание текста:
   text=JOptionPane.showInputDialog("Введите целое число от 0 до 10:");
   // Преобразование текста в число:
   num=Integer.parseInt(text);
   // Оператор выбора:
   switch(num){
      case 0:
         str="Вы ввели нулевое значение!";
         break;
      case 1:
         str="Вы ввели единичное значение!";
         break;
      case 2:
      case 3:
      case 5:
      case 7:
         str="Вы ввели простое число!";
         break;
      case 4:
      case 6:
      case 8:
      case 10:
         str="Вы ввели четное число!";
         break;
      default:
      str="Вы ввели число 9 или\nвыходящее за пределы диапазона!";
      }
   JOptionPane.showMessageDialog(null,str);
   }
}
```

В начале выполнения программы на экране появляется окно с предложением ввести в поле ввода целое число в диапазоне от 0 до 10. После считывания введенного пользователем значения начинается перебор вариантов. В зависимости от значения числа, отображается окно сообщения с текстом. Исходное окно с полем ввода и окна, отображаемые для разных числовых значений, введенных пользователем, представлены и описаны в табл. 4.2.

Табл. 4.2. Окна, выводимые программой с оператором выбора

Окно	Описание
	Диалоговое окно для ввода числового значения. Отображается в начале выполнения программы
	Окно с сообщением о том, что введено нулевое значение. Отображается, если пользователь ввел ноль
	Окно с сообщением о том, что пользователь ввел единичное значение. Отображается тогда же
	Окно с сообщением о том, что введено простое число. Окно отображается, если пользователь ввел число 2, 3, 5 или 7
	Окно с сообщением о том, что пользователем введено четное число. Отображается, если введено число 4, 6, 8 или 10
	Окно с сообщением о том, что введено число 9 или другое число, выходящее за пределы диапазона от 0 до 10. Окно отображается в соответственных случаях

Разберем программный код, представленный в листинге 4.3. Начальная часть программы, думается, особых комментариев не требует. Там объявляется целочисленная переменная `num` и две текстовые переменные `text` и `str`. Введенное пользователем числовое значение считывается в виде текста и записывается в переменную `text`. Текст преобразуется в число, и результат присваивается переменной `num`. После этого в игру вступает оператор выбора.

После ключевого слова `switch` в скобках указано имя переменной `num`. Это означает, что значение переменной `num` будет последовательно сравниваться со значениями, указанными в case-блоках, до первого совпадения или до

достижения конца оператора выбора. Первый case-блок предназначен для значения переменной num, равного 0. В этом случае (то есть если пользователь ввел нулевое значение) выполняется команда str="Вы ввели нулевое значение!" и затем инструкцией break дальнейшее выполнение оператора выбора прекращается. Команды второго case-блока запускаются при значении переменной num равном 1. В этом случае переменная str получает значение "Вы ввели единичное значение!".

Как и в предыдущем случае, завершается работа блока инструкцией break. Дальше становится интереснее. На пути встречается сразу четыре case-инструкции, причем три из них пустые, то есть не содержат команд в своем блоке. Эффект от такой удручающей на первый взгляд пустоты состоит в том, что команды четвертого case-блока выполняются для любого из четырех значений, перечисленных в списке case-инструкций. Объясняется все просто. Как мы уже знаем, в процессе выполнения оператора выбора производится поиск совпадений. Если совпадение найдено, в том числе и на пустой case-инструкции, то начинают выполняться команды в соответствующем блоке, пока не встретится команда break. Поскольку в пустом блоке такой команды нет, этот блок пропускается и выполняется команда следующего блока. Если и этот блок пустой, то выполняется команда первого непустого блока.

Использование пустых case-блоков оправдано, если для нескольких значений проверяемого выражения должны выполняться одни и те же действия. В данном случае для значений переменной num равных 2, 3, 5 и 7 выполняется команда str="Вы ввели простое число!". Если же пользователем введено число из набора значений 4, 6, 8 и 10 (это следующий case-блок), то выполняется команда str="Вы ввели четное число!". Наконец, если совпадений не найдено (а это возможно, если введено число 9 или число, не попадающее в диапазон значений от 0 до 10), то в блоке default выполняется команда str="Вы ввели число 9 или\nвыходящее за пределы диапазона!". Кульминацией программы становится отображение командой JOptionPane.showMessageDialog(null,str) окна сообщения с соответствующим текстом.

Операторы цикла while и do-while

Оператор while **и его модификация** do-while **позволяют создавать в программе** *циклы с условием*, **то есть выполнять какое-то количество раз одни и те же команды пока истинно какое-либо условие.** Эффект от использования оператора цикла связан с тем, что команды в программном коде набираются один раз, а выполняются столько раз, сколько нужно.

Сначала рассмотрим оператор цикла `while`. У оператора такой, достаточно простой, синтаксис:

```
while(условие){
   команды
   }
```

Начинается оператор ключевым словом `while`, после которого в круглых скобках указывается *условие* – выражение логического типа, которое может принимать значение `true` или `false`. Затем блок команд в фигурных скобках (команды основного тела оператора `while`).

Выполняется оператор `while` так. Сначала вычисляется значение выражения, указанного в круглых скобках после ключевого слова `while` (другими словами, проверяется *условие*). Если значение выражения равно `true` (*условие* истинно), выполняется блок команд в фигурных скобках. Затем снова проверяется *условие*. Если условие истинно, снова вычисляются команды основного тела оператора `while` и проверяется *условие*. Такая чехарда продолжается до тех пор, пока не перестанет выполняться *условие*. Как только для выражения-условия получено значение `false`, выполнение оператора `while` прекращается и управление передается следующему после него оператору или команде.

На заметку:

Обращаем внимание, что условие проверяется только после того, как выполнены *все* команды основного тела оператора `while`. То есть проверяется *условие*, и если оно истинно, выполняются *все* команды тела оператора. И только после этого снова проверяется *условие*.

На рис. 4.9 проиллюстрирован алгоритм выполнения оператора цикла `while`.

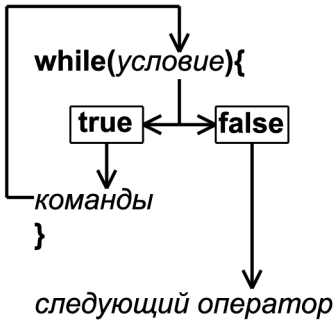


Рис. 4.9. Алгоритм выполнения оператора цикла `while`

Акцентируем внимание вот на чем: в основном теле оператора `while` *необходимо предусмотреть* возможность изменения *условия*, иначе получится бесконечный цикл, который никогда не закончится без применения грубой силы.

Модификация `do-while` оператора цикла отличается от рассмотренной выше `while`-версии тем, что в операторе `do-while` сначала выполняются команды основного тела оператора, а уже затем проверяется условие. Синтаксис оператора `do-while` такой:

```
do{
   команды
   }while(условие);
```

После ключевого слова `do` в фигурных скобках указываются команды основного тела оператора цикла. После блока команд следует ключевое слово `while`, после которого в круглых скобках указано условие (выражение логического типа). Заканчивается вся эта конструкция точкой с запятой.

Начинает выполняться оператор `do-while` с блока команд, после чего проверяется *условие*. Если *условие* истинно, то снова выполняются команды основного тела цикла, проверяется условие, и так далее (пока не перестанет выполняться условие). Ситуацию иллюстрирует рис. 4.10.

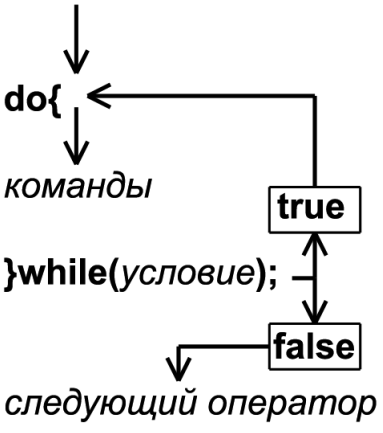


Рис. 4.10. Алгоритм выполнения оператора `do-while`

На заметку:

Выполнение оператора цикла `while` начинается с проверки условия. Выполнение оператора цикла `do-while` начинается с выполнения блока команд. Поэтому в операторе `do-while` блок команд выполняется, по крайней мере, один раз. Во всем остальном различия между этими операторами иллюзорные.

Программа, код которой представлен в листинге 4.4, дает представление о том, как операторы цикла `while` и `do-while` используются на практике.

Листинг 4.4. Операторы цикла `while` и `do-while`

```
import javax.swing.*;
public class UsingWhile{
   public static void main(String[] args){
   // Граница суммы, индексные переменные и переменные для записи суммы:
   int count,i=1,j=1,s1=0,s2=0;
   // Считывание границы суммы:
   count=Integer.parseInt(JOptionPane.showInputDialog("Введите границу суммы"));
   // Текстовые переменные:
   String text="Сумма натуральных чисел от 1 до "+count+".\n";
   String str1="Оператор while: ";
   String str2="Оператор do-while: ";
   // Вычисление суммы оператором while:
   while(i<=count){
      s1+=i;
      i++;}
   // Вычисление суммы оператором do-while:
   do{
      s2+=j;
      j++;
      }while(j<=count);
   // Уточнение текста для сообщения:
   str1+=s1+"\n";
   str2+=s2;
   // Вывод окна сообщения на экран:
   JOptionPane.showMessageDialog(null,text+str1+str2);
   }
}
```

Суть программы проста: с ее помощью вычисляется сумма натуральных чисел. Верхняя граница суммы вводится пользователем в начале выполнения программы. Сумма вычисляется двумя способами. Точнее, способ один, но операторы используются разные: `while` и `do-while`. Результат, полученный с помощью этих двух операторов, выводится на экран с помощью окна сообщения. Искренне хочется верить, что манипуляции с текстовыми переменными читателю понятны и во многом знакомы. Вооружившись этой верой, рассмотрим лишь часть кода, связанную с выполнением операторов `while` и `do-while`.

В операторе `while` используются три целочисленные переменные: индексная переменная `i` с начальным значением `1`, переменная `count`, в которую записана верхняя граница суммы, и переменная `s1` с начальным нулевым значением, в которую пошагово записывается результат вычисления суммы.

В операторе цикла while проверяется условие i<=count. Другими словами, оператор цикла выполняется до тех пор, пока индексная переменная не превышает значение переменной count. Начинается работа оператора while именно с проверки этого условия. Если оно истинно (а истинно оно при значении переменной count не меньшем единицы – поскольку начальное значение переменной i равно 1), выполняются две команды в основном теле оператора цикла. Первой командой s1+=i текущее значение переменной s1 увеличивается на величину i, а второй командой i++ на единицу увеличивается значение индексной переменной i. После выполнения оператора цикла значение суммы натуральных чисел от 1 (начальное значение переменной i) до значения count записано в переменную s1.

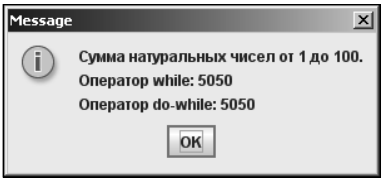


Рис. 4.11. Выполнение операторов while и do-while приводит к одинаковому результату

Та же сумма вычисляется с помощью оператора do-while, в котором использованы такие переменные: индексная переменная j с начальным значением 1, переменная count и переменная s2 для записи суммы натуральных чисел. С точностью до обозначений этот тот же джентльменский набор переменных, что и у оператора while. Собственно, и вычисления выполняются также – с поправкой на синтаксис оператора do-while. Более того, и результат, получаемый в обоих случаях для "нормального режима", одинаков. В этом несложно убедиться. Например, запускаем программу и вводим в качестве верхней границы суммы число 100. Получим результат, как на рис. 4.11.

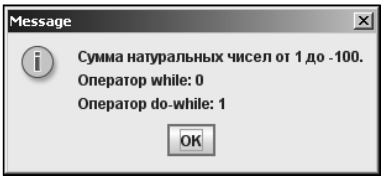


Рис. 4.12. Выполнение операторов while и do-while приводит к разным результатам

Открывшееся диалоговое окно содержит информацию о результате вычисления суммы двумя методами (двумя операторами), и несложно заметить, что результаты вычислений полностью совпадают. Это не случайно – если для верхней границы суммы введено число, большее или равное 1, оба результата тождественны. Ситуация изменится, если для границы суммы ввести отрицательное число. Хотя с точки зрения здравого смысла это бред, программа все равно работает. На рис. 4.12 показано окно с сообщением о вычислении суммы для верхней границы, равной -100.

Причем конкретное значение верхней границы здесь совершенно не важно – главное, чтобы число было отрицательным. Если так, то оператором while вычисляется значение 0, а оператором do-while вычисляется значение 1.

Объяснение простое. В операторе `while` сначала проверяется условие, что значение индексной переменной не больше верхней границы суммы. Поскольку индексная переменная инициализируется со значением `1`, а верхняя граница отрицательна, это условие не выполняется. Поэтому команды в основном теле оператора `while` тоже не выполняются. А в качестве результата возвращается значение переменной `s1`. Переменная имеет начальное нулевое значение, и в операторе цикла она не менялась. Отсюда нулевой результат для суммы.

В операторе `do-while` все так же, но команды в основном теле оператора `do-while` один раз выполняются, и только после этого проверяется условие. За этот "один раз" значение переменной `s2` (сумма чисел) с нуля увеличивается до `1`. Поэтому оператор `do-while` для суммы натуральных чисел возвращает единичное значение.

Оператор цикла `for`

Очень полезным и эффективным является оператор цикла `for`, который позволяет осуществлять какие-либо действия **заданное количество раз**. Синтаксис вызова оператора следующий:

```
for(инициализация;условие;изменение_знчений){
   команды
}
```

Начинается все с ключевого слова `for`, после которого следуют круглые скобки. В круглых скобках три блока выражений. Блоки разделяются точкой с запятой. Команды внутри блока разделяются запятыми.

🕮 На заметку:

Несколько команд может быть только в первом и третьем блоке. Второй блок состоит из одного логического выражения (или может быть пустым). Пустыми (то есть не содержащими команд) могут быть также первый и третий блоки. Первый блок обычно называется блоком *инициализации*, второй – блоком *условия*, третий – блоком *изменения* переменных.

Если в основном теле оператора цикла всего одна команда, фигурные скобки можно не указывать. Однако лучше их использовать всегда, поскольку это позволяет избежать случайных ошибок. Например, если забыть указать фигурные скобки для блока команд, то к оператору цикла будет относиться только первая команда этого блока. Такие ошибки компилятором обычно не отслеживаются, и программа хоть и работает, но неправильно.

После этого в фигурных скобках следует блок команд основного тела оператора цикла `for`. Схема выполнения оператора цикла `for` проиллюстрирована на рис. 4.13.

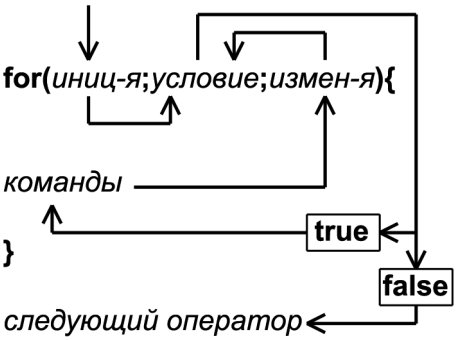


Рис. 4.13. Алгоритм выполнения оператора `for`

Начинается выполнение оператора цикла `for` с выполнения команды или команд первого блока (блока инициализации). Эти команды выполняются один-единственный раз и только в начале работы оператора цикла. После их выполнения вычисляется выражение, размещенное во втором блоке. Выражение должно иметь логический тип, поэтому обычно называется *условием*. Оператор цикла выполняется, пока истинно условие.

Другими словами, если при вычислении значения условия оказалось, что это значение `false`, работа оператора цикла завершается и управление передается следующему после него оператору или команде.

Если же значение условия равно `true` (условие истинно), то выполняются команды в основном теле оператора цикла. После них выполняются команды в третьем блоке ключевого слова `for`. Там обычно размещают команду или команды, связанные с изменением значения индексной переменной (или переменных). После этого снова проверяется *условие* (второй блок).

Если условие ложно, работа оператора цикла завершается. Если условие истинно, выполняются команды основного тела оператора цикла и так далее. Круг замкнулся.

Это стандартная и неэкзотическая схема выполнения оператора цикла. Проиллюстрируем ее на простом примере. Рассмотрим листинг 4.5.

Листинг 4.4. Операторы цикла `for`

```
import javax.swing.*;
public class UsingFor{
   public static void main(String[] args){
 // Граница суммы, индексная переменная и переменная для записи суммы:
   int count,i,s=0;
   // Считывание границы суммы:
   count=Integer.parseInt(JOptionPane.showInputDialog("Введите границу суммы"));
```

```
    // Текстовая переменная:
    String text="Сумма натуральных чисел от 1 до "+count+": ";
    // Вычисление суммы оператором for:
    for(i=1;i<=count;i++){
        s+=i;}
    // Вывод окна сообщения на экран:
    JOptionPane.showMessageDialog(null,text+s);
    }
}
```

Программой вычисляется сумма натуральных чисел, только теперь для расчетов используется оператор цикла `for`. Как и в предыдущих примерах, сначала выводится окно с просьбой ввести верхнюю границу для суммы (количество слагаемых). Затем вычисляется сумма, и результат отображается с помощью окна сообщения. На рис. 4.14 показан результат вычисления суммы натуральных чисел от `1` до `100`.

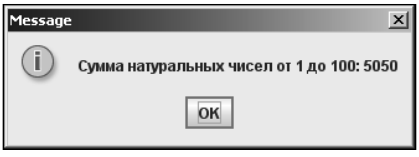


Рис. 4.14. Вычисление суммы натуральных чисел с помощью оператора цикла `for`

Рассмотрим подробнее способ вызова оператора цикла `for` в программе. В операторе цикла используется три целочисленные переменные: переменная `count`, в которую на момент вызова оператора цикла записана верхняя граница суммы (количество слагаемых), индексная переменная `i` (на момент вызова оператора цикла не инициализирована), а также переменная `s`, которая на момент вызова оператора цикла имеет нулевое значение (в эту переменную записывается результат вычисления суммы натуральных чисел).

Первый блок (блок *инициализации*) инструкции `for` состоит из одной команды `i=1`, которой индексной переменной присваивается единичное значение. После того, как переменная `i` получила значение, проверяется условие `i<=count` (второй блок). Если это условие истинно (а при значении `100` переменной `count` это так), выполняется блок команд в фигурных скобках. Блок состоит всего из одной команды `s+=i`, которой значение переменной `s` увеличивается на текущее значение индексной переменной `i` (на первом итерационном шаге это `1`). Затем выполняется команда `i++` в третьем блоке инструкции `for`. Этой командой индексная переменная увеличивается на единицу. Затем снова, уже при новом значении индексной переменной, проверяется условие `i<=count` и так далее. Поле выполнения оператора цикла в переменной `s` содержится значение суммы натуральных чисел. Именно это значение выводится на экран в окне сообщения.

Как отмечалось ранее, первый и третий блок оператора цикла могут содержать несколько команд. Более интересно то, что некоторые или даже все блоки могут быть пустыми. В любом случае алгоритм выполнения оператора цикла остается неизменным. Просто если какой-то блок пустой, он просто пропускается. В табл. 4.3 приведены некоторые примеры альтернативного к рассмотренному выше способу вызова оператора цикла для вычисления суммы натуральных чисел. Поскольку в некоторых случаях изменение синтаксиса вызова оператора цикла предполагает внесение правок в способ объявления переменных, вместе с кодом вызова оператора цикла приводится также и команда объявления переменных, используемых в операторе цикла. Остальной код программы следует полагать неизменным.

Табл. 4.3. Разные способы вызова оператора цикла

Код вызова оператора цикла	Описание

`int count,i,s;`
`...`
`for(i=1,s=0;i<=count;i++){`
`s+=i;}`
	В этом варианте переменная `s` объявляется, но не инициализируется. Команда инициализации переменной `s` вынесена в первый блок (блок инициализации)

`int count,i,s;`
`...`
`for(i=1,s=0;i<=count;s+=i,i++);`
	Команда `s+=i` из основного тела оператора цикла перенесена в третий блок. Порядок следования команд в третьем блоке важен: сначала должна выполняться команда `s+=i`, а затем команда `i++`. Основное тело оператора цикла состоит из пустого оператора – фигурные скобки отсутствуют, команды отсутствуют, только точка с запятой

`int count,s=0;`
`...`
`for(int i=1;i<=count;s+=i++);`
	В блоке инициализации можно не только присваивать значения, но и объявлять переменные. В данном случае объявление индексной переменной `i` вынесено в первый блок оператора цикла. Основное тело оператора цикла не содержит команд, а две команды третьего блока объединены в одну команду `s+=i++`

`int count,i=0,s=0;`
`...`
`for(;i<count;s+=i){`
`i++;}`
	Оператор цикла с пустым первым блоком. Инициализация индексной переменной выполняется при объявлении до запуска оператора цикла. В третьем блоке размещена команда `s+=i` изменения значения переменной `s`, а команда изменения индексной переменной `i++` вынесена в основное тело оператора цикла. Обращаем внимание, что поскольку в результате такой рокировки меняется порядок выполнения команд изменения значений переменных `i` и `s`, начальное значение переменной `i` устанавливается равным нулю

`int count,i=1,s=0;`
`...`
`for(;i<=count;){`
`s+=i;`
`i++;}`
	Первый и третий блоки оператора цикла пустые. Инициализация индексной переменной `i` и переменной `s` выполнена при их объявлении, а команды изменения этих переменных вынесены в основное тело оператора

`int count,i=0,s=0;`
`...`
`for(;;){`
`s+=++i;`
`if(i>=count) break;}`
	Все три блока оператора цикла пустые. Инициализация переменных `s` и `i` выполняется при объявлении. Команды изменения переменных `s` и `i` объединены в одну команду `s+=++i`. Поскольку в этой команде оператор инкремента использован в префиксной форме, переменная `i` инициализируется не с единичным, а с нулевым значением. Поскольку второй блок пустой, то формально цикл бесконечный. Чтобы предусмотреть своевременное завершение цикла, в основное тело оператора добавлена команда `if(i>=count) break`. В соответствии с этой командой, если индексная переменная `i` становится не меньше переменной `count`, работа оператора цикла завершается (инструкция `break`)

Мы уже второй раз сталкиваемся с инструкцией `break`. Сначала мы использовали ее, чтобы завершить работу оператора выбора, здесь, в последнем примере, с помощью этой инструкции завершался формально бесконечный оператор цикла. Кроме инструкции `break` есть еще одна полезная инструкция: `continue`. Если инструкцией `break` завершается весь оператор цикла, то инструкцией `continue` досрочно завершается выполнение текущего цикла и начинает выполняться следующий цикл.

На заметку:

Начиная с версии JDK 5, в Java есть еще одна форма оператора цикла `for`, которая позволяет перебирать элементы из набора данных (в нашем случае массива) без использования индексной переменной. Эта форма оператора цикла `for` обсуждается в следующей главе.

Резюме

1. Операторы цикла и условные операторы позволяют создавать точки ветвления в программе и многократно повторять однотипные действия.
2. К условным операторам относят `if`-оператор и `switch-case`-оператор (оператор выбора). К операторам цикла относят операторы `for`-оператор, `while`-оператор и его модификацию `do-while`-оператор.
3. В условном `if`-операторе проверяется условие, и если оно истинно, выполняется блок команд после условия. В противном случае выполняются команды после ключевого слова `else`. Может использоваться упрощенный вариант оператора без `else`-ветки.
4. В `switch-case`-операторе вычисляется значение выражения, и после этого в соответствии с полученным значением выполняется тот или иной `case`-блок.
5. В операторах цикла блок команд выполняется циклически до тех пор, пока остается истинным некоторое условие. В зависимости от типа оператора цикла, условие проверяется перед началом выполнения блока команд или после него. Если при вычислении условия получено значение `false`, выполнение оператора цикла прекращается.
6. Инструкция `break` используется для завершения работы оператора цикла или оператора выбора. Инструкция `continue` позволяет досрочно завершить один итерационный цикл.

Глава 5
Создание и работа с массивами

Один за всех, и все за одного!
(Из к/ф "Д'артаньян и три мушкетера")

В программировании без массивов сложно обойтись, хотя не многие о них знают. Массивы позволяют достаточно просто группировать данные. Формально массив представляет собой набор однотипных переменных (то есть таких переменных, которые относятся к одному типу, причем не обязательно базовому), объединенных общим именем. Переменные, которые входят в массив, называются *элементами* этого массива. Для того чтобы различить разные переменные (элементы) в пределах массива, каждый элемент массива *индексируется*. Это позволяет обращаться к элементу массива, указав имя массива и индекс (или индексы) элемента в массиве. Образно выражаясь, все переменные в массиве перенумерованы с помощью индексов. Индекс может быть один, а может быть и не один. Количество индексов, необходимых для однозначного определения элемента массива, называется размерностью массива. Самый простой тип массива – одномерный массив. Элементы одномерного массива индексируются посредством одного индекса. Именно с одномерных массивов начнем рассмотрение вопроса о работе с массивами в Java.

Одномерные массивы

В Java все массивы *динамические*. Если читателю эта сакраментальная фраза ни о чем не говорит, ее можно проигнорировать. Смысл сказанного станет понятен из нижеизложенного.

В Java достаточно изысканная схема создания массивов, в том числе одномерных. Главная идея состоит в том, что создается так называемая *переменная массива*. Это переменная, через которую осуществляется доступ к массиву. **Сам массив представляет собой область памяти, содержащую фактические данные, то есть значения элементов массива. Переменная массива фактически содержит адрес этой области памяти.** Создателями Java мудро предусмотрено, чтобы пользователю не было необходимости (и возможности тоже) участвовать в манипуляциях с адресами. Весь этот жуткий механизм надежно скрыт. **Для доступа к элементу массива достаточно знать имя переменной массива и индекс элемента в массиве.** Тем не менее, объективности ради, следует отметить, что процедура *создания* массива на первый взгляд несколько сложновата. Создание массива в Java подразумевает реализацию трех этапов:

1. Создание (объявление) переменной массива.
2. Создание непосредственно массива – выделение в памяти места для хранения значений элементов массива.
3. Присваивание переменной массива в качестве значения ссылки на массив. В данном случае речь идет о том, что переменная массива получает в качестве значения адрес памяти, где хранится массив.

Таким образом, переменная массива с технической точки зрения не есть массив. Это переменная, которая "знает", где массив расположен. Тем не менее, после создания операции с переменной массива выполняются так, как если бы эта переменная и была массивом. Это удобно.

Рассмотрим последовательно каждый из упомянутых этапов создания массива. В первую очередь создаем переменную массива. Создаем – означает *"объявляем"*. Собственно, здесь ничего сложного нет. По большому счету, это такая же переменная, как и все прочие. Поэтому объявляется она точно так же, как и другие переменные: указывается тип переменной и ее имя. Имя для переменной выбираем сами исходя из соображений наглядности, легитимности и целесообразности.

С типом переменной массива несколько сложнее. Проиллюстрируем на примере. Предположим, планируется создать целочисленный массив, то есть массив, элементы которого являются переменными целочисленного типа (пускай это будет тип `int`). Этот факт необходимо как-то отобразить при создании переменной массива. Если просто написать тип элементов массива и имя переменной массива, то это будет объявление обычной целочисленной переменной. Поэтому, чтобы компилятор понял, что речь идет о переменной массива, а не об обычной переменной, после имени типа элементов массива указываются пустые квадратные скобки. Так, если мы хотим создать переменную для целочисленного массива с именем `MyArray`, используем такую команду:

```
int[] MyArray;
```

Это правило остается справедливым и для массивов других типов, причем не только базовых. Оно гласит: *при объявлении переменной одномерного массива указывается тип элементов массива, пустые квадратные скобки и имя переменной.*

📖 На заметку:

Кроме базовых типов, которые мы уже обсуждали, в Java существуют *ссылочные* типы. Переменные ссылочных типов используются при работе с объектами. Текстовые переменные, которые постоянно используются в этой книге в примерах, являются примером переменных ссылочного типа.

Объявление переменной массива совсем не означает, что массив создан. При объявлении переменной массива в памяти выделяется место для этой переменной. Значением переменной, напомним, является ссылка на массив. Но если массива нет, то и ссылку не на что делать. Массив создается независимо от переменной массива. После создания массива переменной массива необходимо в качестве значения присвоить ссылку на массив. Другими словами, нужно:

1. Создать массив.
2. Каким-то образом этот массив отловить – получить на него ссылку.

Обе задачи решаются с помощью оператора `new`. Это удивительный оператор, который мы будем использовать очень часто, и не только при создании массивов. Его общее назначение – выделение памяти в динамическом режиме.

📖 На заметку:

В процессе работы программа оперирует с данными, которые реализуются в виде простых переменных, массивов, объектов. Для них нужно выделять память. Память можно выделять:

а) при компиляции программы (то есть до того, как программа запущена на выполнение);

б) непосредственно при выполнении программы.

В первом случае говорят о *статическом* распределении памяти. Если память выделяется в процессе выполнения программы, говорят о *динамическом* распределении памяти. В Java память для массивов и объектов выделяется непосредственно в процессе выполнения программы, то есть *динамически*. Именно для динамического выделения памяти служит оператор `new`. При этом для *переменной массива* память выделяется *статически*.

Действие оператора `new` состоит в том, что в памяти выделяется необходимое место для хранения данных. Оператор возвращает результат. Как несложно догадаться, результатом оператора `new` является ссылка на область памяти, которая выделена.

📖 На заметку:

Строго говоря, оператор `new` пытается выделить память. Эта операция может в принципе закончиться неудачно – например, если свободная память закончилась. В таких случаях возникают *ошибки времени исполнения*. Пока мы будем делать вид, что никаких неприятностей при выделении памяти произойти не может. Обработке ошибок времен исполнения посвящена отдельная глава книги.

При выделении памяти оператору `new` нужно как-то намекнуть, для чего эта память выделяется и в каком объеме. В данном случае нас интересует выделение памяти под одномерный массив. Память для массива выде-

ляется так: указывается оператор new, после чего тип элементов массива и в квадратных скобках количество элементов в массиве. Например, чтобы выделить память для целочисленного массива из 10 чисел, используем инструкцию вида new int[10]. Эту инструкцию в качестве значения нужно присвоить переменной массива. Весь процесс создания массива из 10 чисел типа int может выглядеть следующим образом:

```
// Объявление переменной массива:
int[] MyArray;
// Создание массива и запись ссылки на массив в переменную массива:
MyArray=new int[10];
```

Однако на практике команды объявления переменной массива и создания массива обычно объединяют в одну, так что приведенный выше фрагмент кода может быть переписан следующим образом:

```
int[] MyArray=new int[10];
```

На заметку:

Иногда при объявлении переменной массива пустые квадратные скобки указывают не после названия типа элементов массива, а после имени переменной. Например, вместо инструкции объявления массива int[] MyArray можно воспользоваться командой int MyArray[]. Так тоже правильно. Тем не менее, существует разница в том, как объявлять переменные. Например, командой double[] x,y объявляются две переменные x и y массива из double-элементов, а командой double x[],y объявляется переменная массива x и обычная переменная y типа double.

Обращение к элементу массива выполняется так: указывается имя массива (имя переменной массива, которая ссылается на массив) и в квадратных скобках индекс массива. *Индексация элементов массива в Java всегда начинается с нуля!* Другими словами, самый первый элемент массива имеет индекс 0. Например, если для упоминавшегося выше массива MyArray обращение к первому элементу выполняется как MyArray[0]. Индекс последнего элемента массива на единицу меньше размера массива (количество элементов в массиве). Последним элементом массива MyArray[0] будет MyArray[9].

На заметку:

Для каждого массива автоматически вычисляется свойство length, определяющее размер массива (количество элементов в массиве). Обращение к свойству length выполняется через имя массива с использованием "точечного" синтаксиса: указывают имя массива и, через точку, свойство length. Например, чтобы узнать, сколько элементов в массиве MyArray, используем инструкцию MyArray.length.

Кроме того, в Java выполняется автоматическая проверка на предмет выхода за пределы массива. То есть если программный код содержит инструкцию обращения к заведомо несуществующему элементу массива, во время выполнения программы появится сообщение об ошибке. Хотя это и кажется естественным, в языке C++, например, такая проверка не выполняется.

В листинге 5.1 приведен пример программы, в которой создается одномерный целочисленный массив, который заполняется нечетными числами и затем поэлементно выводится на экран.

Листинг 5.1. Одномерный массив из нечетных чисел

```
import javax.swing.*;
public class SimpleArray{
public static void main(String[] args){
// Индексная переменная и размер массива:
int i,size;
// Текст для отображения:
String text="Массив из нечетных чисел:\n";
// Ввод размера массива:
size=Integer.parseInt(JOptionPane.showInputDialog("Укажите размер массива:"));
// Создание массива:
int[] nums=new int[size];
// Заполнение массива и формирование текста для вывода:
for(i=0;i<size;i++){
   nums[i]=2*i+1;
   text+=nums[i]+" ";
   }
// Вывод окна сообщения:
JOptionPane.showMessageDialog(null,text);
}
}
```

Программа работает так: выводится окно с полем для ввода размера массива. Введенное пользователем значение считывается и создается массив. Массив заполняется натуральными нечетными числами. Элементы массива последовательно, строчкой, выводятся на экран.

На рис. 5.1 показано диалоговое окно, которое отображается в начале выполнения программы и в поле которого нужно указать, из скольких элементов будет состоять массив.

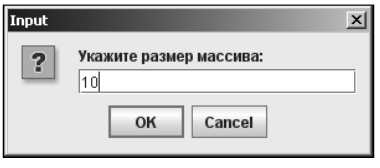


Рис. 5.1. Ввод значения для размера массива

В данном случае создаем массив из 10 элементов. После подтверждения ввода отображается новое окно, показанное на рис. 5.2.

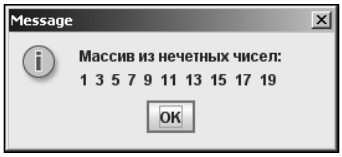


Рис. 5.2. Массив из нечетных чисел

Разберем программный код из листинга 5.1. Командой `int i,size` объявляются две целочисленные переменные: индексная переменная `i` для оператора цикла и переменная `size` для считывания в нее размера массива.

На заметку:

В данной программе мы вводим размер массива уже после того, как была запущена программа. Другими словами, на момент запуска программы неизвестно, из скольких элементов будет состоять массив. Это возможно благодаря тому, что массивы динамические. Если бы массивы были статическими, их размер пришлось бы указывать в виде константы, поскольку размер статического массива должен быть известен до запуска программы.

Командой `String text="Массив из нечетных чисел:\n"` задается начальное значение для текста, который будет отображаться во втором окне. После создания массива, в процессе заполнения его элементов, этот текст будет дополнен.

Отображение первого окна с полем ввода размера массива, считывание введенного пользователем значения и преобразование в число выполняется с помощью команды `size=Integer.parseInt(JOptionPane.showInputDialog("Укажите размер массива:"))`. После этого можно создавать массив, что и делаем командой `int[] nums=new int[size]`. Следующий после команды создания массива оператор цикла нужен для заполнения элементов массива нечетными числами и внесения соответствующих добавок к текстовой переменной `text` (в переменную последовательно записываются элементы массива).

В рамках оператора цикла индексная переменная `i` пробегает значения от `0` до `size-1` (в условии использовано строгое неравенство `i<size`, поскольку для массива размера `size` индекс последнего элемента равен `size-1`). Для фиксированного значения индексной переменной `i` значение элементу массива с таким индексом присваивается командой `nums[i]=2*i+1`. После этого значение элемента массива дописывается в текстовую переменную `text` командой `text+=nums[i]+" "`. Для большей наглядности между числами добавляется двойной пробел.

На заметку:

При создании числового массива в Java элементам массива автоматически присваиваются нулевые значения. Убедиться в этом просто: достаточно закомментировать (поставить в начале соответствующей строки двойную косую черту //) в приведенном выше в листинге 5.1 коде команду `nums[i]=2*i+1`. Этим самым блокируется команда, которой элементам массива присваиваются значения. По логике, это ошибочная ситуация, поскольку в следующей команде идет обращение к элементам массива, а значения этим элементами не присвоены. Тем не менее, программа будет работать: при выводе элементов массива окажется, что все они нулевые.

Наконец, после завершения оператора цикла командой `JOptionPane.showMessageDialog(null,text)` отображается окно сообщения со значениями элементов массива.

Присваивание и сравнение массивов

Способ определения массивов, используемый в Java и базирующийся на концепции переменных массива, имеет ряд особенностей. В этом разделе кратко остановимся на таких особенностях, и в частности, на процедуре присваивания массивов и сравнении массивов. Начнем с простого примера. Представим, что имеет место серия команд:

```
int[] first,second;
first=new int[10];
second=first;
```

В данном случае объявляется две переменные массива: `first` и `second`. Это переменные типа `int[]`. То есть каждая из них может ссылаться на одномерный массив из целых чисел. Командой `first=new int[10]` создается одномерный массив из десяти целых чисел, и ссылка на этот массив присваивается в качестве значения переменной `first`. Следующей командой `second=first` значение переменной `first` присваивается переменной `second`. Что произойдет?

Чтобы понять это, вспомним, что фактическим значением переменной массива является адрес области памяти, где реально расположен массив. Когда мы одной переменной массива присваиваем в качестве значения то, что записано в другую переменную массива, мы на самом деле делаем так, что обе эти переменные имеют одно и то же значение – адрес массива. Таким образом, обе переменные в результате будут ссылаться на один и тот же массив. Другими словами, у массива будет два названия. А можно сделать так, чтобы этих названий было три и больше. Правда, смысла в такой щедрости нет никакого. Например, к первому элементу массива можно обратиться как `first[0]`, а можно с помощью инструкции `second[0]`. Это будут не просто одинаковые элементы, а *один и тот же элемент*!

На описанной особенности переменных массива базируется процедура присваивания и сравнения массивов. Рассмотрим такой программный код:

```
int[] first,second;
first=new int[10];
second=new int[5];
second=first;
```

В отличие от предыдущего случая, здесь перед присваиванием переменных массива создается не только первый (команда `first=new int[10]`), но и второй массив (команда `second=new int[5]`). Первый массив состоит из 10 элементов, а второй массив состоит из 5 элементов. Теперь выполняем присваивание `second=first`. Мы уже знаем, что после выполнения такой команды переменные `first` и `second` будут ссылаться на один и тот же массив (из 10 элементов). Что произойдет со вторым массивом? Если на него нет других ссылок (то есть никакая другая переменная массива на него не ссылается), массив будет потерян безвозвратно.

На заметку:

При создании массивов один, как отмечалось, состоит из десяти элементов, а второй – из пяти. Выполнение команды `second=first` приводит к тому, что переменная `second`, которая раньше ссылалась на массив из пяти элементов, теперь ссылается на массив из десяти элементов. Это означает, что свойство `second.length` меняет свое значение с 5 на 10. Интересно и то, что совместимость (проявляется в возможности присваивания одной переменной значения другой переменной) переменных массива не зависит от количества элементов массива, а только от размерности и типа базовых элементов.

Крайнюю осторожность следует проявлять и при сравнении массивов. Очевидно, что в этом контексте имеет смысл говорить лишь об операциях == (равно) или != (не равно). Результатом выражения `first==second` является `true`, только если переменные `first` и `second` ссылаются на *один и тот же* массив. Даже если массивы одинаковые, но физически разные, результатом сравнения будет `false`. Чтобы сравнить два физически разных массива, необходимо выполнять поэлементную проверку.

Двумерные массивы

В Java можно создавать не только одномерные, но и многомерные массивы. В многомерном массиве индексация элементов выполняется с помощью нескольких индексов. Количество индексов определяет размерность массива. В Java размерность массива может быть практически любой, но на практике массивы большей размерности, чем два, используют крайне редко. На них и остановимся.

С точки зрения языка Java двумерный массив – это одномерный массив, элементами которого являются одномерные массивы. Точнее, элементами двумерного массива являются переменные одномерных массивов.

На заметку:

Эта же идея реализуется и в массивах более высоких размерностей. Например, трехмерный массив – это одномерный массив, элементами которого являются двумерные массивы, которые, в свою очередь, являются одномерными массивами, состоящими из одномерных массивов.

Как и в случае одномерного массива, следует различать переменную двумерного массива и непосредственно массив. При объявлении переменной двумерного массива указывается тип элементов двумерного массива, дважды пустые квадратные скобки и имя переменной массива. Например, так объявляется двумерный массив `NewArray` с элементами типа `double`:

```
double[][] NewArray;
```

При создании двумерного массива используется все тот же оператор `new`. После оператора указывается тип элементов двумерного массива и размер (количество элементов) по каждому из индексов. Для каждой размерности используется своя пара квадратных скобок. Так, если создается двумерный массив размера 3×4, то команда создания массива и присваивания ссылки на этот массив переменной массива будет выглядеть так:

```
NewArray=new double[3][4];
```

Команды объявления переменной массива и создания массива можно объединить в одну:

```
double[][] NewArray=new double[3][4];
```

На заметку:

Выше мы говорили, что двумерный массив – это одномерный массив, элементами которого являются переменные массива. Для конкретности будем говорить о двумерном массиве, который состоит из действительных чисел. То есть технически такой двумерный массив представляет собой одномерный массив, элементы которого – переменные массива типа `double`. Переменная массива типа `double` имеет тип `double[]`. С другой стороны, если мы хотим создать переменную массива с элементами определенного *типа*, то при объявлении переменной массива мы указываем этот *тип* с пустыми квадратными скобками. Если типом является `double[]`, то соответствующая переменная массива объявляется с идентификатором типа `double[][]`. Поэтому нет ничего удивительного в том, что переменную массива `NewArray`, кроме описанного выше способа, можно было объявить как `double[] NewArray[]` или как `double NewArray[][]`.

При обращении к элементу двумерного массива указываются имя массива (имя переменной массива) и два индекса – каждый в отдельных квадратных скобках. По каждой из размерностей *индексация элементов массива начинается с нуля*. Например, инструкцией `NewArray[0][0]` выполняется обращение к первому (начальному) элементу по каждому из индексов массива `NewArray`.

На заметку:

Если после имени двумерного массива указать только одну пару квадратных скобок с индексом, получим доступ к "строке" массива с соответствующим индексом – это один из тех одномерных массивов, из которых состоит двумерный массив. Вообще же двумерный массив удобно представлять в виде прямоугольной матрицы, заполненной элементами исходного типа. Первый индекс означает строку в матрице, а второй – столбец, на пересечении которых находится элемент. Правда, в Java можно создавать так называемые "рваные" массивы – у них в каждой строке разное количество элементов. Методы создания таких массивов обсуждаются в этой главе несколько позже.

В листинге 5.2 приведен пример программы с двумерным массивом.

Листинг 5.2. Заполнение двумерного массива случайными числами

```
import javax.swing.*;
public class NewSimpleArray{
public static void main(String[] args){
// Размеры массива:
int n,m;
// Текст для отображения в окне:
String text="Двумерный массив случайных чисел:";
// Считывание размеров массива:
n=Integer.parseInt(JOptionPane.showInputDialog("Строк в массиве:"));
m=Integer.parseInt(JOptionPane.showInputDialog("Столбцов в массиве:"));
// Создание массива:
int[][] nums=new int[n][m];
// Заполнение массива случайными числами:
for(int i=0;i<n;i++){
text+="\n";
for(int j=0;j<m;j++){
   nums[i][j]=(int)(10*Math.random());
   text+=nums[i][j]+" ";}
   }
// Вывод окна сообщения с элементами массива:
JOptionPane.showMessageDialog(null,text);}
}
```

Программой последовательно отображаются два окна ввода, в которых указываются размеры массива: в первом окне в поле ввода указываем количе-

ство строк, а во втором окне – количество столбцов. Введенные значения считываются, преобразуются из текстового формата в числовой и присваиваются в качестве значений переменным `n` и `m` соответственно. Текстовая переменная `text` инициализируется с начальной фразой, к которой затем будут дописаны значения элементов двумерного массива. Массив создается командой `int[][] nums=new int[n][m]`. Два вложенных оператора цикла нужны для заполнения массива случайными числами. Внешний цикл перебирает строки массива, а внутренний – элементы в строке. Первой командой `text+="\n"` в текст добавляется инструкция перехода к новой строке, чтобы каждая строка массива отображалась в новой текстовой строке. После этого запускается внутренний оператор цикла, в котором значение элементу массива `nums` с индексами `i` и `j` присваивается командой `nums[i][j]=(int)(10*Math.random())`. Здесь для генерирования случайных чисел использован статический метод `random()` класса `Math`. Метод при вызове генерирует случайное действительное число в диапазоне от 0 до 1 (строго меньше единицы). Мы хотим сгенерировать случайное целое число в диапазоне от 0 до 9 (включительно). Для этого случайное число в диапазоне от 0 до 1 (полученное вызовом метода `Math.random()`) умножаем на 10. Получаем случайное действительное число в диапазоне от 0 до 10 (строго меньшее десяти). Осталось отбросить дробную часть числа. С этой целью выполняем явное приведение типов, для чего добавляем интеллигентную инструкцию `(int)`. В итоге получается случайное число в диапазоне от 0 до 9 включительно.

На заметку:

В классе `Math` много других полезных методов, которые реализуют основные математические функции. Так что к помощи этого класса мы еще будем прибегать.

Командой `text+=nums[i][j]+" "` новоиспеченный элемент записывается в переменную `text` (три пробела между элементами добавляются для большей наглядности).

После выполнения вложенных операторов цикла двумерный массив заполнен и переменная `text` полностью "укомплектована". Окно с элементами массива отображается командой `JOptionPane.showMessageDialog(null,text)`. На рис. 5.3 показано, как может выглядеть результат заполнения массива размерами 5×12.

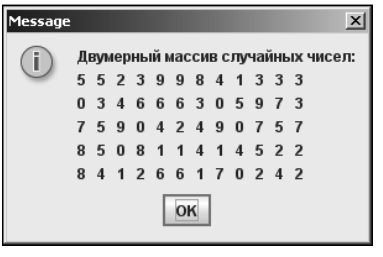


Рис. 5.3. Массив случайных чисел

У двумерного массива также есть свойство `length`. Этим свойством возвращается количество строк в двумерном массиве. Хотя на первый взгляд это может показаться, на самом деле все логично, если вспомнить, что двумерный массив – это массив (одномерный), состоящий из переменных массива. Количество таких переменных определяется размером по первому индексу, то есть количеством строк массива. Поэтому, например, если имеет место объявление `double[][] NewArray=new double[3][4]`, то инструкцией `NewArray.length` возвращается значение `3`. Можно узнать и количество столбцов в массиве, и тоже с помощью свойства `length`. При обращении к свойству в этом случае следует указать не имя массива, а ссылку на какую-то строку. Например, это может быть инструкция `NewArray[0].length`, которая возвращает значение `4`. Здесь `NewArray[0]` означает одномерный массив, который является начальным элементом внешнего одномерного массива переменных массива. Если массив прямоугольный, то в каждой строке одинаковое количество элементов, и для какой строки вычислять длину значения не имеет.

Инициализация массива

В этом и предыдущем примере (с одномерным массивом) заполнение массива значениями выполнялось с помощью операторов цикла. Однако такой номер не всегда проходит, поскольку для его эффективности необходимо, чтобы значения элементов массива подчинялись какой-то закономерности (или, например, были случайными числами), а это не всегда так. Другими словами, нередко приходится задавать значения элементов массива вручную, явно указывая значение для каждого элемента. Разумеется, можно сделать это "в лоб": выписать для каждого элемента команду присваивания. Но такой подход крайне неудобен. Можно поступить проще. А именно, при создании массива сразу указать список значений его элементов. Есть два способа сделать это, которые с точки зрения конечного результата практически не отличаются. Наиболее простой синтаксис такой: после объявления переменной массива указывается оператор присваивания и в фигурных скобках список значений элементов массива. Если это одномерный массив, то значения в фигурных скобках просто перечисляются через запятую. Для двумерного массива внешний список состоит из внутренних списков (каждый список выделяется фигурными скобками) – каждый внутренний список соответствует строке двумерного массива. Размер массива вычисляется автоматически в соответствии с количеством значений, переданных для инициализации. Ниже приведены примеры:

```
int[] nums={1,3,5,7};
double[][] x={{1,2,3,4},{5,6,7,8},{9,10,11,12}};
char[][] symbs={{'a'},{'b','c'},{'d','e','f'}};
```

Первой командой создается одномерный массив из целых чисел. Массив состоит из четырех элементов, которые последовательно получают такие значения: 1 (элемент `nums[0]`), 3 (элемент `nums[1]`), 5 (элемент `nums[2]`) и 7 (элемент `nums[3]`). Второй командой создается двумерный прямоугольный массив `x` действительных чисел `double` размера 3×4 (три строки по четыре элемента в каждой строке). Элементы последовательно заполнены натуральными числами: в первой строке числа от 1 до 4, во второй – от 5 до 8 и так далее. Небезынтересной представляется последняя, третья команда. Этой командой создается "рваный" массив: у него в каждой из трех строк разное количество элементов. Это символьный (тип `char`) массив `symbs`, у него первая строка состоит всего из одного элемента (элемент `symbs[0][0]`, равный `'a'`), вторая – из двух (элемент `symbs[1][0]`, равный `'b'`, и элемент `symbs[1][1]`, равный `'c'`) и третья – из трех (элемент `symbs[2][0]`, равный `'d'`, элемент `symbs[2][1]`, равный `'e'`, и элемент `symbs[2][2]`, равный `'f'`). К таким массивам мы еще вернемся.

Другой способ инициализации подразумевает, что при выделении места в памяти под массив с помощью оператора `new` в конце соответствующей команды указывается список значений для инициализации. При этом в инструкции с оператором `new` в квадратных скобках (сколько бы их там не было) количество элементов не указывается. Ниже приведены команды объявления массивов, альтернативные тем, что рассматривались в этом разделе ранее:

```
int[] nums=new int[]{1,3,5,7};
double[][] x=new double[][]{{1,2,3,4},{5,6,7,8},{9,10,11,12}};
char[][] symbs=new char[][]{{'a'},{'b','c'},{'d','e','f'}};
```

Результат один и тот же. В следующем разделе приведены некоторые примеры использования массивов.

Работа с массивами

В программе в листинге 5.3 создается "рваный" символьный массив. Размеры массива (количество строк и количество элементов в каждой строке) определяются случайным образом. Сформированный таким образом массив заполняется последовательно буквами латинского алфавита, а результат выводится на экран через окно сообщения.

Листинг 5.3. "Рваный" символьный массив

```
import javax.swing.*;
public class CharArray{
   public static void main(String[] args){
   // Размер массива (количество строк) - случайное число от 2 до 4:
```

```
   int size=2+(int)(Math.random()*3);
   // Переменная для записи количества элементов строке массива:
   int n;
   // Начальная буква для заполнения массива:
   char s='a';
   // Текст для вывода на экран в окне сообщения:
   String text="\"Рваный\" символьный массив:";
   // Создание массива - задано количество строк:
   char[][] symbs=new char[size][];
   // Добавление строк в массив:
   for(int k=0;k<size;k++){
      // Случайное число от 1 до 8:
      n=1+(int)(Math.random()*8);
      // Создание строки случайной длины:
      symbs[k]=new char[n];}
   // Заполнение массива буквами:
   for(int i=0;i<symbs.length;i++){
   // Переход на новую строку и символ вертикальной черты:
   text+="\n| ";
   // Переменная j для разных значений i имеет разные ограничения:
   for(int j=0;j<symbs[i].length;j++){
   // Заполнение элемента символом:
   symbs[i][j]=s;
   // Следующий символ:
   s++;
   // Добавление записи в текст:
   text+=symbs[i][j]+" | ";}
   }
   // Вывод сообщения на экран:
   JOptionPane.showMessageDialog(null,text);}
}
```

На рис. 5.4 показан возможный результат выполнения программы.

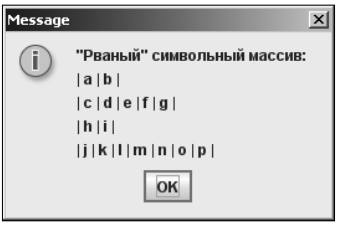


Рис. 5.4. "Рваный" символьный массив случайного размера

Командой `int size=2+(int)(Math.random()*3)` определяется случайное число в диапазоне от 2 до 4 включительно, которое определяет количество строк в создаваемом массиве. Начальный текст для последующего отображения в окне сообщения создается командой

`String text="\"Рваный\" символьный массив:"`. Обращаем внимание, что символ двойных кавычек в текст добавляется с обратной косой чертой (то есть, чтобы в текст добавить двойные кавычки **"**, перед ними нужно поставить косую черту \, получится **"**).

Массив создается в несколько этапов. Сначала командой `char[] [] symbs=new char[size][]` формируется остов: задаются параметры одномерного массива из переменных массива. Затем каждая такая переменная массива будет связана с одномерным символьным массивом. Особенность команды состоит в том, что при создании массива размер по второму индексу не указывается – каждая строка массива будет иметь свой уникальный размер. Вся уникальность строк реализуется в рамках оператора цикла. Там индексная переменная `k` пробегает значения от `0` и до конца внешнего массива (условие `k<size`). Командой `n=1+(int)(Math.random()*8)` генерируется случайное число от `1` до `8` включительно и присваивается в качестве значения переменной `n`. Это количество элементов в строке с индексом `k`. Ссылка на эту строку присваивается переменной `symbs[k]`. А именно, используем команду `symbs[k]=new char[n]`, которой создается одномерный символьный массив соответствующего размера и ссылка на него записывается в переменную `symbs[k]`. В результате после выполнения оператора цикла массив создан, и его остается лишь заполнить. Для этого запускается двойной, вложенный оператор цикла.

Индексная переменная `i` внешнего оператора цикла пробегает значения от `0` до `symbs.length-1` (количество строк в массиве минус один). Индексная переменная `j` внутреннего оператора цикла пробегает значения от `0` до `symbs[i].length-1` (уменьшенное на единицу количество элементов в строке с индексом `i`). Но перед запуском внутреннего оператора цикла выполняется команда `text+="\n|  "`, которой в предполагаемый для вывода текст добавляется инструкция перехода к новой строке и символ вертикальной черты, используемый в качестве разделителя при отображении элементов массива в окне сообщения.

Командой `symbs[i][j]=s` элементу массива присваивается текущее буквенное значение переменной `s`, после чего значение этой переменной с помощью команды `s++` "увеличивается на единицу" – переменная `s` получит в качестве значения следующую букву алфавита.

📖 На заметку:

Операция инкремента или декремента в отношении переменных типа `char` сводится соответственно к увеличению или уменьшению на единицу кода символа, записанного в переменную. Для букв это означает перемещение вперед или назад на одну позицию в алфавите.

Командой `text+=symbs[i][j]+" | "` в текстовую переменную `text` добавляется новый символ из массива, плюс окруженная пробелами вертикальная черта. Традиционно, сообщение выводится командой `JOptionPane.showMessageDialog(null,text)`.

В листинге 5.4 представлена программа, в которой выполняется сортировка изначально неупорядоченного, заполненного случайным образом одномерного символьного массива *методом пузырька*.

На заметку:

Метод пузырька применяется для сортировки массивов, элементы которых допускают операцию сравнения значений. Это могут быть числовые или символьные массивы. Для символьных массивов разумно полагать, что при сравнении букв (символов) сравниваются их коды. Суть метода состоит в том, что последовательно перебираются все элементы массива, и соседние элементы сравниваются. При сортировке элементов по возрастанию, если элемент слева больше элемента справа, они меняются местами. При сортировке по убыванию, элементы меняются местами, если элемент справа больше элемента слева.

За один перебор элементов массива по крайней мере один элемент (самый большой или самый маленький) окажется в нужной позиции. За второй перебор элементов уже два элемента будут на правильной позиции и так далее.

Листинг 5.4. Сортировка символьного массива методом пузырька

```
import javax.swing.*;
public class CharArraySorting{
   public static void main(String[] args) {
   // Текст для окна сообщения:
   String text="Исходный массив:\n";
   // Размер массива и индексные переменные:
   int size,i,j;
   // Получение размера массива:
   size=Integer.parseInt(JOptionPane.showInputDialog("Размер массива:"));
   // Создание символьного массива:
   char[] symbs=new char[size];
   // Начальная буква:
   char s='a';
   // Заполнение массива (случайным образом):
   for(i=0;i<size;i++){
   symbs[i]=(char)(s+(byte)(Math.random()*26));
   text+=symbs[i]+" ";}
   text+="\nПосле сортировки:\n";
   // Сортировка массива:
   for(i=0;i<size;i++){
   for(j=0;j<size-i-1;j++){
   if(symbs[j]>symbs[j+1]){
   s=symbs[j+1];
```

```
    symbs[j+1]=symbs[j];
    symbs[j]=s;}}
    }
    // Запись в текстовую переменную элементов массива после сортировки:
    for(i=0;i<size;i++){
    text+=symbs[i]+" ";}
    // Отображение окна сообщения:
    JOptionPane.showMessageDialog(null,text);
    }
}
```

Программа работает так: выводится окно с полем ввода, в котором следует указать количество элементов в массиве. Далее введенное пользователем число считывается, и создается одномерный символьный массив `symbs` с соответствующим количеством элементов. Для заполнения массива выполняется оператор цикла, в котором индексная переменная `i` перебирает все элементы массива, а значение элементам массива присваивается командой `symbs[i]=(char)(s+(byte)(Math.random()*26))`. Символьная переменная `s` предварительно, до начала оператора цикла получила значение `'a'`. Этот код, пожалуй, требует некоторых пояснений. Так, командой `(byte)(Math.random()*26)` генерируется случайное целое число в диапазоне от 0 до 25 включительно. Полученное значение складывается с символьной переменной `s`. Благодаря автоматическому приведению типов такая операция выполняется на уровне сложения кодов. Чтобы полученный код преобразовать в символ, используем инструкцию явного приведения типов `(char)`. Таким образом, получаем случайным образом сгенерированную букву, чего мы и добивались.

Командой `text+=symbs[i]+" "` значение элемента несортированного массива дописывается в переменную `text`. После завершения оператора цикла командой `text+="\nПосле сортировки:\n"` в этот текст добавляется заглавная фраза и инструкция перехода к новой строке. Самое время приступить к сортировке массива.

Сортировка выполняется по возрастанию значений элементов с помощью двух вложенных операторов цикла. Внешний цикл перебирает "заходы": каждый заход – это один полный перебор элементов массива со сравнением и перестановкой элементов, если необходимо. Внутренний цикл нужен для непосредственного перебора элементов. Здесь принято во внимание, что после каждого "захода" перебирать элементов нужно все меньше и меньше: при первом переборе сравниваются все элементы массива, при втором переборе сравниваются все элементы, кроме последнего – он и так самый "большой", при третьем переборе нет смысла проверять значения последних двух элементов и так далее. Поэтому для индексной переменной внутреннего цикла указано условие `j<size-i-1`, зависящее от индексной переменной внешнего оператора цикла.

Основу процедуры сравнения составляет условный оператор: если элемент слева больше элемента справа, выполняется их перестановка.

На заметку:

Хотя элементы массива – символы, для них применимы операции сравнения (в частности, *больше* и *меньше*). Несложно догадаться, что в этом случае сравниваются числовые коды символов. Кроме того, при перестановке для "технических" нужд использовалась переменная `s`. Свое основное предназначение она выполнила (когда генерировались значения элементов массива), а здесь мы ее задействовали как контейнер для временного хранения значения одного из элементов при перестановке.

Еще один оператор цикла потребовался для того, чтобы добавить в текстовую переменную `text` последовательность символов из отсортированного массива. После этого результат отображается в окне сообщения, которое может иметь, например, вид, как на рис. 5.5.

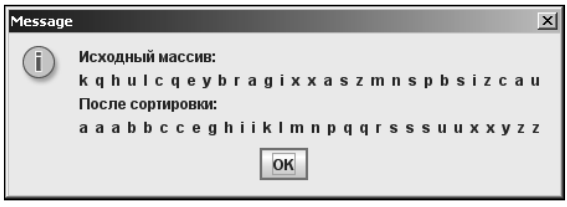


Рис. 5.5. Сортировка символьного массива методом пузырька

В данном случае создавался массив из 30 элементов.

Оператор цикла `for` для перебора элементов массива

Как отмечалось в предыдущей главе, существует особая форма оператора `for`, которая позволяет перебирать элементы массива без использования индексной переменной. Синтаксис вызова этой формы оператора следующий:

```
for(тип_переменной переменная: массив){
   команды
   }
```

Здесь фактически речь идет об упрощенной форме оператора цикла `for`. В круглых скобках после ключевого слова `for` указывается тип локальной переменной, имени этой локальной переменной и, через двоеточие, имя массива. Тип элементов массива и тип локальной переменной совпадают. Затем в фигурных скобках следует блок команд. Локальная переменная по очереди принимает значения элементов массива, и для каждого такого значения ло-

кальной переменной выполняется блок команд в фигурных скобках. Пример использования этой формы оператора цикла приведен в листинге 5.5.

Листинг 5.5. Упрощенная форма оператора цикла

```
public class ForEachDemo{
public static void main(String[] args){
// Исходный массив для отображения:
int[] a=new int[]{1,2,4,7,3,1,8,9,2};
// Упрощенная форма оператора цикла:
for(int s:a){
System.out.print(s+" ");}
System.out.println();}}
```

В результате в консольном окне появится последовательность чисел:

```
1 2 4 7 3 1 8 9 2
```

Выясним, что же происходит при выполнении программы. В главном методе программы командой `int[] a=new int[]{1,2,4,7,3,1,8,9,2}` создается и инициализируется целочисленный массив. В операторе цикла for объявлена локальная целочисленная переменная `s`, а после нее через двоеточие указывается имя массива, то есть `a`. Это означает, что переменная `s` по очереди "перебирает" элементы массива `a`, принимая значение этих элементов. В теле оператора цикла выполняется команда `System.out.print(s+" ")`, которой на экран выводится (в одну строку) текущее значение переменной `s` и пробел. После окончания оператора цикла командой `System.out.println()` выполняется переход к новой строке.

На заметку:

Следует понимать, что упрощенная форма оператора цикла `for` не заменяет ее полную форму и подходит только для решения задач определенного типа.

Резюме

1. Массивы в Java создаются динамически с помощью оператора `new`. Ссылка на массив записывается в переменную массива.
2. Для объявления переменной одномерного массива указывают базовый тип элементов, пустые квадратные скобки и имя массива (переменную массива).
3. Для создания одномерного массива после оператора `new` указывается тип элементов массива и в квадратных скобках - его размер. Оператором `new` возвращается ссылка на созданный массив. Эта ссылка может быть записана в переменную массива.

4. Индексация элементов массива всегда начинается с нуля.
5. Для доступа к элементу одномерного массива после имени массива в квадратных скобках указывается индекс элемента в массиве.
6. Для определения размера массива используют свойство `length`.
7. Двумерный массив - это массив, элементами которого являются переменные одномерных массивов.
8. При объявлении двумерных массивов используются две пары квадратных скобок.

Глава 6
Классы и объекты

Восток — дело тонкое...
(Из к/ф "Белое солнце пустыни")

Классы мы уже использовали, что называется, по полной – без них не обошлась ни одна наша программа. Тем не менее читатель, если он до этого не имел дела с объектно-ориентированным программированием, скорее всего очень смутно представляет, что же такое *класс*. Здесь мы посмотрим проблеме в глаза и постараемся устранить любое недопонимание. Стержневым моментом является четкое понимание фундаментальных различий между *классом* и *объектом*. С них и начнем.

Классы и объекты

Как известно, лучше любой теории – хороший пример, а лучше хорошего примера – только простая и понятная аналогия. Так и поступим. Представим, что программа, которую мы хотим написать, – это большой и красивый дом. Мы этот дом строим. Мы точно знаем, каким он должен быть, но не знаем, из чего и как его построить. Какие есть варианты?

Разумеется, можно решать проблему традиционным методом: заготовить кирпичей, рам, дверных проемов, нанять каменщиков, которые будут по кирпичику выстраивать здание. В этом случае есть все шансы рассчитывать на то, что к моменту окончания строительства дом морально устареет и придется строить новый. Кроме того, чем больше здание, тем больше народу его будет строить. Кто-то кладет кирпичи красиво и ровно, а кто-то – косо и криво. От одной неправильно положенной стенки может завалиться весь дом. Нам это надо? Ответ очевиден.

Другой подход такой: заказать готовые блоки, скомпонованные из плит, в которые уже вставлены окна, дверные проемы и прочие полезные вещи. Для здания не очень экзотической конструкции таких блоков будет всего несколько типов, причем, скорее всего, достаточно стандартных. Закупаем (или создаем сами – в нашем случае это больше подходит) блоки в нужном количестве, нанимаем высококлассных монтажников (их надо будет задействовать в меньшем количестве, чем каменщиков, поэтому зарплату им можно заплатить более высокую) и ждем, пока вся эта мозаика соберется в красивую картину в виде здания. Здесь тоже есть свои проблемы, но они ни в какое сравнение не идут с теми проблемами, что возникают при первом сценарии постройки здания.

Несложно догадаться, что наиболее тонкий момент во втором сценарии связан с изготовлением блоков, которые пойдут на постройку здания. Во-первых, следует определить, какие именно блоки нужны. Во-вторых, необходимо знать, в каком количестве нужны эти блоки.

Первый подход иллюстрирует традиционный способ программирования, который называется *процедурным* или *структурным*. Этот способ хорошо зарекомендовал себя при написании относительно небольших или в каком-то смысле уникальных программ. Небольшие или нестандартные здания тоже проще выкладывать из кирпичиков – блочный метод здесь может не пройти. Второй подход иллюстрирует *объектно-ориентированное программирование* (сокращенно *ООП*). Этот метод используется при написании больших программ, а особенно он хорош, если большая программа еще и стандартная. Понятно, что если мы строим по типовому проекту большое здание, удобнее всего его сложить, как конструктор, из блоков.

Где же здесь классы и объекты? Классы появляются, когда мы определяем, какого типа *блоки* нам нужны. Например, мы решили, что нам нужен блок с одним дверным проемом и двумя окнами. У нас такого блока еще нет, есть только его *проект*, или *эскиз*. Такой проект или эскиз – как раз и есть *класс*. А вот когда мы по этому эскизу создаем реальный блок, появляется *объект*. По одному эскизу мы можем сделать один блок, два блока, сто блоков или ни одного. Все блоки, выполненные по одному эскизу, с одной стороны, похожи, поскольку все их штамповали по стандартной схеме (точнее, по эскизу), но в то же время все это физически разные объекты. Каждый из них может использоваться по-разному: один покрасили, другой выбросили, третий разбили. Так и программные объекты. Объекты, или экземпляры класса, создаются на основе одного шаблона (то есть класса), но каждый из них уникален.

На заметку:

В любом объектно-ориентированном языке программирования реализуется три базовых принципа ООП: *инкапсуляция*, *полиморфизм* и *наследование*. Структурирование программы через систему классов и объектов является реализацией механизма инкапсуляции. В основу механизма инкапсуляции положена идея об объединении данных и программного кода для обработки этих данных в отдельные структурные единицы - объекты.

Благодаря полиморфизму существенно сокращается количество методов (функций) для обработки данных, поскольку полиморфизм подразумевает использование общих методов для решения однотипных задач и выполнения похожих действий. В Java полиморфизм реализуется через систему перегрузки и переопределения методов. Об этом мы еще поговорим, как и о наследовании – механизме, который позволяет создавать одни классы на основе других, уже существующих классов.

Когда мы пишем объектно-ориентированную программу, как блоки для строительства здания, описываем классы, и на их основе в нужном количестве создаем блоки – объекты, из которых программа строится, как дом из блоков.

Объявление класса и создание объекта

С небес опускаемся на землю и сразу же задаемся простым и прозаичным вопросом: как создать класс? Точнее, как его объявить? Некоторые намеки уже были сделаны, тем более что формально класс мы объявляли, и не раз. Здесь только формализуем основные моменты.

Во-первых, в программе может (и так обычно и есть) использоваться больше чем один класс. Более того, если классов больше одного (а меньше одного не бывает), то обычно еще и создаются объекты.

На заметку:

Хотя это противоречит всему вышесказанному, класс, в принципе, может использоваться без создания объектов. Для этого класс должен содержать статические члены (поля и/или методы). Так, например, обстоят дела с главным методом программы `main()` – метод статический, поэтому вызывается прямо из класса, без создания объекта. Статические поля и методы обсуждаются несколько позже.

Но мы жизнь будем усложнять постепенно, и пока обсудим исключительно классы: как они объявляются и, самое главное, что в них объявляется.

Объявление класса начинается с ключевого слова `class`. После ключевого слова `class` указывается имя класса. Затем в фигурных скобках вводится код тела класса, которым, собственно, и реализуется класс. Синтаксис объявления класса имеет такой вид:

```
class имя_класса{
код класса
}
```

На заметку:

В принципе, заголовок объявления класса может иметь несколько иной вид. В классах, которые мы уже использовали, помимо перечисленных атрибутов используется ключевое слово `public`. Дальше мы узнаем, что дополнительные конструкции добавляются в заголовок класса при наследовании классов и реализации интерфейсов.

Чтобы написать код класса в фигурных скобках, нужно знать, что там пишут в принципе. Если характеризовать ситуацию обще, то там описывают поля и методы. Поля – это переменные, которые описываются в пределах

класса. Методы – это функции, которые описываются в пределах класса. Собственно, все.

На заметку:

Как отмечалось выше, в классе объединены данные и код для их обработки. За данные отвечают поля класса, а за код для их обработки – методы.

Напомним, что *переменной* называется именованная область памяти, к которой в программе можно обращаться по этому имени. *Функцией* называется именованный блок программного кода, который можно вызывать в программе по имени.

Поля описываются точно так же, как переменные: в теле класса указывается тип поля (тип переменной) и имя. Может также указываться спецификатор доступа, но эта тема описывается позже. Несколько сложнее описываются методы. Методы пока трогать не будем, а рассмотрим класс, содержащий только поля. Вот пример такого класса:

```
class MyClass{
int num;
char symb;
String text;
}
```

Здесь объявляется класс с названием `MyClass`, который содержит три поля: поле `num` типа `int`, поле `symb` типа `char` и поле `text` типа `String`.

Поскольку класс уже есть, на его основе можно создать объект. Процесс создания объекта класса условно разделяют на два этапа. Первый этап связан с созданием *объектной переменной* – переменной, которая содержит в качестве значения адрес объекта. Второй этап подразумевает непосредственно создание объекта и определение ссылки на этот объект. Создаются объекты динамически с использованием оператора `new`.

На заметку:

Процесс создания объекта достаточно сильно напоминает создание массива. Причем напоминает не случайно. Синтаксис другой, но идея та же.

Объектная переменная объявляется практически так же, как и обычная переменная базового типа: в качестве типа переменной указывается имя класса, а затем непосредственно имя объектной переменной.

Объект создается с помощью команды вида `new` *имя_класса*`()`, то есть после оператора `new` указывается имя класса с пустыми круглыми скобками (пока что пустыми).

На заметку:

После оператора `new` указывается конструктор класса. Конструктор класса – это метод, который вызывается при создании объекта класса. Далее мы научимся описывать и использовать конструкторы. Хотя в данном случае никакой конструктор мы не описывали, всегда существует *конструктор по умолчанию*. Пока же нам достаточно знать, что конструктор – это метод (а методы указываются с круглыми скобками, в которых могут указываться аргументы), а имя конструктора всегда совпадает с именем класса.

В качестве результата оператором `new` возвращается ссылка на созданный объект. Таким образом, чтобы создать объектную переменную с названием `obj` класса `MyClass`, создать объект этого класса и записать ссылку на объект в переменную, используем такие команды:

Обе команды можно совместить, и вместо них использовать одну:

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

После того как объект создан и ссылка на него записана в объектную переменную, этот объект можно использовать. Доступ к объекту получают через соответствующую объектную переменную. Собственно, использование объекта подразумевает обращение к полям и методам объекта. Методов у нашего объекта нет, а вот полей – сколько угодно (точнее, целых три). При обращении к полю объекта необходимо явно указывать, к какому объекту относится поле.

На заметку:

Если для данного класса создано несколько объектов, то у каждого из этих объектов один и тот же набор полей. Поэтому само по себе имя поля не дает однозначного ответа на вопрос, о поле какого объекта идет речь. Для этого и нужно указывать явно, к какому объекту относится поле.

Синтаксис обращения к полю объекта такой: имя объекта, точка и имя поля. Это так называемый *точечный синтаксис*. Например, чтобы обратиться к полю `num` объекта `obj`, используем инструкцию `obj.num`. В листинге 6.1 приведен пример программы, в которой описывается, кроме основного, еще один класс, создается объект этого класса, заполняются поля объекта, и информация из полей объекта считывается и выводится на экран в окне сообщения.

Листинг 6.1. Создание класса и объекта

```
import javax.swing.*;
// Описание класса:
class MyClass{
int num;
```

```
char symb;
String text;
}
// Главный класс программы:
public class ClassDemo{
public static void main(String[] args){
   MyClass obj=new MyClass();
   // Заполнение полей объекта:
   obj.num=12;
   obj.symb='A';
   obj.text="Текстовое поле";
   // Текст для отображения в окне сообщения:
   String str="Значения полей объекта:\n";
   str+="num="+obj.num+"\n";
   str+="symb="+obj.symb+"\n";
   str+="text="+obj.text;
   // Отображение окна:
   JOptionPane.showMessageDialog(null,str);}
}
```

Окно сообщения, которое открывается в результате выполнения программы, представлено на рис. 6.1.

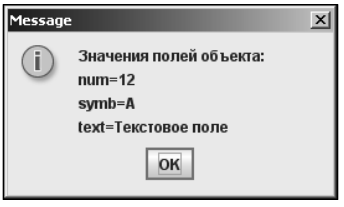


Рис. 6.1. Результат выполнения программы с классом

Программный код состоит и двух классов. Первый класс `MyClass`, думается, комментариев не требует – он описывался выше. У этого класса три поля: целочисленное, символьное и текстовое. В классе `ClassDemo` описан главный метод программы, в котором и происходит все самое интересное. А именно, командой `MyClass obj=new MyClass()` создается объект `obj` класса `MyClass`.

На заметку:

Здесь и далее, если это не будет вызывать недоразумений, под объектами будем подразумевать соответствующие объектные переменные.

Объект создан, у него есть поля, но этим полям нужно присвоить значения. Значения полям объекта `obj` присваиваются командами `obj.num=12`, `obj.symb='A'` и `obj.text="Текстовое поле"`. Особой интриги здесь

нет: поля объекта – это те же переменные, только при обращении к ним нужно указывать имя объекта. После того, как поля заполнены, на основе их значений формируется текстовая строка для отображения в окне сообщения. Как и при присваивании значения полю, считывание его значения выполняется через точечный синтаксис: перед именем поля через точку указывается имя объекта.

Методы

Метод представляет собой именованный блок программного кода, который выполняется при обращении к методу. В Java методы описываются в классах. **Описание метода состоит из** *сигнатуры* (заголовка, или шапки) **и тела метода.** У сигнатуры метода есть определенная структура, которой нужно придерживаться. Начинается описание метода с ключевого слова, которое обозначает тип результата, возвращаемого методом. Результатом метода может быть значение как базового, так и ссылочного типа (то есть объект). Если метод не возвращает результат (такое тоже допустимо), в качестве типа результата указывается ключевое слово `void`.

На заметку:

С ключевым словом `void` мы уже сталкивались – оно указывается в сигнатуре главного метода программы `main()`. Этот метод не возвращает результат.

После этого указывается имя метода и в круглых скобках список аргументов метода. Аргументы указываются через запятую вместе с идентификатором типа. Тип указывается для каждого аргумента персонально, даже если несколько аргументов относятся к одному типу. Как и результат метода, аргумент метода может быть всем, чем угодно. Тело метода указывается сразу после сигнатуры в фигурных скобках:

```
тип_результата имя_метода(аргументы){
   тело метода
   }
```

Вот пример описания метода:

```
double func(double x){
   double t;
   t=2*x+1;
   return t;
   }
```

Методом запрограммирована функция $f(x) = 2x + 1$. Метод называется `func()`, и у него один аргумент типа `double`. Название аргумента в описании метода является формальным – называть можно как угодно, главное, чтобы

название аргумента было уникальным в пределах метода и не совпадало ни с одним из ключевых слов Java. Методом возвращается значение типа `double`.

В теле метода объявляется локальная (локальная – потому что доступна только в пределах метода) переменная `t` тип `double`. Переменной `t` значение присваивается командой `t=2*x+1`. Значение переменной `t` возвращается в качестве результата метода. В последнем случае используется инструкция `return t`.

На заметку:

Инструкцией `return` завершается выполнение метода. Если после этой инструкции указано значение, то оно возвращается в качестве результата метода. Что касается видимости переменных, то переменная доступна (и ее имя должно быть уникальным) в пределах того программного блока, в котором она объявлена. В отношении метода это означает, что объявленная в теле класса (или в списке аргументов метода) переменная доступна только в пределах этого метода. Вообще же напомним, что блоки выделяются парами фигурных скобок.

В качестве иллюстрации работы с методами в классе доработаем рассмотренный выше пример так, чтобы класс, который мы создали, содержал кроме полей еще и метод. Обратимся к коду в листинге 6.2.

Листинг 6.2. Класс с методом

```
import javax.swing.*;
// Описание класса:
class MyClass{
   int num;
   char symb;
   String text;
   // Метод класса:
String showText(){
// Локальная текстовая переменная:
String str="Значения полей объекта:\n";
// Формирование результата (значения локальной переменной):
str+="num="+num+"\n";
str+="symb="+symb+"\n";
str+="text="+text;
// Метод возвращает значение:
return str;
}
}
// Главный класс программы:
public class ClassDemo2{
   public static void main(String[] args){
      MyClass obj=new MyClass();
```

```
        // Заполнение полей объекта:
        obj.num=12;
        obj.symb='A';
        obj.text="Текстовое поле";
        // Отображение окна:
        JOptionPane.showMessageDialog(null,obj.showText());}
    }
```

Программа работает практически так же, как и в предыдущем случае, но код программы организован несколько иначе: процесс формирования текста для отображения в диалоговом окне реализуется в виде метода `showText()` класса `MyClass`. У метода нет аргументов, и в качестве значения методом возвращается текстовая величина. Об этом говорит ключевое слово `String` в сигнатуре метода. В самом методе объявляется локальная текстовая переменная `str` с начальным значением (инструкция `String str="Значения полей объекта:\n"`). Значение этой переменной уточняется командами `str+="num="+num+"\n"`, `str+="symb="+symb+"\n"` и `str+="text="+text`. Хотя такого типа команды нам уже встречались, особенность этих состоит в том, что в них при обращении к полям `num`, `symb` и `text` имя объекта не указывается. Дело в том, что поскольку соответствующие команды размещены внутри кода метода класса, в котором эти поля объявлены, автоматически подразумевается, что имя поля является обращением к полю того объекта, из которого вызывается метод. Другими словами, если в теле метода `showText()` есть инструкция `num`, то при выполнении метода `showText()` будет выполняться обращение к полю `num` того объекта, из которого вызывается метод.

После того, как переменная `str` получает свое окончательное значение, она инструкцией `return str` возвращается в качестве результата метода.

Изменения в методе `main()` в классе `ClassDemo2` (по сравнению с предыдущим примером) сводятся к тому, что отсутствует фрагмент кода, связанный с формированием текстовой переменной для отображения в окне сообщения, а в команде отображения окна сообщения вторым аргументом указана не текстовая переменная, а инструкция вызова метода `showText()` из того объекта, который ранее создан и полям которого присвоены значения. В частности, окно отображается командой `JOptionPane.showMessageDialog(null,obj.showText())`. Нас интересует второй аргумент `obj.showText()` метода `showMessageDialog()`. В данном случае из объекта `obj` вызывается метод `showText()`. Метод вызывается так же, как и выполняется обращение к полю: после имени объекта указывается точка и имя метода. Если методу передаются аргументы, они указываются в круглых скобках через запятую. Но даже если аргументов нет, то пустые круглые скобки после имени метода все равно указываются.

Поскольку в данном случае результатом метода `showText()` является текстовое значение, то его вполне можно передать вторым аргументом методу `showMessageDialog()`. Важно то, что инструкцией `obj.showText()` возвращается текст, сформированный на основе значений полей объекта `obj`. Если бы, например, в программе (методе `main()`) был еще один объект (для определенности назовем его `newObj`), то инструкцией `newObj.showText()` возвращается текст, сформированный на основе значений полей объекта `newObj`.

Работу с несколькими объектами и несколькими методами в пределах одного класса иллюстрирует пример из листинга 6.3.

Листинг 6.3. Несколько объектов

```
import javax.swing.*;
// Описание класса:
class MyClass{
   int num;
   char symb;
   String text;
   // Метод класса для отображения сообщения:
void showText(){
// Локальная текстовая переменная:
String str="Значения полей объекта:\n";
// Формирование результата (значения локальной переменной):
str+="num="+num+"\n";
str+="symb="+symb+"\n";
str+="text="+text;
// Метод отображает окно сообщения:
JOptionPane.showMessageDialog(null,str);}
// Метод класса для присваивания значений полям:
void set(int n,char s,String str){
   // Заполнение полей:
   num=n;
   symb=s;
   text=str;}
   }
// Главный класс программы:
public class ClassDemo3{
   public static void main(String[] args){
   // Создание объектов:
   MyClass obj1=new MyClass();
   MyClass obj2=new MyClass();
   // Заполнение полей объектов:
   obj1.set(10,'A',"Всем привет!");
   obj2.set(200,'b',"Еще раз, привет!");
```

```
// Отображение окон:
obj1.showText();
obj2.showText();}
}
```

В результате выполнения программы последовательно отображаются два окна, которые представлены на рис. 6.2 и рис. 6.3.

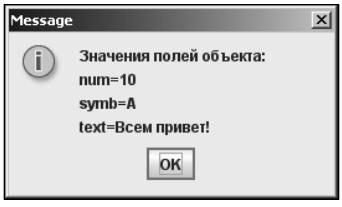


Рис. 6.2. Первое окно, отображаемое при выполнении программы

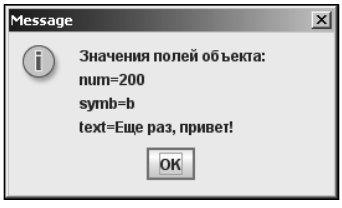


Рис. 6.3. Второе окно, отображаемое при выполнении программы

Новый вариант класса `MyClass` стоит некоторых пояснений. Тот фрагмент кода класса, который касается объявления полей, остался без изменений. Изменился код метода `showText()` и добавлен еще один метод `set()`. Рассмотрим последовательно оба эти метода.

В очередном варианте метод `showText()` не возвращает результат (в сигнатуре метода указано ключевое слово `void`). В методе, как и раньше, определяется локальная переменная `str`, которой присваивается (в несколько этапов) нужное значение. Однако если в предыдущей версии метода это значение возвращалось как результат, то теперь командой `JOptionPane.showMessageDialog(null,str)` на экране сразу отображается окно сообщения с соответствующим текстом. Таким образом, если раньше метод `showText()` только "вычислял" нужную текстовую строку для отображения, то теперь при вызове метода отображается окно с такой текстовой строкой.

Метод `set()` является новичком в классе `MyClass`. Метод имеет три аргумента и предназначен для заполнения полей объектов класса. Первый аргумент метода `set()` – значение типа `int`. Это значение присваивается полю `num`. Полю `symb` присваивается значение второго аргумента, переданного методу `set()`. Третий текстовый аргумент метода определяет значе-

ние текстового поля text. В теле метода выполняются соответствующие присваивания.

В главном методе main() программы в классе ClassDemo3 командами MyClass obj1=new MyClass() и MyClass obj2=new MyClass() создаются два объекта (obj1 и obj2) класса MyClass. Это разные объекты, хотя и принадлежат одному классу. Заполнение полей объектов (присваивание полям объектов значений) выполняется с помощью метода set(), который последовательно вызывается из каждого из объектов (с разным набором аргументов). Поля объекта obj1 заполняются вызовом команды obj1.set(10,'A',"Всем привет!"). В результате поле num этого объекта получает значение 10, поле symb получает значение 'A', а текстовое поле text получает значение "Всем привет!".

Значения полям объекта obj2 присваиваются с помощью команды obj2.set(200,'b',"Еще раз, привет!"). Поле num объекта obj2 в результате равно 200, значение поля symb этого объекта равно 'b', а текстовое поле содержит значение "Еще раз, привет!".

На заметку:

Мы использовали в классе текстовое поле (поле типа String). Как уже отмечалось, String – это класс, предназначенный для реализации текста. Другими словами, упомянутое текстовое поле – это объектная переменная класса String. Как и любая объектная переменная, она в качестве значения содержит ссылку на текст. Но поскольку при обращении к этой переменной автоматически возвращается текст, на который ссылается переменная, мы пока что не будем делать различий между текстовой объектной переменной и текстом, на который она ссылается. Работе с текстом посвящена отдельная глава книги.

Для отображения окна сообщения с информацией о значениях поля объекта obj1 используем команду obj1.showText(). Данные по второму объекту выводятся в окне сообщения, которое отображается в результате выполнения команды obj2.showText().

На заметку:

Хотя создание специального метода для вывода информации об объекте является приемом достаточно удобным, существует еще более простой способ решения этой задачи. Состоит он в *переопределении* метода toString(). Как это делается, описано в главе, посвященной работе с текстом.

Конструкторы

Идея создания *конструктора* напрашивается сама собой. В предыдущем примере мы создавали объекты, а затем присваивали полям этих объектов значения. Логично это было бы делать еще на этапе создания объектов.

Именно в таких случаях полезны *конструкторы*. Мы уже знаем, что *конструктор* – это метод, который выполняется вначале, при создании объекта. Разберемся в том, зачем конструктор нужен и как его создать (описать).

Конструктор создается практически так же, как и обычный метод класса, но есть некоторые ограничения.

- Имя конструктора совпадает с именем класса.
- Конструктор не возвращает результат, а в сигнатуре конструктора *не указывается* идентификатор типа возвращаемого результата – *никакой*!
- У конструктора могут быть аргументы, которые описываются так же, как и аргументы обычного метода.

📖 На заметку:

Еще одно очень важное свойство конструктора состоит в том, что его, как и обычный метод класса, можно *перегружать*. Что такое перегрузка методов (и конструкторов), описывается в следующем разделе.

Рассмотрим очередную модификацию программного кода, который рассматривался ранее. В листинге 6.4 представлен новый вариант программы, в которой у класса `MyClass` появился еще и конструктор.

Листинг 6.4. Конструктор класса

```
import javax.swing.*;
// Описание класса:
class MyClass{
int num;
char symb;
String text;
// Конструктор класса с тремя аргументами:
MyClass(int n,char s,String str){
// Заполнение полей:
set(n,s,str);
// Отображение сообщения с информацией об объекте:
showText();
}
// Метод класса для отображения сообщения:
void showText(){
// Локальная текстовая переменная:
String str="Значения полей объекта:\n";
// Формирование результата (значения локальной переменной):
str+="num="+num+"\n";
str+="symb="+symb+"\n";
str+="text="+text;
```

```
// Метод отображает окно сообщения:
JOptionPane.showMessageDialog(null,str);}
// Метод класса для присваивания значений полям:
void set(int n,char s,String str){
   // Заполнение полей:
   num=n;
   symb=s;
   text=str;}
}
// Главный класс программы:
public class ClassDemo4{
   public static void main(String[] args){
      // Создание объекта с передачей аргументов конструктору:
      MyClass obj=new MyClass(15,'Ы',"Просто текст.");}
   }
```

В описание класса `MyClass` добавлено описание конструктора:

```
MyClass(int n,char s,String str){
   set(n,s,str);
   showText();
   }
```

Поскольку класс называется `MyClass`, то так же называется и конструктор. У конструктора три аргумента: целое число, символ и текст. В теле конструктора вызывается метод `set()` с аргументами, переданными конструктору. Напомним, что конструктор автоматически вызывается при создании объекта. Поэтому те аргументы, которые мы передадим конструктору, будут автоматически использованы при вызове метода `set()`, и, значит, поля создаваемого объекта получат соответствующие значения. Кроме того, после вызова метода `set()` в конструкторе вызывается еще и метод `showText()`. Таким образом, после того, как поля объекта получают значения, на экран выводится сообщение с информацией о значении этих полей.

Главный метод программы состоит всего из одной команды `MyClass obj =new MyClass(15,'Ы',"Просто текст.")`. Этой командой создается объект `obj` класса `MyClass`. Интерес представляет часть команды с оператором `new`. После этого оператора указано имя класса и в круглых скобках три аргумента. Несложно догадаться, что это те самые аргументы, которые передаются конструктору. Это общее *правило для передачи аргументов конструктору в Java*: аргументы указываются в круглых скобках после имени конструктора в соответствии с тем порядком, как они описаны в коде конструктора.

В результате выполнения программы на экране появляется диалоговое окно, показанное на рис. 6.4.

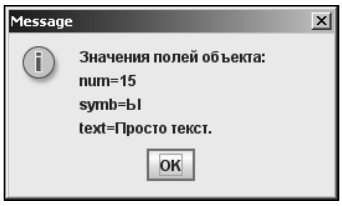


Рис. 6.4. Окно отображается при вызове конструктора

Обращаем внимание читателя, что главный метод программы не содержит явно команды отображения такого окна, и метод `showText()` напрямую в нем не вызывается. Отображение окна происходит в результате вызова конструктора. Сам конструктор вызывается автоматически.

На заметку:

Те, кто знаком с языком программирования C++, знают, что в C++ помимо *конструкторов*, есть еще и *деструкторы*. Деструктор вызывается автоматически при удалении объекта из памяти. В Java деструкторов нет, хотя и имеется концептуально близкий к деструкторам метод `finalize()`. Однако этот метод нельзя рассматривать как замену деструктора. Тем не менее проблемы здесь нет, поскольку в отличие от C++ в Java действует система автоматической "сборки мусора" – нет необходимости на программном уровне контролировать своевременность и корректность освобождения памяти от неиспользуемых объектов.

Даже если в классе конструктор не описан, существует так называемый *конструктор по умолчанию*, который не имеет аргументов. Именно такой конструктор вызывался в примерах, кроме последнего. Тем не менее, как только класс обзаводится явно описанным конструктором, конструктор по умолчанию становится недоступным. Это означает, например, что в последнем примере команда создания объекта без передачи аргументов конструктору вызовет ошибку компиляции. Чтобы создавать объекты можно было разными способами (передавая или не передавая аргументы конструктору), используют перегрузку конструктора.

Перегрузка методов и конструкторов

Перегрузка методов – очень мощный механизм, который позволяет создавать простые и элегантные программы. Перегрузка является одним из проявлений принципа *полиморфизма* – одного из трех основополагающих принципов ООП. Что же такое перегрузка методов? Это создание нескольких версий одного метода – все версии имеют одно и то же имя, но разные сигнатуры.

На заметку:

Напомним, сигнатура метода состоит из идентификатора типа результата, имени метода и списка аргументов. Таким образом, разные версии перегружаемого метода имеют одно название, но отличаются типом результата и/или списком аргументов.

Зачем нужна перегрузка? Практическое ее применение сводится к тому, что в программном коде можно вызывать один и тот же метод, передавая ему разные аргументы, и получать результат, *mun* которого может зависеть от списка аргументов, переданных методу. Это удобно, поскольку обычно разные версии перегружаемого метода предназначены для выполнения похожих или однотипных действий, но только с разным набором параметров – например, сортировки массива (числового или символьного). В этом случае нет необходимости создавать разные методы для выполнения операции над разным набором данных. Каждый раз вызывается один и тот же метод.

На заметку:

На самом деле это иллюзия – вызываются разные методы, просто все они вызываются через одно и то же имя, поэтому создается иллюзия, что это один метод.

Решение о том, какую именно версию метода нужно вызвать, принимается исходя из контекста команды, в которой вызывается метод, то есть на основе списка аргументов и типа выражения, которое предполагается получить.

Для того чтобы перегрузить метод, необходимо и достаточно описать каждую из его версий – так, как если бы это был отдельный метод. Это же правило относится и к конструкторам.

На заметку:

Поскольку для конструктора идентификатор типа не указывается, то разные версии конструктора класса могут и должны отличаться только списком аргументов.

Обратимся к примеру в листинге 6.5.

Листинг 6.5. Перегрузка методов и конструкторов

```
import javax.swing.*;
// Описание класса:
class MyClass{
int num;
char symb;
String text;
// Конструктор класса без аргументов:
MyClass(){
```

```
   // Заполнение полей:
   set();
   // Отображение сообщения с информацией об объекте:
   showText();
   }
// Конструктор класса с целочисленным аргументом:
MyClass(int n){
   // Заполнение полей:
   set(n);
   // Отображение сообщения с информацией об объекте:
   showText();
   }
// Конструктор класса с символьным аргументом:
MyClass(char s){
   // Заполнение полей:
   set(s);
   // Отображение сообщения с информацией об объекте:
   showText();
   }
// Конструктор класса с тремя аргументами:
MyClass(int n,char s,String str){
   // Заполнение полей:
   set(n,s,str);
   // Отображение сообщения с информацией об объекте:
   showText();
   }
// Метод класса для отображения сообщения:
void showText(){
   // Локальная текстовая переменная:
   String str="Значения полей объекта:\n";
   // Формирование результата (значения локальной переменной):
   str+="num="+num+"\n";
   str+="symb="+symb+"\n";
   str+="text="+text;
   // Метод отображает окно сообщения:
   JOptionPane.showMessageDialog(null,str);}
// Метод класса для присваивания значений полям (нет
// аргументов):
void set(){
   // Заполнение полей:
   num=0;
   symb='a';
   text="Нет аргументов.";}
// Метод класса для присваивания значений полям (целочисленный
// аргумент):
void set(int n){
   // Заполнение полей:
   num=n;
   symb='b';
```

```
   text="Целочисленный аргумент.";}
// Метод класса для присваивания значений полям (символьный
// аргумент):
void set(char s){
   // Заполнение полей:
   num=1;
   symb=s;
   text="Символьный аргумент.";}
// Метод класса для присваивания значений полям (три
// аргумента):
void set(int n,char s,String str){
   // Заполнение полей:
   num=n;
   symb=s;
   text=str;}
   }
// Главный класс программы:
public class ClassDemo5{
   public static void main(String[] args){
      // Создание объектов:
      MyClass obj1=new MyClass();
      MyClass obj2=new MyClass(2);
      MyClass obj3=new MyClass('Z');
      MyClass obj4=new MyClass(3,'A',"Три аргумента.");}
   }
```

В классе `MyClass` описаны такие варианты конструктора:

- Без аргументов.
- С одним целочисленным аргументом.
- С одним символьным аргументом.
- С тремя аргументами (целочисленный, символьный и текстовый).

Такие же варианты перегрузки предусмотрены для метода `set()`. В каждом из вариантов конструктора вызывается соответствующая версия метода `set()` со списком аргументов, которые передаются конструктору. В частности, если у конструктора нет аргументов, то метод `set()` тоже вызывается без аргументов. В этом случае поле `num` объекта получает нулевое значение. Символьное поле `symb` получает значение `'a'`, а текстовое поле `text` получает значение `"Нет аргументов."`. Далее, если конструктору передается всего один целочисленный аргумент, то с этим целочисленным аргументом вызывается метод `set()` и именно этот аргумент определяет значение поля `num`. Полю `symb` присваивается значение `'b'`, а полю `text` присваивается значение `"Целочисленный аргумент."`. Если конструктору передается единственный символьный аргумент, то при вызове метода `set()` с этим аргументом его значение присваивается полю `symb`. Числовое поле `num` получает значение `1`, а текстовому полю `text` присваивается

значение "Символьный аргумент.". И, разумеется, остался традиционный вариант конструктора с передачей трех аргументов – он уже описывался в предыдущем примере.

В главном методе программы по очереди создаются четыре объекта – каждый раз с использованием нового конструктора. В результате последовательно открываются четыре окна (каждое следующее открывается после закрытия предыдущего). Окна представлены на рис. 6.5 – рис. 6.8.

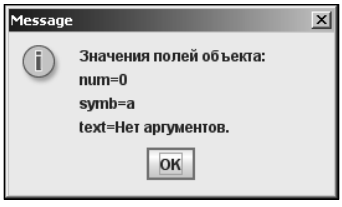


Рис. 6.5. Результат вызова конструктора без аргументов

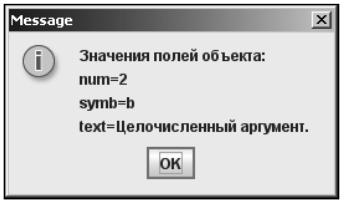


Рис. 6.6. Результат вызова конструктора с целочисленным аргументом

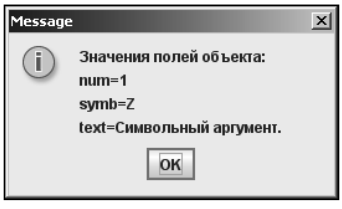


Рис. 6.7. Результат вызова конструктора с символьным аргументом

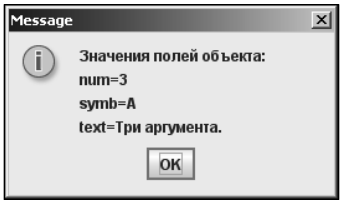


Рис. 6.8. Результат вызова конструктора с тремя аргументами

Выше отмечалось, что при перегрузке методов разные версии метода обычно выполняют однотипные операции. Это не жесткое требование, а скорее рекомендация и правило хорошего тона. Формально можно переопределить

метод так, что разные его версии будут совершенно непохожими друг на друга (в плане конечного своего предназначения). Но это сомнительный подход.

Присваивание объектов

Аналогично тому, как это было с массивами, при создании и работе с объектами мы используем переменные, которые в качестве значения содержат ссылку на объект. Такие переменные мы называем *объектными переменными*. Важно четко понимать, что **объектная переменная – это не сам объект, а всего лишь ссылка на него.** Во многих случаях такие тонкие материи на практике последствий не имеют. Но *во многих* еще не означает *всегда*. Классический пример того, как объектные переменные проявляют свою коварную сущность – присваивание объектов. Рассмотрим простой программный код, представленный в листинге 6.6.

Листинг 6.6. Присваивание объектов

```
import javax.swing.*;
class AssignObj{
// Текстовое поле класса:
String text;
// Конструктор класса (без аргумента):
AssignObj(){
   text="Новый объект!";}
// Конструктор класса (с одним аргументом):
AssignObj(String str){
   text=str;}
// Метод для отображения текста:
void show(){
   JOptionPane.showMessageDialog(null,text);}
   }
public class AssignObjDemo{
   public static void main(String[] args){
      // Создание объектов:
      AssignObj objA=new AssignObj();
      AssignObj objB=new AssignObj("Второй объект!");
      // Отображение текстовых полей объектов:
      objA.show();
      objB.show();
      // Присваивание объектов:
      objA=objB;
      // Текстовое поле первого объекта:
      objA.show();
      // Изменение поля второго объекта:
      objB.text="Изменен второй объект!";
      // Текстовое поле первого объекта:
      objA.show();}
   }
```

В результате выполнения программы отображается несколько диалоговых окон, которые содержат разные сообщения – в соответствии со значениями текстовых полей объектов, из которых вызывается метод `show()` класса `AssignObj`. В частности, в классе `AssignObj` объявляется текстовое поле `text`, метод `show()` для отображения сообщения со значением этого поля и перегруженный конструктор (без аргументов и с одним текстовым аргументом). Если для создания объекта используется конструктор без аргументов, текстовое поле получает значение `"Новый объект!"`. При создании объекта с помощью конструктора с аргументом этот аргумент присваивается в качестве значения текстовому полю объекта. По первой схеме в методе `main()` создается объект `objA` класса `AssignObj`, а по второй – объект `objB` (конструктору передается аргумент `"Второй объект!"`). Затем метод `show()` последовательно вызывается из каждого объекта. В результате выполнения команды `objA.show()` отображается окно, представленное на рис. 6.9.

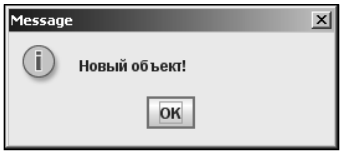


Рис. 6.9. Окно отображается при вызове метода `show()` из объекта `objA` до присваивания объектов `objA=objB`

Командой `objB.show()` отображается окно, представленное на рис. 6.10.

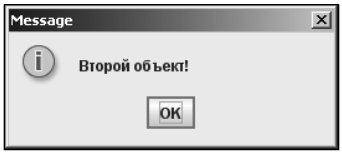


Рис. 6.10. Окно отображается при вызове метода `show()` из объекта `objB` до присваивания объектов и из объекта `objA` после присваивания объектов `objA=objB`

После этого выполняется команда `objA=objB`, которой одной объектной переменной `objA` в качестве значения присваивается другая объектная переменная `objB`. Как только присваивание выполнено, метод `show()` вызывается из объекта `objA`. В результате снова увидим окно, представленное на рис. 6.10. Объяснение достаточно простое. Дело в том, что значение объектной переменной – это ссылка на объект. Поэтому в результате выполнения команды `objA=objB` переменная `objA` будет ссылаться на тот же объект, что и переменная `objB`. Убедиться в последнем достаточно просто. Командой `objB.text="Изменен второй объект!"` меняем текстовое поле второго объекта, а затем командой `objA.show()` вызываем метод `show()` из первого объекта. Результат представлен на рис. 6.11.

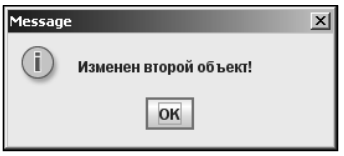


Рис. 6.11. Окно отображается при вызове метода `show()` из объекта `objA` после присваивания объектов `objA=objB` и внесения изменений в текстовое поле объекта `objB`

Таким образом, присваивание объектов свелось к "перебрасыванию" ссылки из одного объекта на другой. В результате на первый объект не ссылается ни одна переменная, а на второй – две.

На заметку:

Объекты, на которые в программе нет ни одной ссылки, автоматически отслеживаются и удаляются из памяти.

Создание копии объекта

Возникает естественный вопрос: что же делать, если необходимо создать копию объекта? Один из выходов – описать конструктор создания копии, а лучше – специальный метод. Обратимся к коду в листинге 6.7.

Листинг 6.7. Создание копии объекта

```
class MakeObjCopy{
// Текстовое поле класса:
String name;
// Числовое поле класса:
int number;
// Конструктор класса (два аргумента):
MakeObjCopy(String str,int num){
name=str;
number=num;}
// Конструктор создания копии:
MakeObjCopy(MakeObjCopy obj){
   name=obj.name;
   number=obj.number;}
// Метод для создания копии объекта:
MakeObjCopy copy(){
   // Создание объекта-копии:
   MakeObjCopy tmp=new MakeObjCopy(name,number);
   // Возвращение результата:
   return tmp;}
// Метод для вывода информации об объекте:
void show(){
   String text="Поля объекта:\n";
```

```
   text+="name: "+name+"\nnumber: "+number;
   // Вывод на консоль:
   System.out.println(text);}
   }
public class MakeObjCopyDemo{
   public static void main(String[] args){
      // Создание объектов:
      MakeObjCopy objA=new MakeObjCopy("Первый объект",100);
      MakeObjCopy objB=new MakeObjCopy(objA);
      // Отображение информации об объектах:
      objA.show();
      objB.show();
      // Изменение второго объекта:
      objB.name="Второй объект";
      objB.number=200;
      // Отображение информации об объектах:
      objA.show();
      objB.show();
      // "Копирование" объектов:
      objA=objB.copy();
      // Отображение информации об объекте:
      objA.show();
      // Изменение второго объекта:
      objB.name="Второй объект изменен!";
      objB.number=300;
      // Отображение информации об объектах:
      objA.show();
      objB.show();}
   }
```

При выполнении программы в консольном окне появляется следующая серия сообщений:

```
Поля объекта:
name: Первый объект
number: 100
Поля объекта:
name: Первый объект
number: 100
Поля объекта:
name: Первый объект
number: 100
Поля объекта:
name: Второй объект
number: 200
Поля объекта:
name: Второй объект
number: 200
```

```
Поля объекта:
name: Второй объект
number: 200
Поля объекта:
name: Второй объект изменен!
number: 300
```

Разберем, почему появляются именно такие сообщения. Для этого обратимся к коду, которым описывается класс `MakeObjCopy`. У класса два поля: текстовое поле `name` и целочисленное поле `number`. Перегруженный конструктор позволяет создавать объекты класса, передавая два аргумента (текст и целое число – значения полей объекта) или уже существующий объект класса `MakeObjCopy`. В последнем случае поля создаваемого объекта заполняются в соответствии со значениями полей объекта, переданного аргументом конструктору.

На заметку:

Если объектная переменная передается аргументом методу (конструктору) или возвращается в качестве результата метода, в качестве идентификатора типа указывается имя соответствующего класса, к которому принадлежит объектная переменная.

Метод `copy()` предназначен для создания копии объекта. У метода нет аргументов. Он "копирует" тот объект, из которого вызывается. При выполнении метода создается еще один, новый объект. Значения полей этого нового объекта совпадают со значениями полей исходного объекта (из которого вызывается метод `copy()`). В качестве результата метод возвращает ссылку на созданный объект.

В теле метода командой `MakeObjCopy tmp=new MakeObjCopy(name, number)` создается локальный объект `tmp` класса `MakeObjCopy`. Для создания объекта используется конструктор с двумя аргументами – значениями полей исходного объекта (из которого вызывается метод). Это фактически и есть та копия объекта, о которой шла речь.

На заметку:

Вместо команды `MakeObjCopy tmp=new MakeObjCopy(name,number)` при создании локального объекта `tmp` можно было использовать команду `MakeObjCopy tmp=new MakeObjCopy(this)`. Особенность команды состоит в том, что в ней использована инструкция `this`. Данная инструкция является ссылкой на объект, из которого вызывается метод. Другими словами, ключевое слово `this`, когда оно используется в программном коде метода, означает тот объект, из которого этот метод вызывается. В данном случае ссылка `this` передается в качестве аргумента конструктору создания копии.

Следующей командой `return tmp` ссылка на созданный объект возвращается в качестве результата метода.

На заметку:

В общем случае все локальные переменные метода (переменные, объявленные в теле метода) существуют до тех пор, пока выполняется код метода, и доступны только в пределах программного кода этого метода. Но поскольку в данном случае объектная ссылка `tmp` возвращается в качестве значения, то по завершении работы метода объект, на который ссылается переменная `tmp`, из памяти не удаляется.

Поскольку в данном случае выводится довольно много однотипного текста, вместо создания окон информацию выводим на консоль. Для этих целей в классе `MakeObjCopy` описан метод `show()`. У метода нет аргументов, и он не возвращает результат. В теле метода объявляется локальная текстовая переменная `text`, в которую записывается текстовое сообщение, содержащее информацию о значении полей объекта, из которого вызывается метод. Соответствующий текст выводится в консольное окно с помощью команды `System.out.println(text)`, в которой из поля-объекта `out` класса `System` вызывается метод `println()`, который выводит на консоль значение переданного ему аргумента. Это структура класса `MakeObjCopy`.

В главном методе программы создаются два объекта `objA` и `objB`. Для этого используем команды `MakeObjCopy objA=new MakeObjCopy ("Первый объект",100)` и `MakeObjCopy objB=new MakeObjCopy (objA)`. Объект `objA` создается путем явной передачи конструктору значений полей, а объект `objB` создается как копия объекта `objA`. У объекта `objB` значения полей такие же, как и у объекта `objA`, но при этом объекты физически разные (то есть это *не один и тот же объект*). Для проверки командами `objA.show()` и `objB.show()` выводим на экран значения полей каждого из объектов (напомним, значения полей совпадают). Чтобы удостовериться, что объекты при этом разные, командами `objB.name="Второй объект"` и `objB.number=200` меняем значения полей объекта `objB`. После командами `objA.show()` и `objB.show()` еще раз проверяем значения полей объектов. Результат для первого объекта `objA` не изменился, а у второго объекта `objB` поля изменили свои значения в соответствии с тем, что мы им предварительно присвоили. Так и должно быть, поскольку переменные `objA` и `objB` ссылаются на разные объекты.

На заметку:

Фразу вида "*объект* `objA`" следует понимать так: "*объект, на который ссылается объектная переменная* `objA`".

Теперь проверим работу метода `copy()`. Для этого командой `objA=objB.copy()` выполняем "копирование" объектов. В результате выполнения данной команды на основе объекта `objB` создается еще один объект с такими же значениями полей, и ссылка на него записывается в переменную `objA`. Поэтому после выполнения команды `objA.show()` увидим то же сообщение, что и при выводе на экран значений полей объекта `objB`. Но, как и в предыдущем случае, `objA` и `objB` – разные объекты. Убедиться в последнем легко. Командами `objB.name="Второй объект изменен!"` и `objB.number=300` изменяем значения полей второго объекта, после чего командами `objA.show()` и `objB.show()` проверяем значения полей объектов. Несложно убедиться, что изменения коснулись полей лишь одного объекта – объекта `objB`. Поля объекта `objA` значений не изменили.

На этом знакомство с классами и объектами заканчиваем. Более подробно методы работы с классами и объектами обсуждаются в следующей главе.

Резюме

1. Класс представляет собой конструкцию, в которой объединены данные и программный код для обработки этих данных. Конкретный экземпляр класса называется объектом.
2. Класс может содержать переменные (поля класса) и методы для обработки полей и не только (методы класса). Поля и методы класса называются членами класса.
3. Описание класса начинается со слова `class`, затем следует имя класса и в фигурных скобках код класса.
4. Объекты класса создаются динамически с помощью оператора `new`. После оператора указывается имя класса и в круглых скобках аргументы, передаваемые конструктору. Как результат возвращается ссылка на созданный объект. Эта ссылка обычно записывается в объектную переменную.
5. При объявлении объектной переменной указывают имя класса и имя этой переменной.
6. Конструктором класса называется метод, который вызывается автоматически при создании объекта. Конструктор имеет такое же название, как класс, и для него не указывается тип возвращаемого результата.
7. Методы (и конструкторы) могут перегружаться. В этом случае создается несколько вариантов метода с одним и тем же именем, но разными сигнатурами. Какой вариант метода следует вызывать, определяется исходя из контекста команды с вызовом метода.

Глава 7
Тонкости работы с объектами

- Легкие дышат. Сердце стучит.
- А голова?
- А голова - предмет темный, исследованию не подлежит.
- Мудро!
(Из к/ф "Формула любви")

В этой главе мы продолжим знакомство с объектами и, соответственно, классами. Здесь остановимся на тех вопросах, которые касаются использования объектов, и которые не вошли в предыдущую главу. Собственно, речь пойдет не только об использовании объектов как таковых, но и о некоторых особенностях описания классов. Глава во многом является "сборной" и состоит из отдельных, мало связанных между собой разделов. Поэтому разделы, в принципе, можно изучать в произвольном порядке.

Статические поля и методы

Мы уже знаем, что класс представляет собой лишь шаблон, на основе которого создаются объекты. Поля и методы мы вызывали из объектов. На самом деле это не всегда так. Из объектов вызываются обычные поля и методы. Кроме обычных полей и методов еще бывают не очень обычные, а точнее, *статические* поля и методы, или *статические члены класса*.

📖 На заметку:

Откровенно говоря, мы уже использовали статические методы, хотя особо внимания на этом не заостряли. Самый яркий пример – главный метод программы `main()`.

Чтобы сделать поле или метод статическим, достаточно в описание поля или в сигнатуру метода добавить ключевое слово `static`. Это все очень просто, и здесь нет никакой интриги. Интрига (некоторая) есть в том, как используются статические члены класса.

Главная идея состоит в том, что статическое поле или метод являются *общими* для всех экземпляров класса, то есть для всех объектов этого класса. Как следствие, к статическим полям или методам можно обращаться безотносительно к конкретному объекту. Более того, даже если ни один объект для класса не создан, статические поля и методы все же доступны.

На заметку:

Возвращаясь к главному методу программы `main()`, стоит напомнить, что в начале книги мы создавали программы всего с одним классом (содержащим главный метод), но не создавали экземпляров этого класса. То есть главный метод программы вызывался и вызывается без создания объекта класса, в котором этот метод описан.

Для вызова статического метода или обращения к статическому полю вместо имени объекта указывается имя класса. Хотя при этом легитимным остается и старый способ обращения к членам класса, то есть через объект (если таковой имеется). Рассмотрим пример. Сначала обсудим методы работы и особенности статических *полей*. Обратимся к листингу 7.1.

Листинг 7.1. Статические члены класса

```
import javax.swing.*;
class MyClass{
   // Статическое поле класса:
   static int N1;
   // Нестатическое поле:
   int N2;
   // Конструктор класса:
   MyClass(int n1,int n2){
      N1=n1;
      N2=n2;
      // Текст для отображения в окне:
      String text="Создан новый объект!\n";
      text+="Статическое поле: "+N1+"\n";
      text+="Нестатическое поле: "+N2;
      // Отображение окна сообщения:
      JOptionPane.showMessageDialog(null,text);}
   // Метод для отображения значений полей:
   void show(){
      // Текст для отображения в окне:
      String text="Поля объекта!\n";
      text+="Статическое поле: "+N1+"\n";
      text+="Нестатическое поле: "+N2;
      // Отображение окна:
      JOptionPane.showMessageDialog(null,text);}
   }
class UsingStatDemo{
   public static void main(String[] args){
      // Создание объекта:
      MyClass A=new MyClass(10,200);
      // Изменение значения статического поля
      // (использована ссылка на класс):
```

```
    MyClass.N1=-50;
    // Отображение полей объекта:
    A.show();
    // Создание нового объекта:
    MyClass B=new MyClass(1,2);
    // Отображение полей первого объекта:
    A.show();
    // Изменение статического поля
    // (использована ссылка на объект):
    B.N1=-1;
    // Изменение нестатического поля объекта:
    B.N2=-2;
    // Проверка значений первого объекта:
    A.show();}
}
```

В программе описан класс `MyClass`, содержащий два числовых поля: статическое поле `N1` и нестатическое поле `N2`. Конструктору класса передается два аргумента: первый аргумент определяет значение статического поля `N1`, а второй аргумент определяет значение статического поля `N2`. Кроме присваивания значения полям, в конструкторе формируется текстовое сообщение (локальная переменная текстового типа) с информацией о полях вновь созданного объекта (включая значение статического поля) и отображается соответствующее диалоговое окно.

На заметку:

Присваивание значения статическому полю в конструкторе является идеей не очень хорошей. Дело в том, что в такой схеме каждый раз при создании нового объекта будет изменяться значение статического поля, которое, как известно, одно на все объекты. Целесообразность использования статического поля в таком контексте представляется не очень целесообразной. Но в данном случае пример иллюстративный, поэтому проблемы никакой нет.

Что касается инициализации статического поля, то ее можно выполнить прямо в описании класса, как для обычной переменной. Например, если объявить статическое поле в классе `MyClass` как `static int N1=10`, то каждый раз при запуске программы статическое поле будет получать значение `10`.

Также в классе `MyClass` описан метод `show()`, с помощью которого отображаются сведения о полях уже существующего объекта.

В главном методе программы командой `MyClass A=new MyClass(10, 200)` создается объект `A`. Статическое поле получает значение `10`, а нестатическое поле – значение `200`. При создании объекта автоматически отображается окно сообщения, представленное на рис. 7.1.

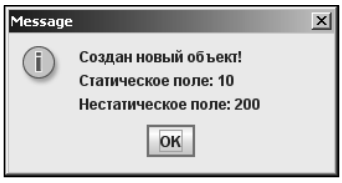


Рис. 7.1. Окно отображается при создании первого объекта

Затем командой `MyClass.N1=-50` изменяется значение статического поля. Обращаем внимание, что при обращении к статическому полю обращение выполняется через имя класса, то есть в формате *имя_класса.статическое_поле*. Изменив значение статического поля, командой `A.show()` проверяем значения полей объекта `A`. Отображаемое при этом диалоговое окно показано на рис. 7.2.

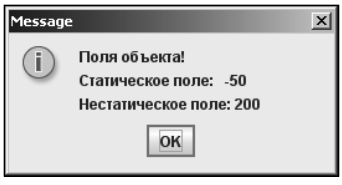


Рис. 7.2. Окно отображается после изменения статического поля

Как видим, статическое поле действительно изменилось. Новую интервенцию на это поле выполним, создав еще один объект (объект `B`), воспользовавшись командой `MyClass B=new MyClass(1,2)`. При создании этого объекта отображается окно, представленное на рис. 7.3.

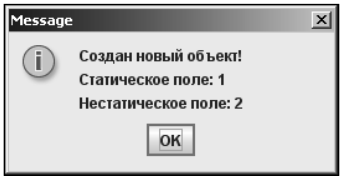


Рис. 7.3. Окно отображается при создании второго объекта

Как уже отмечалось, конструктор класса описан так, что создание нового объекта изменяет значение статического поля. Убеждаемся в этом, воспользовавшись командой `A.show()`. Нестатическое поле объекта `A` не изменилось (причин к этому не было), зато изменилось статическое (рис. 7.4).

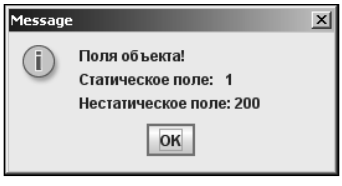


Рис. 7.4. Окно отображается для первого объекта после создания второго объекта

Изменить значение статического поля можно не только через класс, но и через объект, хотя это и не очень распространенная практика. В данном случае командами `B.N1=-1` и `B.N2=-2` изменяем значения полей объекта `B` (первое из полей статическое). Проверку значений первого объекта выполняем командой `A.show()`. Результат показан на рис. 7.5.

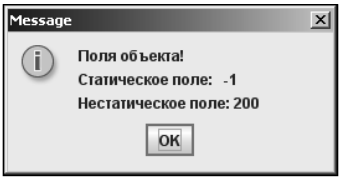


Рис. 7.5. Окно отображается для первого объекта после изменения полей второго объекта

Как и следовало ожидать, статическое поле изменилось.

Еще раз обращаем внимание, что рассмотренный пример иллюстративный. Обычно статические поля используются по-иному. Например, можно использовать статическое поле для подсчета количества созданных объектов: при создании нового объекта в конструкторе на единицу увеличивается значение статического поля. Вообще же статические поля во многих отношениях напоминают глобальные переменные, хотя аналогия и несколько натянутая.

Так же просто, как статические поля, объявляются и используются статические методы. Например, статические методы удобно использовать для создания библиотеки математических функций. Рассмотрим простой иллюстративный пример. Обратимся к листингу 7.2.

Листинг 7.2. Статические методы

```
import javax.swing.*;
// Класс со статическим полем и статическим методом:
class MyMath{
   // Статическая константа:
   final static double PI=3.14159265;
   // Статический метод для вычисления синуса:
   static double sin(double x,int n){
      // Локальные переменные:
      double s=0,q=x;
      int i;
      // Оператор цикла для вычисления ряда Тейлора для синуса:
      for(i=1;i<=n;i++){
         s+=q;
         q*=(-1)*x*x/(2*i)/(2*i+1);}
      // Результат:
      return s+q;}
   }
```

```
// Класс с главным методом:
class StatMethDemo{
   public static void main(String[] args){
      // Текст для отображения в окне сообщения:
      String text="Значения ряда Тейлора для синуса.";
      // Оператор цикла для вычисления синуса (несколько значений):
      for(int k=0;k<5;k++){
         text+="\nСлагаемых "+(k+1)+": ";
         text+="sin(pi/4)="+MyMath.sin(MyMath.PI/4,k);}
      // Отображение окна сообщения:
      JOptionPane.showMessageDialog(null,text);}
   }
```

В классе `MyMath` описано статическое поле и статический метод. Статическое действительное поле `PI` в качестве значения содержит приближение для числа π. Поле является не только статическим, но еще и постоянным, то есть константой, значение которой в программном коде изменить нельзя. О том, что поле является константой, свидетельствует ключевое слово `final` в описании поля.

Кроме статического поля, в классе есть статический метод `sin()`. У метода два аргумента, и в качестве результата методом возвращается значение синуса. Синус вычисляется для значения первого аргумента – это переменная типа `double`. При вычислении используется ряд Тейлора. Речь идет о выражении $\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots + \frac{(-1)^n x^{2n+1}}{(2n+1)!}$. Чем больше слагаемых в сумме взять, тем точнее результат (в идеале точное выражение для синуса – это бесконечная сумма). Второй аргумент метода `sin()` – целое число, которое определяет количество слагаемых при вычислении ряда для синуса.

В теле метода реализуется алгоритм, который по известным значениям аргументов метода `x` и `n` позволяет вычислить (приближенное) значение для синуса. В частности, в теле метода объявляются локальные `double`-переменные: `s` с начальным значением `0` (переменная, в которую записывается сумма для ряда) и переменная `q` с начальным значением `x` (в переменную `q` записывается очередная добавка к сумме). Кроме этого, объявляется целочисленная индексная переменная `i` для оператора цикла.

Что касается самого оператора цикла, то за каждый цикл выполняется две команды: инструкцией `s+=q` текущее значение суммы изменяется на величину добавки, а инструкцией `q*=(-1)*x*x/(2*i)/(2*i+1)` вычисляется значение добавки для следующего итерационного шага. После выполнения оператора цикла в качестве значения возвращается величина `s+q`. Здесь принято во внимание, что последняя вычисленная в рамках операто-

ра цикла добавка не была использована на последнем этапе цикла. На этом описание метода `sin()` и всего класса `MyMath` заканчивается.

На заметку:

В Java есть специальный встроенный класс `Math`, который представляет собой библиотеку основных математических функций. Так что особой необходимости определять собственные математические библиотеки или классы при работе с Java нет.

В главном методе программы в классе `StatMethDemo` для разного количества слагаемых в ряде Тейлора для синуса (от 1 до 5 включительно) вычисляется значение синуса для аргумента $\pi/4$ (точное значение $1/\sqrt{2} \approx 0.707106781$). При вычислениях используется статическое поле PI и статический метод `sin()` класса `MyMath`. Результат для каждого из вычислений заносится в текстовую переменную и потом все это отображается в окне сообщения (рис. 7.6).

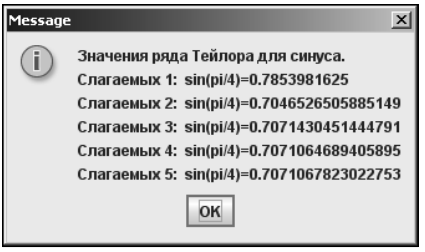


Рис. 7.6. Результат вычисления синуса для разного количества слагаемых в ряде Тейлора

Обращаем внимание читателя, что при обращении к статическому методу используется имя класса (команда вида `MyMath.sin(`*аргументы*`)`), а объект для класса `MyMath` вообще не создается.

Объекты и методы

Мы уже знаем, что объекты могут возвращаться методами в качестве результата. Во всяком случае, в предыдущей главе был рассмотрен пример, в котором с помощью метода создавалась копия объекта. Также объекты могут передаваться методам в качестве аргументов. Общее правило такое: если объект передается аргументом методу, то при описании метода в сигнатуре в качестве типа соответствующей переменной указывается имя класса, к которому относится объект.

На заметку:

На самом деле в данном случае идет речь о передаче аргументами объектных переменных.

Прежде, чем приступить непосредственно к рассмотрению примеров, как объекты передаются аргументами и возвращаются в качестве результата,

уделим немного внимания более общей проблеме: *механизму передачи аргументов методам*.

В Java есть два способа передачи аргументов: *по значению* и *по ссылке*. При передаче аргументов *по значению*, на самом деле в метод для обработки передаются не сами аргументы, а их технические копии. При передаче аргументов *по ссылке* операции выполняются непосредственно с аргументами. В Java переменные базовых типов передаются в качестве аргументов по значению. Так называемые ссылочные переменные (объектные переменные и переменные массива) передаются в качестве аргументов по ссылке. На практике это сводится к следующему. Если аргумент метода относится к базовому типу данных, то в теле метода изменить значение переменной, переданной в качестве аргумента, не удастся. Если аргумент – это объект (или массив, например), то все операции выполняются с этим объектом (или массивом).

На заметку:

Объяснение у такой избирательности к способу передачи аргументов достаточно простое. Ведь при передаче объекта аргументом методу на самом деле передается объектная переменная, значением которой является ссылка на объект. Если для такой переменной создать копию, эта копия будет ссылаться на тот же самый объект. Поскольку операции выполняются с объектом, на который ссылается объектная переменная, то не имеет значения, используется ли исходная объектная переменная или ее копия.

Рассмотрим простой иллюстративный пример в листинге 7.3.

Листинг 7.3. Передача аргументом объекта

```
class MyClass{
// Поля класса:
int number;
char symb;
// Конструктор класса:
MyClass(int number,char symb){
   // Для различения полей и аргументов используем
   // ключевое слово this:
   this.number=number;
   this.symb=symb;}
// Метод для отображения значений полей объекта:
void show(){
   System.out.println("Поля объекта: "+number+" и "+symb);}
   }
public class ArgsDemo{
   // Перегруженный статический метод.
   // Аргумент - объект:
   static void MakeChange(MyClass obj){
```

```
    // Изменение полей объекта - аргумента метода:
    obj.number++;
    obj.symb++;
    // Текст для отображения:
    String text="Аргумент-объект: поля "+obj.number+" и "+obj.symb;
    // Вывод сообщения на консоль:
    System.out.println(text);}
// Перегруженный статический метод.
// Аргументы - переменные базовых типов:
static void MakeChange(int number,char symb){
    // Изменение аргументов метода:
    number++;
    symb++;
    // Текст для отображения:
String text="Аргументы базовых типов: значения "+number+" и "+symb;
    // Вывод сообщения на консоль:
    System.out.println(text);}
// Главный метод программы:
public static void main(String[] args){
    // Создание объекта:
    MyClass obj=new MyClass(1,'a');
    // Значения полей объекта:
    obj.show();
    // Изменение объекта:
    MakeChange(obj);
    // Значения полей объекта:
    obj.show();
    // Изменение полей объекта:
    MakeChange(obj.number,obj.symb);
    // Значения полей объекта:
    obj.show();}
}
```

Класс `MyClass` имеет два поля: целочисленное `number` и символьное `symb`. У конструктора класса два аргумента – это значения, присваиваемые полям.

На заметку:

В описании конструктора класса названия аргументов конструктора совпадают с названиями полей класса. Возникает дилемма: как отличить в коде конструктора аргумент от поля? Правило такое: если несколько переменных имеют одинаковые имена, то приоритет имеет локальная переменная. В данном случае это означает, что простое указание имени (`number` или `symb`) в коде конструктора означает аргумент. Для обращения к полю используем полную ссылку на поля, с указанием объекта. Напомним, что для обращения в коде метода (или конструктора) класса к объекту, из которого осуществляется вызов метода (для конструктора это создаваемый объект), используется ключевое слово `this`. Чтобы обратиться к полю `number` создаваемого объекта, используем инструкцию `this.number`, а к полю `symb` обращаемся как `this.symb`.

Методом `show()`, описанным в классе, на консоль выводится информация о значениях полей объекта.

В классе `ArgsDemo` кроме главного метода программы описан еще один статический перегруженный метод `MakeChange()`. У метода две версии. Одна из них предусматривает передачу методу в качестве аргумента объекта класса `MyClass`. В соответствии с программным кодом метода поля объекта, переданного аргументом, увеличиваются на единицу (в обоих случаях использован оператор инкремента).

На заметку:

Для символьного значения (переменная типа `char`) операция инкремента означает следующую букву в кодовой таблице (алфавите).

Что касается метода `MakeChange()`, то статическим метод объявлен для того, чтобы его можно было вызывать из главного метода программы без создания класса.

Таким образом, здесь речь идет об изменении объекта, который передается аргументом методу. После внесения изменений в объект в теле метода информация о новых значениях полей объекта выводится на консоль. Другой вариант метода предполагает, что ему аргументами передаются два значения: целочисленное и символьное. Обращаем внимание читателя, что в отличие от объектной переменной, эти значения относятся к базовым (не ссылочным) типам. В методе эти значения также увеличиваются на единицу, после чего их новые значения выводятся на консоль.

Следовательно, оба варианта метода `MakeChange()` предполагают изменение своих аргументов, причем информация об изменениях выводится из тела метода. Для проверки способа передачи аргументов ссылочных и базовых типов используем очень простую идею: вносим изменения в аргументы метода `MakeChange()` и проверяем эти изменения в теле метода (то есть при выполнении метода) и после завершения выполнения метода.

В главном методе программы командой `MyClass obj=new MyClass (1,'a')` создается объект класса `MyClass`. Значения полей вновь созданного объекта проверяются командой `obj.show()`. Далее изменяем объект с помощью команды `MakeChange(obj)`. Командой `obj.show()` проверяем, были ли на самом деле внесены изменения в объект `obj` (правильный ответ *"да, были внесены"*). Еще одна попытка изменить поля объекта предпринимается с помощью команды `MakeChange(obj.number,obj.symb)`. В этом случае методу `MakeChange()` передается не объект `obj` целиком, а отдельно каждое из его полей. Затем командой `obj.show()` проверяем эффект (поля на самом деле не изменятся). Результат выполнения программы (текст в консоли) будет выглядеть так:

```
Поля объекта: 1 и a
Аргумент-объект: поля 2 и b
Поля объекта: 2 и b
Аргументы базовых типов: значения 3 и c
Поля объекта: 2 и b
```

Первое сообщение появляется при отображении начальных значений полей объекта `obj` (сразу после создания объекта). Второе сообщение выводится при выполнении метода `MakeChange()` с объектом-аргументом. Третье сообщение является результатом проверки значений полей объекта `obj` уже после вызова метода `MakeChange()`. Что мы видим? Видим, что после выполнения метода поля (значения) изменились по сравнению с тем, что было до вызова метода. Другая ситуация, когда метод вызывается с аргументами базового типа. Хотя методу как аргументы передаются поля объекта, это все равно переменные базовых типов. Значит, на самом деле они передаются по значению. Другими словами, методу передаются не непосредственно эти переменные, а их копии. Все манипуляции в теле метода выполняются с копиями. Поэтому когда проверяем новые значения полей в теле метода, они как бы изменились (четвертое сообщение). Но если мы проверим значения полей после вызова метода `MakeChange()`, значения полей остались неизменными (пятое сообщение).

Работу со ссылочными типами иллюстрирует еще один пример, представленный в листинге 7.4.

Листинг 7.4. Использование ссылочного аргумента

```
class MyClass{
// Имя объекта (текстовое поле):
String name;
// Конструктор класса:
MyClass(String name){
   this.name=name;}
// Метод для отображения имени объекта:
void show(){
   System.out.println("Объект с именем \""+name+"\".");}
   }
public class MoreArgsDemo{
   // Статический метод для изменения ссылки на объект:
   static void ChangeRef(MyClass obj){
      // Создается локальный объект:
      MyClass tmp=new MyClass("Локальный Объект");
      // Ссылка на локальный объект присваивается аргументу метода:
      obj=tmp;
      // Проверка имени объекта, на который ссылается аргумент:
      obj.show();}
```

```
   public static void main(String[] args){
      // Создание объекта:
      MyClass obj=new MyClass("Основной Объект");
      // Проверка имени объекта:
      obj.show();
      // Попытка изменить ссылку на объект:
      ChangeRef(obj);
      // Проверка результата:
      obj.show();}
   }
```

В программе объявлены два класса. Класс `MyClass` достаточно прост. У него всего одно текстовое поле `name` (отождествляем значение этого поля с именем объекта), конструктор с одним аргументом (значение, которое присваивается полю `name`) и метод `show()` для отображения значения текстового поля (имени объекта).

На заметку:

В методе `show()` при выводе консольного сообщения в тексте отображаются двойные кавычки. Для отображения двойных кавычек в тексте перед ними указываем косую черту, то есть для вставки двойных кавычек в текст используем инструкцию `\"`.

Класс `MoreArgsDemo` содержит главный метод программы и статический метод `ChangeRef()`, аргументом которому передается объект `obj` класса `MyClass` (точнее, объектная переменная этого класса). В методе создается локальный объект `tmp` класса `MyClass` (с именем `"Локальный Объект"`) и ссылка на него присваивается аргументу метода (команда `obj=tmp`). После этого командой `obj.show()` проверяем, на какой объект ссылается переменная `obj`.

В главном методе программы создаем объект `obj` с передачей конструктору класса `MyClass` текстового аргумента `"Основной Объект"`. Проверку имени объекта выполняем с помощью команды `obj.show()`. Затем пытаемся изменить значение объектной переменной, вызвав команду `ChangeRef(obj)` (предвосхищая результат, отметим, что попытка будет неудачной). Затем снова проверяем имя объекта, на который ссылается переменная `obj`, для чего используем команду `obj.show()`. В результате выполнения программы в консоль выводится три сообщения:

```
Объект с именем "Основной Объект".
Объект с именем "Локальный Объект".
Объект с именем "Основной Объект".
```

Первое сообщение выводится сразу после создания объекта `obj` в главном методе программы. В этом случае объектная переменная `obj` ссылается на

объект с текстовым полем "Основной Объект". Второе сообщение появляется при вызове метода show() из объекта obj в процессе выполнения метода ChangeRef(). В этом случае создается иллюзия, что переменная obj ссылается на локальный объект с именем "Локальный Объект". В действительности на этот объект ссылается техническая копия переменной obj, которая создается при передаче объектной переменной аргументом методу ChangeRef(). То, что объектная переменная obj не изменила своего значения, легко убедиться, взглянув на результат выполнения команды obj.show() после вызова метода ChangeRef() (третье сообщение в консоли).

Прокомментируем результат выполнения последней программы и одновременно подведем итоги. В рассмотренной выше программе интерес представляет метод ChangeRef(). Его аргументом является объектная переменная, которая, на самом деле, передается *по значению*. Другими словами, когда вызывается метод ChangeRef(), то автоматически для его аргумента создается копия – объектная переменная того же класса с тем же значением, что и переменная-аргумент. Значением объектной переменной является ссылка на объект. Поэтому и аргумент, и его копия ссылаются на один и тот же объект. Это, в свою очередь, означает, что если какие-то манипуляции выполняются с объектом, то не имеет значения, делать это непосредственно через аргумент или его копию – и в том, и в другом случае изменения происходят с одними и тем же объектом. Именно это имелось в виду, когда утверждалось, что объекты передаются *по ссылке*, а *не по значению*. Совсем по-другому обстоят дела, если мы в теле метода (как в случае с методом ChangeRef()) присваиваем значение объектной переменной-аргументу метода. В этом случае присваивание выполняется для технической копии аргумента. Сам аргумент остается неизменным, а на новый (локальный) объект ссылается копия аргумента. Пока выполняется метод, создается иллюзия, что аргумент изменился. Как только метод завершил работу, все становится на свои места. Это и произошло в рассмотренной программе.

Массивы и объекты

Массивы и объекты в Java во многом похожи. Как и в случае с объектами, в массиве следует различать переменную массива (переменную, которая в качестве значения содержит ссылку на массив) и непосредственно сам массив. Методы работы с объектами и массивами подчиняются одной общей идеологии. И, разумеется, массивы и объекты могут использоваться "совместно": как массивы объектов, как поля-объекты и как поля-массивы, например. Рассмотрим достаточно простой пример, представленный в листинге 7.5.

Листинг 7.5. Массивы и объекты

```
// Первый класс с полем - целочисленным массивом:
class First{
   // Поле класса - переменная массива:
   int[] nums;
   // Конструктор класса:
   First(int n){
      // Создание массива и связывание его
      // с полем - переменной массива:
      nums=new int[n];
      // Первый и второй элементы массива:
      nums[0]=1;
      nums[1]=1;
      // Заполнение массива числами Фибоначчи:
      for(int i=2;i<n;i++){
         nums[i]=nums[i-1]+nums[i-2];}
      }
   // Метод для отображения содержимого массива:
   void show(){
      for(int i=0;i<nums.length;i++){
         System.out.print(nums[i]+" ");}
      System.out.println();}
   }
// Второй класс:
class Second{
   // Поле - объектная переменная класса First:
   First obj;
   // Конструктор класса:
   Second(int n){
      // Создание объекта и связывание его
      // с полем - объектной переменной:
      obj=new First(n);
      // Отображение содержимого поля - объекта:
      obj.show();}
   }
public class ArraysAndObjs{
   public static void main(String[] args){
      // Количество элементов в массиве объектов:
      int count=5;
      // Создание массива объектных переменных:
      Second[] ObjAr=new Second[count];
      // Связывание объектных переменных - элементов массива
      // с создаваемыми объектами класса Second:
      for(int i=0;i<ObjAr.length;i++){
         ObjAr[i]=new Second(3*i+4);}}
   }
```

В программе используются классы, которые содержат поля-массивы и поля объекты, а также создается массив объектов.

На заметку:

В подобных случаях важно помнить, что доступ к объектам и массивам реализуется через переменные объектов и массивов соответственно. Эти переменные в качестве значения содержат ссылку на объект или массив. Например, наличие поля – массива у класса означает на самом деле, что у класса есть поле – переменная массива. Аналогично, поле – объект есть не что иное, как поле – объектная переменная, а массив объектов на практике реализуется через массив объектных переменных.

В классе `First` инструкцией `int[] nums` описано поле `nums`, которое является переменной массива, состоящего из целых чисел. Обычно мы будем упрощать ситуацию и наивно утверждать, что `nums` – это поле-массив. В действительности это совершенно не так, в чем легко убедиться, рассмотрев код конструктора класса `First`.

В теле конструктора командой `nums=new int[n]` создается массив из целых чисел и ссылка на этот массив присваивается в качестве значения полю `nums`. Размер массива определяется целочисленным аргументом, который передается конструктору. То есть массив появляется только на этом этапе. До этого была только переменная массива. Здесь она (переменная) получает значение (до этого ссылка была пустой – значение `null`). Но даже после этого массив не инициализирован явно (неявно все его элементы автоматически получают нулевые значения). Значения первому и второму элементам массива присваивается командами `nums[0]=1` и `nums[1]=1`. Каждый следующий элемент массива определяется как сумма двух предыдущих элементов. В результате получаем последовательность Фибоначчи.

На заметку:

Последовательность Фибоначчи определяется так: два первых элемента единичные, а каждое следующее число в последовательности определяется как сумма двух предыдущих элементов последовательности.

Для заполнения прочих элементов массива используем оператор цикла, в котором индексная переменная пробегает значения от `2` до `n-1` включительно (где `n` – аргумент конструктора и размер массива). Для вычисления значений элементов массива (для данного значения индексной переменной `i`) используется выражение `nums[i]=nums[i-1]+nums[i-2]`.

Кроме конструктора, в классе `First` описан еще и метод `show()`, с помощью которого на консоль выводится последовательность элементов массива – поля класса. Элементы выводятся в одну строку, поэтому используем метод `print()`. Для перехода в конце вывода всей последовательности элементов на новую строку используем метод `println()` с пустым аргументом.

На заметку:

В конструкторе размер массива определяется переданным конструктору аргументом. Этот аргумент используется в конструкторе в операторе цикла для определения верхней границы индексирования элементов массива. В методе `show()` ничего не известно о том, какой аргумент передавался конструктору при создании объекта. Поэтому размер поля-массива определяется через свойство `length`, которым возвращается количество элементов в массиве.

В классе `Second` командой `First obj` объявляется поле - объектная переменная класса `First`. Как и в случае с полем – переменной массива, в данном случае поле является всего лишь ссылкой на объект, которого пока нет, и который сам по себе не появляется. Его необходимо создать. Создается объект класса `First` для поля `obj` в конструкторе класса `Second`. У конструктора один целочисленный аргумент, который определяет размер поля-массива в объекте, на который ссылается поле `obj`. Сам объект создается командой `obj=new First(n)`. Командой `obj.show()` содержимое поля-массива объекта `obj` выводится на консоль. Все это происходит, напомним, при вызове конструктора класса `Second`. Собственно, на конструкторе код класса `Second` заканчивается.

В главном методе командой `Second[] ObjAr=new Second[count]` создается массив `ObjAr` из объектных переменных класса `Second`. Размер массива на этапе создания задается целочисленной переменной `count` (в данном случае значение равно `5`). Далее с помощью оператора цикла для каждой из объектных переменных в массиве `ObjAr` создается объект класса `Second` и ссылка на этот объект записывается в соответствующую переменную. Для индексной переменной `i`, которая в операторе цикла пробегает значения от `0` до `ObjAr.length-1` включительно, значение элементу массива `ObjAr[i]` присваивается командой `ObjAr[i]=new Second(3*i+4)`. В результате в объектах класса `Second`, на которые ссылаются объектные переменные в массиве `ObjAr`, поле `obj` ссылается на объекты класса `First` с массивами размера `4`, `7`, `10` и так далее (размер каждого следующего массива на 3 элемента больше, чем у предыдущего). Поскольку при создании объектов `Second` автоматически отображается значение соответствующего массива ("спрятанного" в виде поля в объекте класса `First`, на который ссылается поле `obj` объекта класса `Second`), то в консольном окне появится следующее сообщение:

```
1 1 2 3
1 1 2 3 5 8 13
1 1 2 3 5 8 13 21 34 55
1 1 2 3 5 8 13 21 34 55 89 144 233
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Еще раз отметим, что при работе с массивами, в том числе и массивами объектов, и полями-массивами важно понимать разницу между объектными переменными (переменными массива) и непосредственно объектами (массивами).

Анонимные объекты

Нередко в программах используются *анонимные объекты*. Анонимные объекты – это объекты, ссылка на которые не записывается ни в одну объектную переменную. Несмотря на это ужасное обстоятельство, объект не перестает быть объектом и его можно использовать. Правда, учитывая, что осязаемая ссылка на объект отсутствует, использовать анонимный объект следует быстро, пока он не потерялся.

На заметку:

При создании анонимных объектов команда создания объекта есть (инструкция с оператором `new`), а команды записи (присваивания) ссылки на объект в объектную переменную нет. Несмотря на это, из таких объектов можно, например, вызывать методы или указывать анонимный объект аргументом метода.

Пример создания и использования анонимного приведен в листинге 7.6.

Листинг 7.6. Анонимные объекты

```
import javax.swing.*;
// Класс для создания на его основе
// анонимного объекта:
class MyClass{
   // Поле - переменная массива:
   int[] nums;
   // Конструктор класса:
   MyClass(int n){
      // Создание массива:
      nums=new int[n];
      // Заполнение массива натуральными числами:
      for(int i=0;i<nums.length;i++)
         nums[i]=i+1;}
   // Метод для отображения элементов массива
   // в диалоговом окне:
   void show(){
      // Текст для отображения в диалоговом окне
      // (начальное значение):
      String text="Натуральные числа:\n";
      // Формирование текста:
      for(int i=0;i<nums.length;i++)
         text+=nums[i]+" ";
      // Отображение диалогового окна:
      JOptionPane.showMessageDialog(null,text);}
   }
public class AObjDemo{
   // Статический метод для отображения сообщения
```

```
// на основе объекта класса MyClass:
static void show(MyClass obj){
   // Текст (начальное значение)
   // для отображения в окне:
   String text="Элементы массива:\n";
   // Определение количества элементов в строке:
   int n=(int)Math.ceil(Math.sqrt(obj.nums.length));
   // Формирование текста:
   for(int i=0;i<obj.nums.length-1;i++){
      text+=obj.nums[i]+((i+1)%n==0?" >> дальше\n":" : ");}
   // Финальная "точка":
   text+=obj.nums[obj.nums.length-1]+". конец";
   // Отображение сообщения:
   JOptionPane.showMessageDialog(null,text);
   }
public static void main(String[] args){
   // Вызов метода из анонимного объекта:
   new MyClass(12).show();
   // Анонимный объект - аргумент метода:
   show(new MyClass(24));}
}
```

В программе описывается класс `MyClass`, на основе которого затем создаются анонимные объекты. У класса есть поле-ссылка на целочисленный массив `nums`. Массив создается в конструкторе класса. Конструктору в качестве аргумента передается целое число, которое определяет количество элементов в создаваемом объекте. Ссылка на массив в качестве значения присваивается полю `nums`, а сам массив заполняется натуральными числами. Наконец, в классе `MyClass` описан метод `show()`, которым значения элементов массива выводятся в диалоговом окне в одну строку.

В классе `AObjDemo` кроме главного метода программы описан статический метод `show()` (не путать с одноименным методом класса `MyClass`). Аргументом методу передается объект `obj` класса `MyClass`. На основании поля `nums` этого объекта формируется текстовое сообщение и выводится в диалоговом окне. Текст, кроме заголовка, содержит значения элементов поля-массива `nums`. Числовые значения выводятся построчно. Количество элементов в строке определяется командой `int n=(int)Math.ceil(Math.sqrt(obj.nums.length))`. Этой командой объявляется целочисленная переменная `n`, а в качестве значения ей присваивается округленное (в сторону увеличения) значение корня квадратного из количества элементов массива `nums` объекта `obj`. Ссылка на свойство `length` для массива `nums` из объекта `obj` выполняется в формате `obj.nums.length`. Для вычисления корня квадратного используем метод `sqrt()`. Этот статический метод вызывается из встроенного (то есть его не нужно создавать, он уже суще-

ствует и автоматически доступен) класса `Math`. Из этого же класса `Math` вызывается метод `ceil()`, которым в качестве значения возвращается ближайшее наименьшее целое число, не меньшее, чем аргумент метода (округление с увеличением). Особенность метода `ceil()` состоит в том, что хотя он и возвращает в качестве результата целое число, это целое число реализовано как значение типа `double`. Чтобы записать его в переменную типа `int`, используем инструкцию явного приведения типа.

После того, как определено количество элементов в строке, начинается формирование текста, который будет выводиться в окне сообщения. Для этого используем оператор цикла. В операторе цикла перебираются все элементы массива `obj.nums`, кроме последнего. Каждый элемент добавляется к текущему текстовому значению (переменная `text`). После элемента, если это не последний элемент в строке, добавляется двоеточие (окруженное пробелами). Если элемент последний, добавляется текст `">> дальше\n"`, содержащий, кроме прочего, инструкцию перехода к новой строке. Для определения, какой текст добавлять после элемента, используем тернарный оператор в конструкции `((i+1)%n==0?" >> дальше\n":" : ")`. В данном случае проверяется на предмет равенства нулю остатка от деления значения `(i+1)` на `n`. Последний элемент записывается в переменную `text` уже после завершения оператора цикла, и после этого элемента добавляется текст `". конец"`.

В главном методе программы командой `new MyClass(12).show()` создается анонимный объект, из которого вызывается метод `show()`. В результате выполнения этой команды появляется диалоговое окно, представленное на рис. 7.7.

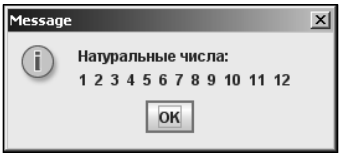


Рис. 7.7. Результат вызова метода из анонимного объекта

Затем командой `show(new MyClass(24))` отображается еще одно диалоговое окно, представленное на рис. 7.8.

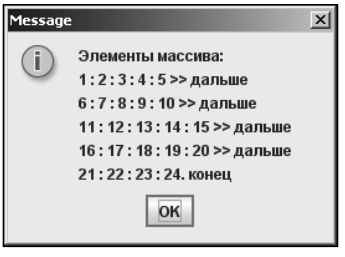


Рис. 7.8. Результат вызова метода с аргументом - анонимным объектом

В данном случае при вызове метода аргументом этому методу передается анонимный объект, который создается командой `new MyClass(24)`.

Хотя использование анонимных объектов вносит некоторую экстравагантность в программный код, использовать их имеет смысл лишь в тех случаях, когда предполагается одноразово использовать лишь один объект класса.

Внутренние классы

Один класс может описываться внутри другого класса. Класс, который описывается внутри другого класса, называется *внутренним*. Класс, в котором описывается внутренний класс, называется *внешним*, или *классом-контейнером*. Главная особенность внутреннего класса состоит в том, что в нем доступны поля и методы внешнего класса. Вместе с тем, поля и методы внутреннего класса не доступны во внешнем. Недоступны в том смысле, что к ним нельзя обратиться напрямую. Однако при этом можно создать объект внутреннего класса и из него вызвать метод внутреннего класса или обратиться к полю внутреннего класса.

🕮 На заметку:

Другими словами, если во внешнем классе описан внутренний класс, то в программном коде этого внутреннего класса можно обращаться к полям и методам внешнего класса. Однако поля и методы внутреннего класса *не являются* полями и методами внешнего класса. Для их использования в программном коде методов внешнего класса обычно сначала создается объект внутреннего класса, у которого есть поля и методы, описанные во внутреннем классе.

Пример объявления и использования внутреннего класса приведен в листинге 7.7.

Листинг 7.7. Внутренний класс

```
import javax.swing.*;
// Внешний класс:
class TheBig{
// Текстовое поле внешнего класса:
String name;
// Поле внешнего класса - объектная
// переменная внутреннего класса:
TheSmall ID;
// Конструктор внешнего класса:
TheBig(String name,int n){
// Значение текстового поля внешнего класса:
this.name=name;
// Создание объекта внутреннего класса:
ID=new TheSmall(n);
```

```
// Вызов метода внутреннего класса
// через объект внутреннего класса:
ID.show();}
// Метод внешнего класса.
// Возвращает текстовую строку с именем объекта
// внешнего класса:
String getName(){
String txt="Имя объекта: "+name+".\n";
return txt;}
// Внутренний класс:
   class TheSmall{
   // Поле внутреннего класса - переменная массива:
   int[] code;
   // Конструктор внутреннего класса:
   TheSmall(int n){
   // Создание массива:
   code=new int[n];
   // Заполнение массива случайными числами (от 0 до 9):
   for(int i=0;i<code.length;i++)
   code[i]=(int)(10*Math.random());}
   // Метод внутреннего класса.
   // Возвращает строку с кодом объекта
   // внутреннего класса:
   String getCode(){
   // Начальная строка:
   String txt="Код объекта: |";
   // Формирование полного текста:
   for(int i=0;i<code.length;i++)
   txt+=code[i]+"|";
   // Результат:
   return txt;}
   // Метод для отображения сообщения с именем
   // объекта внешнего класса и кодом объекта
   // внутреннего класса:
   void show(){
   // Текстовая строка:
   String txt=getName()+getCode();
   // Отображение сообщения:
   JOptionPane.showMessageDialog(null,txt);}
   }
}
// Класс с главным методом программы:
public class InnerClassDemo{
public static void main(String[] args){
// Анонимный объект внешнего класса:
new TheBig("БОЛЬШОЙ",7);}
}
```

В результате выполнения программы на экране появляется диалоговое окно, схожее с тем, что представлено на рис. 7.9.

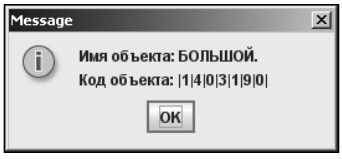


Рис. 7.9. Такое окно отображается в результате выполнения программы с внутренним классом

Отличие может быть только в коде объекта (вторая текстовая строка в окне сообщения). Теперь разберем программный код в листинге 7.7.

Там объявляется класс `TheBig`, который, как окажется впоследствии, содержит внутренний класс `TheSmall`. У класса `TheBig` есть банальное текстовое поле `name`, а также поле с названием `ID`, которое является объектной переменной внутреннего класса `TheSmall`. Еще у внешнего класса есть конструктор и метод `getName()`. Метод `getName()` в качестве результата возвращает текстовое значение, которое состоит из фразы `Имя объекта` и значения текстового поля `name`. В теле метода командой `String txt="Имя объекта: "+name+".\n"` создается текстовая переменная `txt`, которая командой `return txt` возвращается в качестве результата метода.

Конструктору класса `TheBig` передаются два аргумента: текстовый и целочисленный. Текстовый в качестве значения присваивается полю `name`. Целочисленный нужен для передачи в качестве аргумента конструктору внутреннего класса. В частности, в описании конструктора класса `TheBig` первый аргумент обозначен как `name`, второй – как `n`. Командой `this.name=name` полю `name` в качестве значения присваивается ссылка на объект, переданный аргументом конструктору. Поскольку имя поля и имя аргумента в данном случае совпадают, при ссылке на поле используется ключевое слово `this`.

На заметку:

Здесь следует учесть одно немаловажное обстоятельство. Текстовая строка (в данном случае) – объект класса `String`. Аргументом конструктору передается объектная переменная, значение которой – ссылка на объект. После выполнения команды `this.name=name` поле `name` объекта будет ссылаться на *тот же объект*, что и объектная переменная, переданная аргументом конструктору. Это далеко не то же самое, что скопировать текст! Основы работы с классом `String` описаны в *главе 9*.

Следующей командой `ID=new TheSmall(n)` создается объект внутреннего класса и ссылка на него записывается в поле `ID`. Командой `ID.show()`

из вновь созданного объекта вызывается метод `show()`. Чтобы понять, как выполняются эти две команды и в чем их особенность, имеет смысл рассмотреть программный код внутреннего класса.

У внутреннего класса `TheSmall` есть поле `code`. Это переменная массива, она ссылается на массив из целых чисел. На какой именно массив, определяется в конструкторе. Конструктору в качестве аргумента передается целое число, определяющее, сколько элементов будет в массиве. Создание массива и присваивание ссылки на него полю `code` внутреннего класса выполняется командой `code=new int[n]`. Далее в операторе цикла массив заполняется случайными числами в диапазоне значений от `0` до `9` включительно. Для генерирования случайного числа используем метод `random()` из встроенного класса `Math` (инструкция `Math.random()`). Проблема, однако, в том, что методом генерируется случайное действительное число (тип `double`) в диапазоне от `0` (включительно) до `1` (не включая). Чтобы из такого числа получить целое число, умножаем результат вызова метода `Math.random()` на `10`. Получаем действительное число от `0` включительно и меньшее `10`. Отбрасываем дробную часть (преобразуем действительное число в целое с помощью инструкции явного приведения типов `(int)`). В результате получаем целое число в диапазоне от `0` до `9` включительно.

Метод `getCode()` класса `TheSmall` возвращает в качестве значения текстовую строку (тип `String`). Эта строка содержит вспомогательный текст и код, который представляет собой последовательность цифр, записанных в массив `code`. В теле метода командой `String txt="Код объекта: |"` объявляется текстовая строка с начальным значением, к которому последовательно дописываются в рамках оператора цикла цифры из массива `code`. В качестве разделителя между цифрами используется вертикальная черта. Полученное текстовое значение (переменная `txt`) командой `return txt` возвращается в качестве результата метода.

Еще один метод в классе `TheSmall` предназначен для отображения сообщения в диалоговом окне. Это метод `show()`. В методе командой `String txt=getName()+getCode()` создается текстовая строка, которая отображается в диалоговом окне с помощью команды `JOptionPane.showMessageDialog(null,txt)`. Обращаем внимание читателя, что в данном случае в методе внутреннего класса вызывается метод внешнего класса.

На заметку:

Обратную процедуру, то есть вызвать из метода внешнего класса метод внутреннего класса, произвести не получится. Так, в конструкторе внешнего класса метод `show()` внутреннего класса вызывался из объекта `ID` внутреннего класса. Вызвать метод напрямую (без объекта) не получится.

Аргументы командной строки

При вызове программы ей можно передавать аргументы. Это так называемые *аргументы командной строки*. Есть два момента, на которые следует обратить внимание, что мы и сделаем (то есть обратим внимание). Во-первых, выясним, как эти аргументы передать при вызове программы. Во-вторых, рассмотрим, что с ними (с этими аргументами) можно делать в программе. И начнем как раз со второго вопроса.

В главном методе программы `main()` указан аргумент – массив текстовых значений. Он, этот массив, и содержит аргументы командной строки. Особенность массива в том, что все параметры (или аргументы) командной строки интерпретируются как текстовые. Поэтому основной код, связанный с использованием аргументов командной строки, обычно связан с преобразованием того или иного аргумента из текстового в нужный (числовой, например). Рассмотрим пример, представленный в листинге 7.8.

Листинг 7.8. Аргументы командной строки

```
import javax.swing.*;
// Класс для записи персональных данных:
class Fellow{
   // Имя:
   String name;
   // Фамилия:
   String surname;
   // Возраст:
   int age;
   // Вес (в килограммах):
   double weight;
   // Конструктор класса:
   Fellow(String name,String surname,int age,double weight){
      this.name=name;
      this.surname=surname;
      this.age=age;
      this.weight=weight;
      show();}
   // Метод для отображения сообщения:
   void show(){
      String str="Имя: "+name+"\n";
      str+="Фамилия: "+surname+"\n";
      str+="Возраст: "+age+" лет\n";
      str+="Вес: "+weight+" кг";
      JOptionPane.showMessageDialog(null,str);}
   }
public class InputArgsDemo{
```

```
public static void main(String[] args){
   // "Считывание" параметров командной строки:
   String name=args[0];
   String surname=args[1];
   int age=Integer.parseInt(args[2]);
   double weight=Double.parseDouble(args[3]);
   // Создание анонимного объекта класса Fellow:
   new Fellow(name,surname,age,weight);}
}
```

В программе объявляется класс `Fellow`, который предназначен для хранения персональных данных, таких, как имя, фамилия, возраст и вес. Для этого у класса описывается текстовое поле `name` (имя), текстовое поле `surname` (фамилия), целочисленное поле `age` (возраст) и `double`-поле `weight` (вес). В конструкторе класса, в соответствии с переданными конструктору аргументами, полям присваиваются значения. Затем вызывается метод `show()`, которым отображается сообщение с информацией, считанной из полей объекта.

В главном методе программы в классе `InputArgsDemo` основная часть кода связана с обработкой аргументов командной строки. В частности, аргумент метода `main()` имеет название `args`, и это, как уже упоминалось, текстовый массив (в качестве идентификатора типа указано `String[]`). То есть работаем с элементами массива `args`. Как и для любого другого массива, количество элементов в массиве `args` определяется свойством `length`. Однако в данном случае мы предполагаем, что при запуске программы в командной строке передается ровно четыре аргумента (имя, фамилия, возраст и вес), поэтому размер массива `args` не проверяем, а полагаемся на пунктуальность пользователя. Первым передается имя пользователя (или кого-то там еще). Это текстовый аргумент. Поэтому для запоминания имени используем команду `String name=args[0]`. Второй аргумент – фамилия. Это тоже текст. Этот текст запоминаем в переменной `surname`, для чего используем команду `String surname=args[1]`. С третьим аргументом хуже. Это должно быть целое число, а считывается оно как текст. Поэтому используем для запоминания этого аргумента команду `int age=Integer.parseInt(args[2])`. Здесь текстовый аргумент `args[2]` передается аргументом статическому методу `parseInt()` класса-оболочки `Integer`. В результате текстовое представление целого числа преобразуется в целое число. Аналогично обстоят дела с последним аргументом, только его необходимо преобразовать в значение типа `double`. Для этого используем команду `double weight=Double.parseDouble(args[3])`. Здесь преобразование выполняется статическим методом `parseDouble()`, вызываемым из класса-оболочки `Double`. Наконец, командой `new Fellow (name,surname,age,weight)` создаем анонимный объект, в результате

чего отображается сообщение с персональной информацией, переданной аргументами командной строки. Осталось только выяснить, как эти самые аргументы передать.

Если программа компилируется и запускается из командной строки, то ситуация выглядит следующим образом. Допустим, после компиляции программного кода из листинга 7.8 получаем, кроме прочего, файл `InputArgsDemo.class`. Тогда команда запуска на выполнение программы с аргументами `Александр`, `Иванов`, `56` и `78.4` может выглядеть так (между аргументами указывается пробел):

```
java.exe InputArgsDemo Александр Иванов 56 78.4
```

Но это путь смелых. Умные, скорее всего, будут использовать интегрированную среду разработки NetBeans. В среде NetBeans для запуска программы с аргументами достаточно выполнить следующие нехитрые действия.

Во-первых, выбираем команду **Run ▶ Set Project Configuration ▶ Customize** (рис. 7.10).

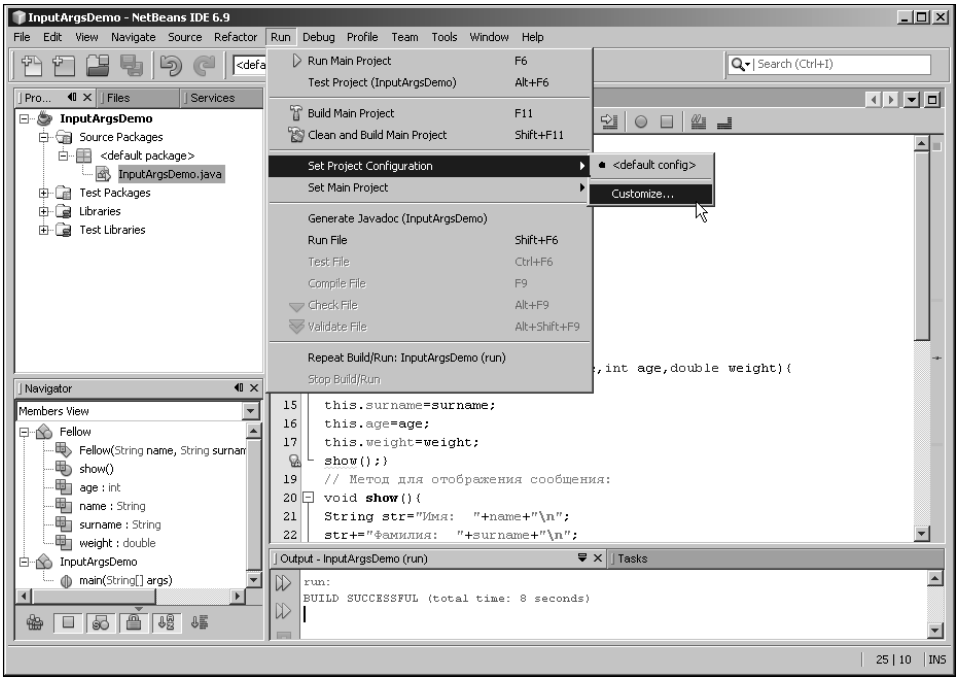


Рис. 7.10. Для ввода аргументов командной строки в NetBeans выбираем команду ***Run ▶ Set Project Configuration ▶ Customize***

Открывается окно, представленное на рис. 7.11.

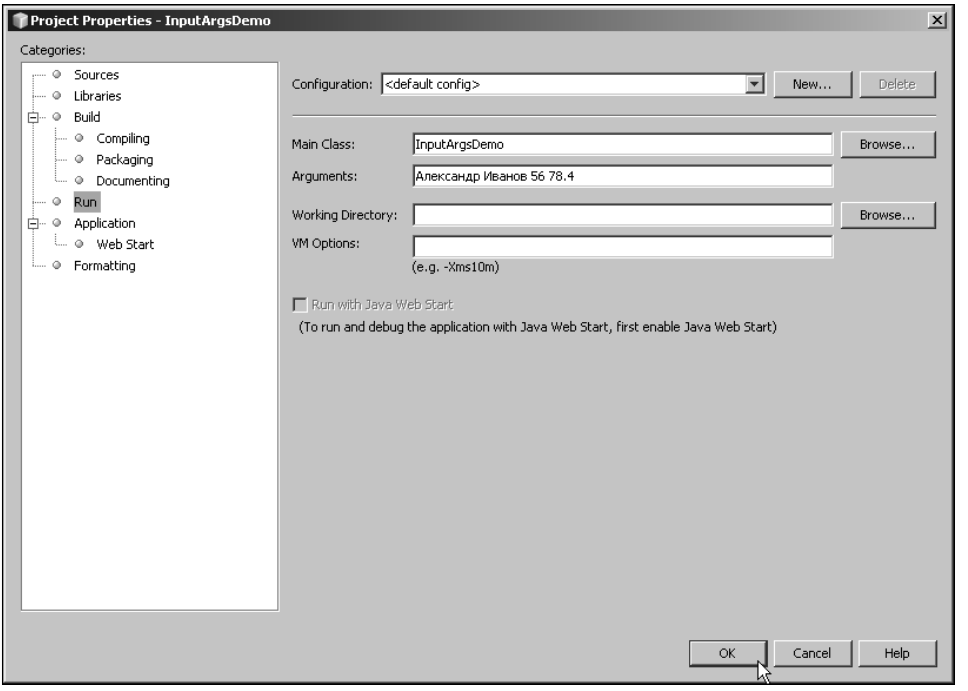


*Рис. 7.11. Аргументы командной строки вводятся через пробел в поле **Arguments***

В этом чудесном окне есть поле **Arguments**, в которое через пробел вводятся аргументы командной строки. Для подтверждения щелкаем кнопку **OK**. Если после этого запустить приложение на выполнение, программа будет запущена с соответствующими аргументами. В результате появляется диалоговое окно, представленное на рис. 7.12.

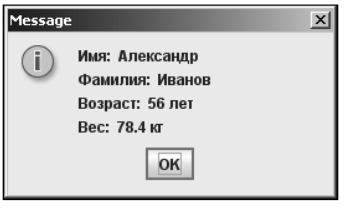


Рис. 7.12. Результат выполнения программы с передачей аргументов командной строки при запуске

Справедливости ради все же следует отметить, что передача аргументов описанным выше способом далеко не всегда себя оправдывает. Обычно разумнее реализовать процесс ввода нужной информации уже в процессе выполнения программы.

Резюме

1. В классе могут существовать статические поля и методы. Они описываются с ключевым словом `static`, являются общими для всех объектов класса и существуют независимо от того, сколько создано объектов и созданы ли они вообще.
2. Для обращения к статическому полю или вызова статического объекта указывают имя класса и через точку имя поля или метода.
3. Объекты могут передаваться аргументами методам, могут возвращаться в качестве результата. При этом следует помнить, что объект и переменная объекта - не одно и то же. Объект - это объект. Переменная объекта - ссылка на объект.
4. Существует два механизма передачи аргументов методам: по значению и по ссылке. При передаче аргументов по значению на самом деле в метод передаются их копии. При передаче аргументов по ссылке передаются непосредственно аргументы. Аргументы базовых типов передаются по значению. Объекты передаются по ссылке (за счет использования объектных переменных).
5. Ссылку на объект не обязательно записывать в объектную переменную. В этом случае получаем анонимный объект.
6. Один класс может описываться внутри другого. Внутренний класс имеет доступ ко всем полям и методам класса-контейнера.
7. Если программа запускается с параметрами, то эти параметры передаются аргументом методу `main()` в качестве текстового массива.

Глава 8
Наследование, интерфейсы и пакеты

- Я тоже похож на своего отца...
- Так ведь тут все дело в том, какой отец. Я - сын лейтенанта Шмидта.
(Из к/ф "Золотой теленок")

В этой главе обсуждается несколько тем, которые объединены одной общей идеей – наследованием. *Наследование* является одним из фундаментальных механизмов ООП. **Механизм состоит в том, что на основе уже существующих классов можно создавать новые классы, которые получают, или наследуют, свойства исходных классов.** Такой подход имеет несколько преимуществ. Перечислим только некоторые из них. В первую очередь обеспечивается высокая степень прямой и обратной совместимости программного кода. То есть создание новых классов возможно без внесения изменений в уже существующие. Модификация уже существующих классов автоматически учитывается в новых классах. Во-вторых, реализация программного кода на принципах наследования повышает его гибкость и снижает вероятность совершения ошибок. Теперь подробности.

Основы наследования

Представим, что у нас есть класс, красивый и функциональный. Но мы хотим создать еще более красивый и функциональный класс. При создании нового класса за основу берем уже существующий класс. В принципе, есть два пути. Первый состоит в том, чтобы внести изменения в программный код уже существующего класса. Если этот класс еще нигде не использовался, то в принципе так можно поступить. Но, скорее всего, окажется, что старый класс уже использовался в каких-то проектах. То есть старый класс проигнорировать нельзя, и он продолжит свое существование. Плюс еще новый класс. Если каждый раз идти указанным путем, классы будут множиться, как кролики.

Второй способ создания нового класса на основе уже существующего состоит в использовании механизма *наследования*. В этом случае при создании нового класса с помощью специальных синтаксических конструкций делается указание на автоматическое включение содержимого исходного класса в новом классе. Хотя при этом и старый, и новый классы существуют раздельно, но они в известном смысле взаимозависимы. Особенность этой зависимости такова, что позволяет выстраивать целую иерархию классов,

достаточно удобную в использовании. Именно на методах реализации наследования остановимся в этом разделе.

В первую очередь определимся с терминологией. Класс, на основе которого создается новый класс, называется *родительским* классом, или *суперклассом*. Новый класс, который создается на основе суперкласса, называется *подклассом*. Как создать на основе суперкласса подкласс? Очень просто. Для этого в сигнатуре подкласса после ключевого слова `class` и имени подкласса указывается ключевое слово `extends` и имя суперкласса. Таким образом, синтаксис объявления подкласса имеет такой вид:

```
class имя_подкласса extends имя_суперкласса{
   код_подкласса;
   }
```

Непосредственно код подкласса описывается как в обычном классе. При этом следует иметь в виду, что в подклассе автоматически известны и доступны поля и методы суперкласса (все, за исключением *закрытых*, с которыми мы еще не имели дела).

На заметку:

Наследуются те члены суперкласса, которые не являются *закрытыми*. Те члены классов, с которыми мы имели дело до этого, были по умолчанию *открытыми*. Это особенность языка Java. Здесь по умолчанию члены класса являются открытыми, в отличие, например, от языка C++, в котором по умолчанию члены класса являются закрытыми. Как объявляются закрытые члены класса и в чем их особенность, объясняется несколько позже. Кроме открытых и закрытых членов класса, есть еще *защищенные*. Они также описываются в этой главе. До тех пор, пока речь не зашла о закрытых членах класса, будем исходить из того, что *все* члены суперкласса наследуются в подклассе.

Фактически, с некоторыми ограничениями, речь идет о том, что код суперкласса автоматически неявно каптируется в программный код подкласса. И хотя поля и методы суперкласса в подклассе не описаны, они там есть и к ним можно обращаться и использовать, как если бы они были явно описаны в подклассе. Рассмотрим пример, представленный в листинге 8.1.

Листинг 8.1. Наследование классов

```
// Суперкласс:
class SuperClass{
   // Поля суперкласса:
   int number;
   String name;
   // Метод суперкласса:
   void setNumber(int n){
      number=n;}
```

```
   }
// Подкласс:
class SubClass extends SuperClass{
   // Поле подкласса:
   char symbol;
   // Метод подкласса:
   void setAll(int n,String txt,char s){
      // Вызов метода, описанного в суперклассе:
      setNumber(n);
      // Обращение к полю, описанному в суперклассе:
      name=txt;
      // Присваивание значения полю подкласса:
      symbol=s;}
   // Метод подкласса:
   void show(){
      // Отображение значений полей подкласса:
      System.out.println("Числовое поле: "+number);
      System.out.println("Текстовое поле: "+name);
      System.out.println("Символьное поле: "+symbol);}
   }
// Класс с главным методом программы:
public class SubClassDemo{
   // Главный метод программы:
   public static void main(String[] args){
      // Создание объекта подкласса:
      SubClass obj=new SubClass();
      // Присваивание полям объекта значений:
      obj.setAll(100,"ТЕКСТ",'A');
      // Отображение значений полей:
      obj.show();
      // Изменение значений полей объекта:
      obj.setNumber(20);
      obj.name="НОВЫЙ";
      // Отображение значений полей:
      obj.show();}
   }
```

Сначала в программе описан класс `SuperClass`, на основе которого затем будет создан новый класс `SubClass`. То есть класс `SuperClass` является суперклассом для класса `SubClass` (который, в свою очередь, является подклассом для класса `SuperClass`).

📖 На заметку:

Понятие суперкласса является относительным, поскольку класс может быть суперклассом по отношению к одному классу и не быть суперклассом по отношению к какому-то другому классу.

Подкласс создается на основе одного и только одного класса, поскольку в Java запрещено *многократное* наследование, то есть создание нового класса одновременно с использованием нескольких уже существующих. Вместе с тем, *многоуровневое* наследование разрешено. В этом случае подкласс суперкласса сам может быть суперклассом для другого подкласса. То есть допускается такая схема создания классов: на основе класса A путем наследования создается класс B, на основе класса B создается класс C и так далее.

В суперклассе `SuperClass` описано два поля: целочисленное поле `number` и текстовое поле `name`. Также в суперклассе описан метод `setNumber()`, с помощью которого можно задать значение поля `number` (значение, присваиваемое полю `number`, передается аргументом методу `setNumber()`).

Класс `SubClass` создается на основе класса `SuperClass` путем наследования, о чем свидетельствует инструкция `extends SuperClass` в сигнатуре (заголовке) класса `SubClass`. Непосредственно в теле класса `SubClass` описано символьное поле `symbol`. Следует также помнить, что в классе `SubClass` из класса `SuperClass` наследуются поля `number`, `name` и метод `setNumber()`. Поэтому, например, в методе `setAll()`, описанном в классе `SubClass`, вполне законным является обращение к этим полям и методам. Другими словами, ситуация абсолютно такая же, как если бы поля `number` и `name`, а также метод `setNumber()` были описаны непосредственно в классе `SubClass`. Например, командой `setNumber(n)` в теле метода `setAll()` присваивается значение полю `number`. Значение полю `name` присваивается командой `name=txt`, а значение полю `symbol`, описанному непосредственно в подклассе, присваивается командой `symbol=s`. Здесь `n`, `txt` и `s` обозначают аргументы метода `setAll()`. Обращение к унаследованным полям `number` и `name` и "родному" полю `symbol` есть и в методе `show()` класса `SubClass`. Методом в консольное окно выводятся значения полей объекта подкласса.

К унаследованным полям и методам подкласса можно обращаться как к обычным полям и методам и извне кода подкласса. Пример такого обращения можно найти в главном методе программы `main()`. В частности, после создания командой `SubClass obj=new SubClass()` объекта подкласса командой `obj.setAll(100,"ТЕКСТ",'A')` полям объекта присваиваются значения, а командой `obj.show()` выполняется вывод значений полей на консоль. Затем командами `obj.setNumber(20)` и `obj.name="НОВЫЙ"` значения унаследованных полей подкласса меняются, и командой `obj.show()` еще раз выполняется вывод значений полей объекта подкласса в консольное окно. Результат выполнения программы выглядит так:

```
Числовое поле: 100
Текстовое поле: ТЕКСТ
Символьное поле: A
```

```
Числовое поле: 20
Текстовое поле: НОВЫЙ
Символьное поле: A
```

Рассмотренный пример исключительно прост, и он иллюстрирует лишь базовые принципы наследования классов. Возможны и более запутанные и замысловатые ситуации. Рассмотрим вопрос о создании конструктора подкласса.

Конструктор подкласса

Если с конструктором суперкласса все более-менее ясно, поскольку суперкласс является фактически обычным классом, то с созданием конструктора подкласса дела обстоят несколько сложнее. Чтобы очертить проблему, обратим внимание вот на что. Дело в том, что при создании объекта подкласса автоматически сначала вызывается конструктор суперкласса, и уже после этого непосредственно конструктор подкласса. Если конструктору суперкласса аргументы передавать не нужно, то все выглядит не так уж и плохо. Но вот если конструктор суперкласса должен получать аргумент или аргументы и без этого не обойтись никак, то тут есть о чем призадуматься.

Выход из ситуации такой. При создании конструктора подкласса необходимо предусмотреть способ вызова конструктора суперкласса. Для этого в код конструктора подкласса включается инструкция `super` с круглыми скобками, в которых перечисляются аргументы, которые передаются конструктору суперкласса. Инструкция `super` должна быть первой командой в коде конструктора подкласса. Пример создания подкласса с описанием конструктора подкласса приведен в листинге 8.2.

Листинг 8.2. Конструктор подкласса

```
// Суперкласс:
class A{
   // Поля суперкласса:
   int number;
   char symbol;
   // Конструктор суперкласса без аргументов:
   A(){
      number=0;
      symbol='A';
      System.out.println("Конструктор без аргументов!");
      show();}
   // Конструктор суперкласса с одним аргументом:
   A(int n){
      number=n;
      symbol='B';
```

```
      System.out.println("Конструктор с одним аргументом!");
      show();}
   // Конструктор суперкласса с двумя аргументами:
   A(int n,char s){
      number=n;
      symbol=s;
      System.out.println("Конструктор с двумя аргументами!");
      show();}
   // Метод для отображения полей:
   void show(){
      System.out.println("Поля "+number+" и "+symbol+".");}
   }
// Подкласс:
class B extends A{
   // Текстовое поле подкласса:
   String text;
   // Конструктор подкласса без аргументов:
   B(){
      // Вызов конструктора суперкласса (без аргументов):
      super();
      text="Без аргументов";
      showText();}
   // Конструктор подкласса с одним аргументом:
   B(String txt){
      // Вызов конструктора суперкласса (с одним аргументом):
      super(txt.length());
      text=txt;
      showText();}
   // Конструктор подкласса с тремя аргументами:
   B(int n,char s,String txt){
      // Вызов конструктора суперкласса (с двумя аргументами):
      super(n,s);
      text=txt;
      showText();}
   // Метод для отображения текстового поля:
   void showText(){
      System.out.println("Текстовое поле \""+text+"\".");}
   }
public class SubConstrDemo{
   public static void main(String[] args){
      // Создание объектов (анонимных) подкласса.
      // Используются разные конструкторы:
      new B();
      new B("Один аргумент");
      new B(100,'F',"Три аргумента");}
   }
```

Суперкласс называется A, а на его основе создается подкласс, который называется B. У суперкласса A два поля: целочисленное поле number и символьное поле symbol. Для суперкласса описаны конструкторы без аргументов, с одним аргументом и с двумя аргументами. При вызове конструктора суперкласса, помимо присваивания значения полям, выводится консольное сообщение соответствующего содержания. Для отображения значений полей используем метод show(), который вызывается непосредственно из конструктора.

В подклассе описывается еще одно текстовое поле text, а также метод showText(), которым осуществляется вывод значения текстового поля на консоль. В классе описаны три варианта конструктора: без аргументов, с одним текстовым аргументом и с тремя аргументами. В каждом из этих конструкторов первой командой вызывается конструктор базового класса. Так, в конструкторе подкласса без аргументов командой super() вызывается конструктор суперкласса без аргументов. В конструкторе подкласса с одним аргументом (текстовый аргумент txt) вызывается конструктор суперкласса с одним аргументом, для чего используется команда super(txt.length()). Здесь аргументом конструктору суперкласса передается количество символов в текстовом аргументе конструктора подкласса. Для вычисления количества символов в текстовой переменной вызывается встроенный метод length().

На заметку:

Напомним, что текстовая переменная, и переменная txt в частности, – это объектная переменная класса String. В классе есть метод length(), который в качестве значения возвращает количество символов в тексте, который содержится в объекте, из которого вызывается метод. Другими словами, чтобы узнать, сколько символов в тексте, "спрятанном" в txt, достаточно воспользоваться инструкцией txt.length(). Подробнее методы работы с текстом еще будут обсуждаться. Здесь это лишь иллюстративный пример.

В конструкторе подкласса с тремя аргументами первой командой super(n,s) вызывается конструктор суперкласса с двумя аргументами. Здесь n и s – первый и второй аргументы конструктора подкласса соответственно.

В главном методе программы последовательно создается три анонимных объекта подкласса. При этом используются разные конструкторы: без аргументов, с одним аргументом и с тремя аргументами. В результате в окне консоли появляется следующая серия сообщений:

```
Конструктор без аргументов!
Поля 0 и A.
Текстовое поле "Без аргументов".
Конструктор с одним аргументом!
Поля 13 и B.
```

```
Текстовое поле "Один аргумент".
Конструктор с двумя аргументами!
Поля 100 и F.
Текстовое поле "Три аргумента".
```

Если в программном коде конструктора подкласса явно не указать команду вызова конструктора суперкласса, то это совсем не означает, что конструктор суперкласса вызываться не будет. В этом случае автоматически вызывается конструктор суперкласса без аргументов. Если такой конструктор у суперкласса отсутствует, возникает ошибка.

Переопределение методов

Важное место в наследовании занимает *переопределение* методов. Суть переопределения методов связана с тем, что программный код унаследованного в подклассе метода может быть переопределен. В результате подкласс имеет такой же метод (с таким же названием), что и суперкласс, но выполняются они по-разному.

На заметку:

Не путайте *переопределение* метода и *перегрузку* метода. При перегрузке метода в классе создается несколько вариантов одного метода: название одно и то же, но сигнатуры *должны* отличаться. При переопределении нужно, чтобы был суперкласс и подкласс. Переопределенный в подклассе имеет такую же сигнатуру, что и метод в суперклассе.

Для переопределения метода в подклассе необходимо заново описать унаследованный метод в подклассе. Рассмотрим пример, представленный в листинге 8.3.

Листинг 8.3. Переопределение метода

```
// Суперкласс:
class A{
   int first;
   // Метод для присваивания значения
   // полю без аргумента:
   void set(){
      first=0;
      System.out.println("Нулевое значение поля.");
   }
   // Метод для присваивания значения
   // полю с одним аргументом:
   void set(int n){
      first=n;
      // Вызов метода для отображения поля:
```

```
      show();}
   // Метод для отображения поля:
   void show(){
      System.out.println("Поле "+first+".");}
      void showAll(){
      System.out.println("Все поля - на экран!");
      show();}
   }
// Подкласс:
class B extends A{
   // Еще одно поле:
   int second;
   // Переопределение варианта метода с одним аргументом:
   void set(int n){
      first=n;
      second=n;
      // Вызов переопределенного метода:
      show();}
   // Перегрузка метода:
   void set(int m,int n){
      second=n;
      // Вызов варианта метода из суперкласса:
      super.set(m);}
   // Переопределение метода для отображения полей:
   void show(){
      System.out.println("Поля "+first+" и "+second+".");}
   }
public class OverrideDemo{
   public static void main(String[] args){
      // Объект суперкласса:
      A objA=new A();
      // Методы суперкласса:
      objA.set();
      objA.set(100);
      // Объект подкласса:
      B objB=new B();
      // Методы подкласса:
      objB.set();
      objB.second=-1;
      objB.show();
      objB.set(200);
      objB.set(1,2);
      // Метод showAll():
      objA.showAll();
      objB.showAll();}
   }
```

В программе есть суперкласс A и подкласс B. У класса A всего одно целочисленное поле, которое называется first. Также у класса A есть перегруженный метод set() для присваивания значения полю, метод show() для отображения значения поля и еще один метод showAll(), который мы используем для иллюстрации одного очень интересного механизма, связанного с переопределением методов. Код этих методов имеет некоторые особенности, поэтому обсудим их. Во-первых, метод set() имеет два варианта – без аргументов и с одним аргументом. Если метод вызывается без аргументов, то полю first присваивается нулевое значение, а в консоль выводится сообщение "Нулевое значение поля.". Если метод set() вызывается с одним целочисленным аргументом, то этот аргумент определяет значение поля first, а затем с помощью метода show() отображается значение поля. В самом методе show() вызывается команда вывода на консоль текстовой фразы "Поле " и непосредственно значения поля first.

Методом showAll() сначала в консоль отображается текстовое сообщение "Все поля – на экран!", после чего вызывается метод show(). Поэтому, пока не вступило в силу наследование, наличие метода showAll() особого интереса не вызывает – метод делает практически то же, что и метод show(). Но ситуация еще изменится (об этом мы будем говорить при обсуждении результатов выполнения программы).

В подклассе B добавляется еще одно числовое поле, которое называется second. В классе автоматически наследуется поле first и вариант метода set() без аргументов. Вариант метода set() с одним аргументом *переопределяется*. В переопределенном методе set() с одним аргументом (название аргумента n) командами first=n и second=n полям присваиваются одинаковые значения. Далее вызывается метод show(). Важно то, что метод show() переопределяется в классе B. Поэтому в данном случае вызывается тот вариант метода show(), который описан (переопределен) в классе B. Переопределен метод show() так, что для объектов класса B методом отображаются значения обоих полей (то есть first и second). Кроме этого, в классе B перегружается метод set() для случая, когда ему передаются два аргумента – значения полей first и second. Особенность метода set() с двумя аргументами связана с тем, что сначала присваивается значение второму полю (полю second), а затем вызывается метод set() с одним аргументом. Но только тот вариант метода, который описан в классе A.

Если мы просто напишем инструкцию вида set(*аргумент*), то это будет вызов той версии метода set(), который переопределен в подклассе. Чтобы показать, что нам нужен "старый" метод set(), описанный в суперклассе, перед именем метода указываем инструкцию super. В результате получаем инструкцию super.set(*аргумент*), которой вызывается версия метода

set() с одним аргументом, описанная в классе A. На этом особенность данного кода не ограничивается, но сейчас лучше в подробности не вдаваться.

На заметку:

Таким образом, ключевое слово super может использоваться не только для вызова конструктора суперкласса, но и для ссылки на методы (и поля!) суперкласса (используется в случае переопределения методов или перекрытия наследуемых полей). Перекрытие наследуемых полей – это ситуация, когда в подклассе описано поле с таким же именем, что и наследуемое поле из суперкласса. Эту ситуацию рассмотрим позже.

В главном методе программы в классе OverrideDemo сначала создается объект objA класса A, после чего проверяется работа метода set() с разными аргументами. Затем создается объект objB класса B и проверяется работа методов set() и show(), но уже для класса B. Наконец, командами objA.showAll() и objB.showAll() наследуемый в классе B метод showAll() вызывается последовательно из объекта класса A и объекта класса B. Результат выполнения программы имеет следующий вид:

```
Нулевое значение поля.
Поле 100.
Нулевое значение поля.
Поля 0 и -1.
Поля 200 и 200.
Поля 1 и 2.
Все поля - на экран!
Поле 100.
Все поля - на экран!
Поля 1 и 2.
```

Обсудим его подробнее. Первое сообщение "Нулевое значение поля." отображается в результате выполнения команды objA.set(). Здесь все просто. В результате выполнения команды objA.set(100) получаем сообщение "Поле 100.". Данное сообщение — следствие вызова метода show() в теле метода set() (вариант с одним аргументом).

Когда выполняется команда objB.set(), вызывается тот же метод set() без аргументов, который описан в классе A и который наследуется без изменений классом B. Поэтому снова видим сообщение "Нулевое значение поля.". Поскольку при этом только поле first получает значение (нулевое), командой objB.second=-1 присваиваем значение и другому полю. Затем следует команда objB.show(). Здесь вызывается та версия метода show(), что описана в классе B. Поэтому на экране появляется сообщение "Поля 0 и -1.". Когда выполняется команда objB.set(200), вызывается переопределенная версия метода set() с одним аргументом. В

этом методе полям присваиваются одинаковые значения, и вызывается переопределенная версия метода show(). В результате получаем сообщение "Поля 200 и 200.". Далее, когда выполняется команда objB.set(1,2), вызывается вариант метода set(), в котором полям присваиваются значения, причем напомним схему выполнения этого метода. Полю second присваивается значение. Для присваивания значения полю first вызывается исходный, не переопределенный вариант метода set() с одним аргументом. Поле first получает свое значение. Но кроме этого вызывается еще и метод show() (см. код метода set() с одним аргументом в классе A). Хотя все это описано в классе A, поскольку исходный метод set() с двумя аргументами, в том месте, где нужно вызвать метод show(), вызывается переопределенная версия метода show(). Поэтому в консольном окне появляется сообщение "Поля 1 и 2.".

Другими словами, при вызове метода set(), описанного в суперклассе, из подкласса, в теле метода в соответствующих местах автоматически вызываются переопределенные версии других методов. Это свойство иллюстрируется на примере вызова метода showAll() из объектов разных классов. Напомним, в методе выводится сообщение "Все поля - на экран!", после чего вызывается метод show() (который переопределен в подклассе). Так вот, если метод showAll() вызывается из объекта суперкласса, используется версия метода show() из суперкласса. Если метод showAll() вызывается из объекта подкласса, используется версия метода show() из подкласса.

На заметку:

Как отмечалось выше, названия полей подкласса могут дублировать названия полей из суперкласса. Например, если в суперклассе A описано поле int number, а в подклассе B, который создается на основе класса A, также явно описано поле int number, то по умолчанию, если в классе B выполнить ссылку number, получим доступ к тому полю, что описано в этом подклассе. Чтобы получить доступ к полю, описанному в суперклассе, используем ссылку super.number.

Иногда бывает необходимо запретить наследование класса или переопределение метода. В этом случае соответственно в сигнатуре класса или сигнатуре метода необходимо указать ключевое слово final. Наличие ключевого слова final в сигнатуре метода запрещает его переопределение. Наличие ключевого слова final в сигнатуре класса запрещает на его основе создавать подклассы. Наличие ключевого слова final в описании переменной означает, что значение этой переменной не может быть изменено. Такая переменная фактически является константой, и значение ей должно быть присвоено при объявлении.

Закрытые члены класса

Выше мы уже делали намеки на то, что далеко не все члены суперкласса наследуются. Дальше есть рецепт, как этого добиться. А именно, чтобы член класса не наследовался, его необходимо объявить в суперклассе закрытым. Для этого достаточно в описании поля или метода указать ключевое слово `private`.

На заметку:

Здесь еще раз уместно напомнить, что в Java члены класса по умолчанию являются открытыми. Поэтому чтобы их закрыть, приходится предпринимать кое-какие действия.

Таким образом, члены (поля и методы), объявленные в суперклассе с ключевым словом `private`, в подклассе не наследуются. Смысл этого "не наследуются" поясним на простом, но показательном примере. Обратимся к листингу 8.4.

Листинг 8.4. Закрытые члены класса

```
// Суперкласс с закрытыми членами:
class Base{
   // Закрытое текстовое поле:
   private String name;
   // Закрытое числовое поле:
   private int number;
   // Конструктор суперкласса:
   Base(String name,int number){
      this.name=name;
      this.number=number;
      }
   // Закрытый метод суперкласса:
   private void show(){
      System.out.println("Объект с именем "+name);}
   // Открытый метод суперкласса:
   void showAll(){
      // Обращение к закрытому методу суперкласса:
      show();
      // Обращение к закрытому полю суперкласса:
      System.out.println("Числовое поле объекта равно "+number);}
   // Открытый метод суперкласса:
   void setAll(String name,int number){
      // Обращение к закрытым полям суперкласса:
      this.name=name;
      this.number=number;}
   }
```

```
// Подкласс:
class SubBase extends Base{
   // Конструктор подкласса:
   SubBase(String str,int num){
      // Вызов конструктора суперкласса:
      super(str,num);
      // Отображение закрытых "ненаследуемых" полей
      // с помощью наследуемого открытого метода:
      showAll();}
   }
public class PrivateDemo {
   public static void main(String[] args){
      // Создание объекта подкласса:
      SubBase obj=new SubBase("НОВЫЙ",1);
      // Изменение значений "несуществующих" полей объекта
      // подкласса:
      obj.setAll("ТОТ ЖЕ САМЫЙ",2);
      // Отображение значений "несуществующих" полей объекта
      //подкласса:
      obj.showAll();}
   }
```

В программе объявляется суперкласс с именем `Base` и затем на его основе создается подкласс с именем `SubBase`. В суперклассе `Base` объявлены два закрытых поля (с идентификатором `private`): текстовое `name` и целочисленное `number`. Конструктором этим полям присваиваются значения при создании нового объекта суперкласса. Конструктору передается два аргумента. Кроме того, в классе `Base` есть также три метода (один закрытый и два открытых), которые не возвращают результат. Это метод `setAll()`, принимающий два аргумента, которые присваиваются в качестве значений закрытым полям объекта. Закрытым методом `show()` на консольное окно выводится сообщение о значении текстового поля `name`. Аргументов у метода `show()` нет. Помимо этих методов, в классе `Base` есть еще один открытый метод `showAll()`. Этим методом в консольное окно выводятся два сообщения со значениями закрытых полей `name` и `number`. Причем для отображения значения текстового поля `name` в теле открытого метода `showAll()` вызывается закрытый метод `show()`.

На заметку:

С точки зрения программного кода в классе `Base` нет совершенно никакой разницы, имеем мы дело с закрытым или открытым членом класса. Все они доступны внутри класса. Разница между закрытыми и открытыми членами проявляется, только если мы пытаемся получить доступ к членам класса извне, то есть из другого класса.

На этом код класса `Base` исчерпан. На основе класса `Base` путем наследования создается его подкласс `SubBase`. В этом простом, но загадочном классе имеется только конструктор и унаследованные из класса `Base` открытые методы `setAll()` и `showAll()`.

У конструктора подкласса `SubBase` есть два аргумента. Эти аргументы передаются конструктору суперкласса, который, напомним, вызывается с помощью инструкции `super()`. После этого вызывается метод `showAll()`. Собственно, это все.

На заметку:

Если мы попытаемся внутри кода класса `SubBase` обратиться напрямую к полям `number` или `name`, или методу `show()`, получим сообщение об ошибке, поскольку в классе `SubBase` они не наследуются. Вместе с тем, метод `showAll()`, который наследуется и вызывается в конструкторе класса `SubBase`, содержит обращение к закрытым ненаследуемым членам суперкласса. Такой парадокс имеет простое объяснение и во многом связан с тем, что подразумевать под термином *не наследуется*, а также с тем, как создается объект подкласса. Для создания последнего сначала вызывается конструктор суперкласса, который выделяет место, в том числе и под закрытые члены суперкласса. Поэтому технически закрытые члены суперкласса в объекте подкласса присутствуют. Однако объект не имеет к ним прямого доступа. Однако доступ к этим членам имеют открытые методы, унаследованные из подкласса. Соответственно, подкласс "ничего не знает" о "ненаследуемых" членах суперкласса. Именно в таком смысле разумно интерпретировать термин "не наследуются": об этих членах в подклассе ничего не известно.

В главном методе программы командой `SubBase obj=new SubBase ("НОВЫЙ",1)` создается объект подкласса. Аргументы, переданные конструктору подкласса, предназначены для "ненаследуемых" из суперкласса полей. Далее с помощью команды `obj.setAll("ТОТ ЖЕ САМЫЙ",2)` выполняется изменение значений "несуществующих" полей. Здесь вызывается унаследованный из суперкласса метод `setAll()`. Наконец, отображение значений "несуществующих" полей выполняется командой `obj.showAll()`. В результате выполнения программы получаем следующие сообщения в окне консоли:

```
Объект с именем НОВЫЙ
Числовое поле объекта равно 1
Объект с именем ТОТ ЖЕ САМЫЙ
Числовое поле объекта равно 2
```

Первые два сообщения появляются в результате создания объекта подкласса. Это результат вызова метода `showAll()` в теле конструктора. Два другие сообщения появляются после вызова метода `showAll()` через объект подкласса из главного в главном методе программы.

На заметку:

Нет смысла обращаться к полям `number` и `name` или методу `show()` объекта подкласса `obj` в главном методе программы. Таких членов у объекта нет, и не нужно строить по этому поводу иллюзий. Более того, если бы мы создали в главном методе объект суперкласса, то к его закрытым полям и методам также нельзя было бы обратиться, хотя они у объекта были бы. Просто эти члены закрытые. Они доступны только внутри кода класса.

Объектные переменные суперклассов

У объектных переменных суперклассов есть одно очень полезное и важное свойство. Эти переменные могут ссылаться на объекты подклассов. Правда, доступ через объектную переменную суперкласса можно получить только к тем полям и методам подкласса, которые описаны в суперклассе, но от этого прелесть ситуации не уменьшается. Полезность указанной особенности объектных переменных суперклассов проиллюстрируем на примере, представленном в листинге 8.5.

Листинг 8.5. Объектные переменные суперклассов

```
// Суперкласс:
class Base{
   // Текстовое поле суперкласса:
   String name;
   // Конструктор суперкласса:
   Base(String name){
   this.name=name;}
   // Метод для отображения значения поля:
   void show(){
      System.out.println("Объект суперкласса: "+name);}
   // Метод выводит приветствие:
   void sayHello(){
      System.out.println("Всем привет!");}
   }
// Подкласс:
class SubBase extends Base{
   // Символьное поле подкласса:
   char code;
   // Конструктор подкласса:
   SubBase(String name,char code){
      super(name);
      this.code=code;}
   // Переопределение метода:
   void show(){
      System.out.println("Объект подкласса: "+name+". Код: "+code);}
   }
```

```
public class SubRefDemo{
   public static void main(String[] args){
      // Объектная переменная суперкласса:
      Base obj;
      // Ссылка на объект суперкласса:
      obj=new Base("Базовый");
      // Обращение к методам объекта суперкласса:
      obj.sayHello();
      obj.show();
      // Ссылка на объект подкласса:
      obj=new SubBase("Производный",'A');
      // Обращение к методам объекта подкласса:
      obj.sayHello();
      obj.show();}
   }
```

У суперкласса `Base` есть текстовое поле `name` и два метода. Методом `show()` в консоли отображается текстовое сообщение со значением текстового поля. Методом `sayHello()` отображается простое текстовое сообщение. Кроме этого, у суперкласса `Base` имеется конструктор с одним текстовым аргументом. Это значение, которое присваивается текстовому полю `name`.

На основе суперкласса `Base` создается подкласс `SubBase`. Подкласс `SubBase` наследует текстовое поле `name` суперкласса, а также его методы `sayHello()` и `show()`. Однако последний в подклассе переопределяется. Версия метода `show()` из подкласса `SubBase` отображает, кроме значения унаследованного текстового поля `name`, еще и значение символьного поля `code`, которое непосредственно описано в подклассе `SubBase`. У конструктора подкласса `SubBase` два аргумента: первый текстовый аргумент передается конструктору суперкласса, а второй символьный аргумент присваивается в качестве значения символьному полю `code`.

В главном методе программы командой `Base obj` объявляется объектная переменная `obj` суперкласса `Base`. Следующей командой `obj=new Base("Базовый")` в качестве значения этой переменной присваивается ссылка на вновь созданный объект суперкласса со значением `"Базовый"` текстового поля `name` объекта. Командами `obj.sayHello()` и `obj.show()` вызываются методы объекта суперкласса. В результате сначала в консольном окне появляется сообщение `"Всем привет!"`, а затем сообщение `"Объект суперкласса: Базовый"`.

После этого выполняется команда `obj=new SubBase("Производный", 'A')`. Этой командой создается объект подкласса, а ссылка на него записывается в объектную переменную суперкласса. Затем мы снова выполняем уже знакомые команды `obj.sayHello()` и `obj.show()`. Формально команды выглядят так же, как и в предыдущем случае, но теперь переменная `obj`

ссылается на объект подкласса. Выполнение команды `obj.sayHello()` к сюрпризам не приводит: появляется сообщение `"Всем привет!"`. В этом случае через объектную переменную `obj` вызывается метод `sayHello()`, унаследованный из суперкласса. Чудеса начинаются при выполнении команды `obj.show()`. В результате выполнения команды в консоль выводится сообщение `"Объект подкласса: Производный. Код: A"`. Это сообщение выводится методом `show()`, переопределенным в подклассе. Результат выполнения программы имеет вид:

```
Всем привет!
Объект суперкласса: Базовый
Всем привет!
Объект подкласса: Производный. Код: A
```

Точно так же через объектную переменную `obj` можно обратиться к полю `name`, но нельзя обратиться к полю `code`. Причина в том, что поле `name` описано в суперклассе и унаследовано в подклассе, а поле `code` описано непосредственно в подклассе, поэтому обратиться к ней через объектную переменную суперкласса нельзя.

На заметку:

Таким образом, объектная переменная суперкласса позволяет обращаться к полям и методам подкласса, которые унаследованы из суперкласса. В случае методов, если имеет место их переопределение, то вызывается версия метода из подкласса.

Абстрактные классы и интерфейсы

Класс может содержать *абстрактные методы*. Абстрактный метод, в отличие от обычного, имеет сигнатуру, но не содержит блока с программным кодом. Абстрактный метод объявляется с ключевым словом `abstract`. После сигнатуры абстрактного метода ставится точка с запятой. Класс, содержащий хотя бы один абстрактный метод, также называется абстрактным. Описывается такой класс с использованием все того же ключевого слова `abstract`.

На заметку:

Для абстрактного класса не может быть создан экземпляр класса, то есть объект. Причина очевидна и связана с тем, что абстрактный класс содержит абстрактные методы, которые невозможно выполнять, ведь они не содержат тела метода с программным кодом, а только сигнатуру.

Фактически абстрактные классы ценны своими абстрактными методами. Хотя может показаться, что необходимость в существовании абстрактных методов отсутствует, это далеко не так. Абстрактные классы создают для того, чтобы на их основе, путем наследования, создавать другие классы.

В этом отношении абстрактный класс с его абстрактными методами задает шаблон своих подклассов. Конкретная "начинка" такого шаблона определяется программным кодом подкласса. Пример объявления и использования абстрактного класса приведен в листинге 8.6.

Листинг 8.6. Абстрактный класс

```
// Абстрактный класс:
abstract class Base{
   // Текстовое поле:
   String operation;
   // Конструктор:
   Base(String str){
      operation=str;}
   // Абстрактный метод:
   abstract int F(int n);
   // Обычный метод (с вызовом абстрактного):
   void show(int n){
      System.out.println("Операция: "+operation);
      System.out.println("Аргумент: "+n);
      System.out.println("Значение: "+F(n));}
   }
// Подкласс абстрактного суперкласса:
class BaseA extends Base{
   // Конструктор подкласса:
   BaseA(){
      // Вызов конструктора (абстрактного) суперкласса:
      super("факториал");}
      // Переопределение абстрактного метода:
   int F(int n){
      if(n==1) return 1;
      else return n*F(n-1);}
   }
// Еще один подкласс абстрактного суперкласса:
class BaseB extends Base{
   // Вызов конструктора (абстрактного) суперкласса:
   BaseB(){
      // Вызов конструктора (абстрактного) суперкласса:
      super("двойной факториал");}
   // Переопределение абстрактного метода:
   int F(int n){
      if(n==1||n==2) return n;
      else return n*F(n-2);}
   }
public class AbstractDemo {
   public static void main(String[] args){
      // Создание объекта первого подкласса:
      BaseA A=new BaseA();
      // Создание объекта второго подкласса:
```

```
    BaseB B=new BaseB();
    // Вызов метода из первого подкласса:
    A.show(5);
    // Вызов метода из второго подкласса:
    B.show(5);}
}
```

Здесь объявляется абстрактный класс `Base`. Обращаем внимание на наличие ключевого слова `abstract` в описании класса. В классе есть текстовое поле `operation` и конструктор с одним текстовым аргументом (значение аргумента присваивается в качестве значения полю `operation` класса). Это банальная ситуация. Относительно новой является команда `abstract int F(int n)` (заканчивается точкой с запятой!). Этой командой объявляется абстрактный метод. Чтобы можно было проигнорировать тело с программным кодом метода (его просто нет!), сигнатуру метода начинаем с ключевого слова `abstract`. Метод называется `F()`, возвращает значение типа `int`, и у него один аргумент также типа `int`. При создании подкласса на основе класса `Base` необходимо будет переопределить этот метод (на самом деле термин *определить* здесь будет уместнее). Но на этом программный код класса `Base` не заканчивается. У класса есть обычный, неабстрактный, метод `show()`. Метод не возвращает результат, и у него один целочисленный аргумент. Методом в консольное окно выводится три сообщения. Первое состоит из текста `"Операция: "` и текстового значения поля `operation`. Второе сообщение состоит из текста `"Аргумент: "` и значения аргумента, переданного методу `show()`. Третье сообщение состоит из текста `"Значение: "` и значения, возвращаемого методом `F()`, если ему передать в качестве аргумента то же значение, что указано аргументом метода `show()`.

На заметку:

Обратите внимание: в теле неабстрактного метода `show()` вызывается абстрактный метод `F()`, который в классе `Base` объявлен, но не описан.

На основе класса `Base` через наследование создается два разных подкласса. Они называются `BaseA` и `BaseB`. Рассмотрим каждый из этих классов.

В классе `BaseA` конструктор не имеет аргументов. В нем инструкцией `super("факториал")` вызывается конструктор суперкласса с текстовым аргументом `"факториал"`. Это значение получает поле `operation`. Также в классе переопределяется метод `F()`, для чего использована такая инструкция:

```
int F(int n){
   if(n==1) return 1;
   else return n*F(n-1);}
```

Это так называемая *рекурсия* - в коде описания метода вызывается этот же метод (но с другим аргументом). В данном случае метод `F()` запрограммиро-

ван так, что он вычисляет *факториал* (произведение натуральных чисел, например 5!=5·4·3·2·1) для значения аргумента. Методы вычисляется по следующей схеме: если аргумент равен единице, методом возвращается единичное значение. В противном случае вычисляется выражение, равное произведению аргумента на результат вычисления метода с аргументом, уменьшенным на единицу. Для реализации этой схемы использован условный оператор.

На заметку:

Допустим, необходимо вычислить выражение `F(5)`. При его вычислении загружается код метода `F()` и сначала проверяется, равен ли аргумент единице. В данном случае это не так. Поэтому как результат возвращается выражение `5*F(4)`. Чтобы вычислить это выражение, снова загружается код метода и рассчитывается значение `F(4)`. Для этого вычисляется выражение `4*F(3)`, и так далее, пока не будет вычислено выражение `2*F(1)`. Потом эта цепочка сворачивается в обратном направлении. Поэтому следует учесть, что, хотя рекурсия позволяет создавать эффектный программный код, он не всегда эффективный с точки зрения экономии использования системных ресурсов.

В подклассе `BaseB` описан конструктор без аргумента, в котором инструкцией `super("двойной факториал")` вызывается конструктор суперкласса. Также в классе переопределяется метод `F()`. Код метода такой:

```
int F(int n){
   if(n==1||n==2) return n;
   else return n*F(n-2);}
```

Здесь также использована рекурсия. Методом вычисляется *двойной факториал* числа - произведение чисел через одно (например, 5!!=5·3·1, а 6!!=6·4·2). Если значение аргумента метода равно 1 или 2, то возвращается соответственно 1 или 2. В противном случае возвращается выражение, равное произведению значения аргумента на результат вызова метода `F()` с аргументом, уменьшенным на два.

В методе `main()` в классе `AbstractDemo` командами `BaseA A=new BaseA()` и `BaseB B=new BaseB()` создаются объекты подклассов `BaseA` и `BaseB` соответственно. Затем командами `A.show(5)` и `B.show(5)` из каждого объекта вызывается метод `show()`. В этом методе, в свою очередь, вызывается метод `F()`. Но в каждом подклассе этот метод переопределен по-своему. Результат выполнения программы имеет такой вид:

```
Операция: факториал
Аргумент: 5
Значение: 120
Операция: двойной факториал
Аргумент: 5
Значение: 15
```

Продолжением идеи абстрактных классов является концепция *интерфейсов*, широко используемая в Java. В отличие от абстрактного класса, который кроме абстрактных методов, может еще содержать обычные методы и поля, в интерфейсе все методы абстрактные. Точнее, в интерфейсе методы только описываются, но не определяются (при этом в описании методов ключевое слово `abstract` указывать не нужно). Поля также могут присутствовать в интерфейсе, но им сразу нужно присваивать значения, и такие поля являются по умолчанию статическими и неизменными (то есть такими, как если бы они были описаны с идентификаторами `final` и `static`). Ну и разумеется, никаких конструкторов в интерфейсе! Им там просто нечего делать.

На заметку:

Обычно принято названия полей интерфейса вводить большими буквами. Такие поля играют роль констант.

Объявляется интерфейс так же, как и класс (с учетом указанных выше поправок на структуру интерфейса), только вместо ключевого слова `class` указывается ключевое слово `interface`, то есть все это, в общем и целом, выглядит так:

```
interface имя_интерфейса{
   // константы и объявления методов
   }
```

Несложно догадаться, что сами по себе интерфейсы, учитывая невозможность создания на их основе объектов, напрямую не используются. Но на основе интерфейсов можно создавать классы. В этом случае говорят о *реализации* интерфейса. Реализация интерфейса в классе подразумевает, то объявленные в интерфейсе методы описаны в этом классе. При описании метода из интерфейса необходимо в сигнатуре метода указать ключевое слово `public`.

На заметку:

Ключевое слово `public` означает, что соответствующий член класса открытый. Хотя в Java по умолчанию и так все члены классов открытые, это разная степень "открытости". Члены класса, указанные без ключевого слова `public`, доступны в пределах пакета, а описанные с ключевым словом `public` доступны и вне пакета. Что такое пакет, описывается в этой главе несколько позже.

Если в классе описать не все методы интерфейса, то этот класс будет абстрактным и его следует объявлять с ключевым словом `abstract`. То есть ситуация фактически такая, как если класс, при реализации интерфейса, наследует все его члены (в первую очередь объявленные в интерфейсе методы). Поскольку по определению методы в интерфейсе только описаны, класс наследует абстрактные методы. Если хотя бы один из этих абстракт-

ных методов в классе не описан, класс будет содержать абстрактный метод. А это означает, что класс абстрактный.

Для реализации интерфейса в классе при описании класса в заголовке, после имени класса необходимо указать ключевое слово `implements` и имя интерфейса. В наиболее простом случае синтаксис класса, реализующего интерфейс, соответствует такому шаблону:

```
class имя_класса implements имя_интерфейса{
// код класса с реализацией методов интерфейса
}
```

В листинге 8.7 приведен пример программы, в которой используется интерфейс.

Листинг 8.7. Использование интерфейса

```
// Интерфейс:
interface Mathematica{
   // Поле-константа:
   int TEN=10;
   // Объявление методов:
   int factorial(int n);
   double power(double x,int n);
   }
// Класс реализует интерфейс:
class MyMath implements Mathematica{
   // Описание метода интерфейса (факториал):
   public int factorial(int n){
      int s=1,i;
      for(i=n;i>0;s*=i--);
      return s;}
   // Описание метода интерфейса (возведение в степень):
   public double power(double x,int n){
      double s=1;
      for(int i=1;i<=n;s*=x,i++);
      return s;}
   // Метод для отображения результатов вызова методов:
   void show(double x,int n){
      System.out.println("Первый аргумент: "+x);
      System.out.println("Второй аргумент: "+n);
      System.out.println("Факториал: "+factorial(n));
      System.out.println("Возведение в степень: "+power(x,TEN));}
   }
public class UsingInterfaceDemo {
   public static void main(String[] args){
      // Создание объекта класса:
```

```
    MyMath obj=new MyMath();
    // Переменные для передачи аргументами.
    // Определяются через статическое поле-константу:
    int n=Mathematica.TEN/3;
    double x=(double)MyMath.TEN/n-n;
    // Вызов метода show():
    obj.show(x,n);}
}
```

В интерфейсе `Mathematica` объявлена целочисленная константа `TEN` со значением `10` и два метода. Метод `factorial()` имеет целочисленный аргумент и возвращает результатом значение целого типа. Метод `power()` принимает два аргумента - первый типа `double` и второй типа `int`. Возвращается методом значение типа `double`. Это те формальные позиции, которым должны соответствовать описываемые при реализации интерфейса методы.

Интерфейс `Mathematica` реализуется в классе `MyMath`. В частности, в классе описываются методы `factorial()` и `power()` так, что методом `factorial()` вычисляется факториал для целого числа, переданного аргументом методу, а методом `power()` первый аргумент возводится в целочисленную степень, переданную через второй аргумент.

На заметку:

Обратите внимание, что методы, унаследованные в классе из интерфейса, описываются с ключевым словом `public`. Это обязательное условие.

Методом `show()` в консольное окно выводится несколько сообщений, содержащих информацию о том, какие аргументы передаются методу и каков результат вызова методов `factorial()` и `power()`.

В главном методе программы командой `MyMath obj=new MyMath()` создается объект класса `MyMath`. Командами `int n=Mathematica.TEN/3` и `double x=(double)MyMath.TEN/n-n` объявляются и инициализируются на основе значения статической константы `TEN`, которые затем в команде `obj.show(x,n)` передаются аргументами методу `show()`.

На заметку:

Статическое постоянное поле `TEN` объявлено в интерфейсе `Mathematica`. Обратиться к этому полю можно через имя интерфейса, то есть в формате `Mathematica.TEN`. Интерфейс `Mathematica` реализуется в классе `MyMath`. Это как если бы поле было описано в этом классе. Поэтому внутри класса к полю `TEN` можно обратиться по имени. Вне класса, кроме инструкции `Mathematica.TEN`, к полю можно обратиться через инструкцию `MyMath.TEN`, то есть как к статическому полю класса.

В результате выполнения программы в консоли появляются следующие сообщения:

```
Первый аргумент: 0.3333333333333335
Второй аргумент: 3
Факториал: 6
Возведение в степень: 1.6935087808430364E-5
```

На первый взгляд может показаться, что интерфейсы играют точно ту же роль, что и абстрактные классы. У интерфейсов есть несколько важных особенностей, которые дают им существенное преимущество по сравнению с абстрактными классами. Во-первых, как и в случае с обычными классами, один интерфейс может наследовать другой интерфейс. При этом используется все то же ключевое слово `extends`. Процедура наследования одного интерфейса другим называется *расширением* интерфейса. Синтаксис расширения интерфейса имеет такой вид:

```
interface интерфейс_1 extends интерфейс_2{
// программный код интерфейса
}
```

Например, блок кода с интерфейсами, наследующими друг друга, может выглядеть так:

```
interface InterfaceA{
   void F1();
   double F2(int n);
   }
interface InterfaceB extends InterfaceA{
   int F3(double x);
   }
```

В данном случае объявляется интерфейс `InterfaceA`, у которого два метода: `F1()` и `F2()`. На его основе путем наследования создается интерфейс `InterfaceB`. Непосредственно в теле этого интерфейса описан метод `F3()`. Вместе с тем, в интерфейсе `InterfaceB` из интерфейса `InterfaceA` наследуются методы `F1()` и `F2()`. На практике это означает, что класс, реализующий интерфейс `InterfaceB`, должен содержать (если он только не абстрактный) описание всех трех методов: `F1()`, `F2()` и `F3()`.

Как и в случае с классами, интерфейс может наследовать только один интерфейс. Зато класс может реализовать сразу несколько интерфейсов. Если класс реализует несколько интерфейсов, после ключевого слова `implements` реализуемые интерфейсы указываются через запятую. При этом класс, реализующий интерфейсы, может быть подклассом какого-то суперкласса. Другими словами, одновременно с расширением интерфейсов может применяться и механизм наследования классов. Синтаксис объявле-

ния класса, который наследует класс и реализует несколько интерфейсов, имеет следующий вид:

```
class класс_1 extends класс_2 implements интерфейс_1, интерфейс_2, ...{
   // код класса
   }
```

После ключевого слова `class` и имени класса следует ключевое слово `extends` и имя суперкласса, затем ключевое слово `implements` и через запятую перечисляются реализуемые интерфейсы. Следующий программный код иллюстрирует ситуацию, когда в подклассе реализуется несколько интерфейсов.

```
class Base{
   String text;
   void show(){
      System.out.println(text);}
   }
interface InterfaceA{
   String getText();
   void set(String text);}
interface InterfaceB{
   void showText();
   }
class SubBase extends Base implements InterfaceA, InterfaceB{
   public String getText(){
      return text;}
   public void set(String text){
      this.text=text;}
   public void showText(){
      System.out.println("Текстовое поле:");
      show();}
   }
```

В этом небольшом иллюстративном фрагменте программного кода объявляется класс `Base` с текстовым полем `text` и методом `show()`, который отображает в консольном окне значение текстового поля. Также объявлены два интерфейса. В интерфейсе `InterfaceA` объявляются методы `getText()` (без аргументов, возвращает значение типа `String`) и `set()` (принимает текстовый аргумент и не возвращает результат).

В интерфейсе `InterfaceB` объявляется метод `showText()` (без аргументов и не возвращает результат). Класс `SubBase` наследует класс `Base` и реализует интерфейсы `InterfaceA` и `InterfaceB`. Как следствие в классе `SubBase` незримо присутствует текстовое поле `text` и метод `show()`, а также описываются методы `getText()`, `set()` и `showText()`.

На заметку:

Если неабстрактный класс реализует несколько интерфейсов, то в нем должны быть описаны методы из каждого реализуемого интерфейса. Каждый метод описывается с ключевым словом `public`.

Теоретически может оказаться так, что два или более интерфейсов, реализуемых в классе, имеют одинаковые методы. В этом случае в классе, реализующем интерфейсы, описывается один метод. Это означает, что при "накладывании" методов из разных интерфейсов не возникает конфликта, связанного с тем, что один метод описан дважды. Здесь проявляется гибкость интерфейсов и кроется одна из причин, по которой в Java отказались от многократного наследования классов (когда один класс создается на основе сразу нескольких классов).

На заметку:

В отличие от Java, в C++ многократное наследование разрешено. С этим механизмом связан ряд проблем, причины которых, как показывает опыт, кроются не непосредственно в самом принципе наследования, а конкретной реализации методов в наследуемых классах. С другой стороны, многократное наследование - очень мощный и эффективный механизм. В Java эта дилемма разрешена с помощью концепции интерфейсов. Многократное наследование фактически реализовано через реализацию нескольких интерфейсов в классе. При этом, поскольку в интерфейсах методы не описаны, а только объявлены, проблема с несовместимостью разных описаний методов по большей части решена.

Такой подход позволяет реализовать через иерархию интерфейсов нужную структуру программы, а затем на основе этой иерархии уже можно создавать иерархию классов.

Еще одно важное свойство, связанное с интерфейсами, состоит в том, что можно создавать объектные переменные с типом интерфейса - так называемые интерфейсные переменные.

При объявлении интерфейсной переменной в качестве типа переменной указывается имя интерфейса. Для интерфейса объект создать нельзя. В качестве значения интерфейсной переменной может присваиваться ссылка на объект класса, который реализует соответствующий интерфейс. Правда, через такую ссылку можно получить доступ только к тем методам, которые объявлены в интерфейсе. Рассмотрим пример, представленный в листинге 8.8.

Листинг 8.8. Интерфейсная переменная

```
// Интерфейс:
interface Base{
   // Метод объявлен в интерфейсе:
   void show();}
```

```
// Класс реализует интерфейс:
class A implements Base{
   // Описание метода интерфейса:
   public void show(){
      System.out.println("Это метод класса A!");}
   }
// Этот класс также реализует интерфейс:
class B implements Base{
   // Описание метода интерфейса:
   public void show(){
      System.out.println("Это метод класса B!");}
   }
public class InterfaceRefDemo {
   public static void main(String[] args){
      // Интерфейсная переменная:
      Base ref;
      // Ссылка на объект первого класса:
      ref=new A();
      // Вызов метода через интерфейсную переменную:
      ref.show();
      // Ссылка на объект второго класса:
      ref=new B();
      // Вызов того же метода через интерфейсную переменную:
      ref.show();}
   }
```

Интерфейс `Base` содержит объявление всего одного метода `show()`, который не принимает аргументов и не возвращает результат. Интерфейс реализуется в классах `A` и `B`. В классе `A` метод `show()` описан так, что в результате вызова метода в консольное окно выводится сообщение `"Это метод класса A!"`. В классе `B` метод `show()` определен для вывода сообщения `"Это метод класса B!"`.

В главном методе программы командой `Base ref` объявляется интерфейсная переменная `ref`. Затем этой переменной с помощью команды `ref=new A()` в качестве значения присваивается ссылка на объект класса `A`. Поэтому в результате вызова команды `ref.show()` в консоли появляется сообщение `"Это метод класса A!"`. В данном случае, поскольку интерфейсная переменная `ref` ссылается на объект класса `A`, то вызывается версия метода `show()`, описанная в классе `A`.

После этого меняем значение интерфейсной ссылки `ref`. В результате выполнения команды `ref=new B()` интерфейсная ссылка `ref` ссылается на объект класса `B`. Теперь при вызове метода `show()` через интерфейсную переменную `ref` (команда `ref.show()`) вызывается версия метода, описанная в классе `B`. В результате появляется сообщение `"Это метод класса B!"`.

Таким образом, в результате выполнения программы получаем два сообщения:

```
Это метод класса A!
Это метод класса B!
```

Следовательно, в зависимости от того, на объект какого класса ссылается индексная переменная, вызываются разные версии методов, хотя вызываются методы через одну и ту же интерфейсную переменную.

Пакеты и уровни доступа

Пакеты удобно представлять как своеобразные контейнеры, в которых хранятся классы, интерфейсы и другие пакеты (*подпакеты*). Здесь мы рассмотрим, как создаются пакеты, как они используются и каковы особенности структурирования программы с учетом наличия пакетов (подпакетов). В пределах пакета имена классов, интерфейсов и подпакетов уникальны. Таким образом, через пакеты формируются пространства имен.

Для того, чтобы создать пакет, необходимо в начале файла с программным кодом указать ключевое слово `package` и имя создаваемого пакета (в конце ставится точка с запятой). Если создается пакет подпакета, то имя подпакета указывается вместе с именем пакета (имя пакета, точка и имя подпакета). Например, чтобы создать пакет с названием `mypack`, первой командой в файле с программным кодом указываем такую инструкцию:

```
package mypack;
```

Для создания подпакета `subpack`, который размещается в пакете `mypack`, используем такую инструкцию:

```
package mypack.subpack;
```

Все классы и интерфейсы, описанные в файле с `package`-инструкцией, записываются в соответствующий пакет.

📖 На заметку:

Подпакет может содержать, в свою очередь, подпакеты, в которых также могут быть подпакеты. Имя глубоко спрятанных внутренних подпакетов указывается в `package`-инструкции указывается вместе со всеми внешними пакетами-контейнерами с использованием точки в качестве разделителя.

Файл может содержать только одну `package`-инструкцию, и она должна быть первой строкой в файле. Если такой инструкции нет вовсе, то описанные в классе классы и интерфейсы по умолчанию относятся к *безымянному пакету*.

Мало записать класс или интерфейс в пакет. Нужно уметь совместно использовать классы и интерфейсы из разных пакетов. Для использования

файлов и интерфейсов из внешнего пакета (отличного от текущего) используют инструкцию `import`, после которой указывается ссылка на класс или интерфейс с учетом полной иерархии пакетов и подпакетов. Если предполагается использовать все классы пакета, вместо имени класса указывается символ "звездочка" (то есть `*`). Например, чтобы получить доступ к классу `MyClass` из подпакета `subpack` пакета `mypack`, используем такую инструкцию:

```
import mypack.subpack.MyClass;
```

Чтобы получить доступ ко всем классам и интерфейсам подпакета `subpack` пакета `mypack`, используем следующую инструкцию:

```
import mypack.subpack.*;
```

При этом импортируются только те классы, которые описаны с ключевым словом `public`.

На заметку:

Инструкцию `import` можно и не указывать. В этом случае при использовании классов из внешних пакетов придется указывать так называемое *полное имя класса*: с указанием структуры пакетов и подпакетов. Библиотека стандартных классов принадлежит к пакету `java.lang` и доступна без использования `import`-инструкции.

Существует несколько правил, которые следует помнить и которых нужно придерживаться при работе с файлами программного кода, с учетом того, что описанные в файлах классы могут принадлежать к разным пакетам. Вот они:

1. В файле может быть только один *открытый* класс, то есть класс, описанный с ключевым словом `public`.
2. Если в файле есть открытый класс, то имя файла должно совпадать с именем этого класса.
3. Если в файле есть `package`-инструкция, то она указывается первой строкой кода.
4. В файле может быть несколько `import`-инструкций. Все они размещаются в начале файла, но после `package`-инструкции (если такая есть).

Еще один вопрос, на котором кратко остановимся в этой главе - это взаимная доступность членов класса с учетом того, что классы могут находиться в "родстве" (суперкласс и его подкласс), принадлежать к одному или разным пакетам, содержать в описании ключевое слово уровня доступа `public` (открытый член), `private` (закрытый член), `protected` (защищенный член) или не содержать такого ключевого слова (открытый по умолчанию член). В табл. 8.1 проиллюстрированы возможные ситуации.

Табл. 8.1. Доступность членов класса

	`public`	`private`	`protected`	**ничего**
Внутри класса	*доступен*	*доступен*	*доступен*	*доступен*
Подкласс в пакете	*доступен*	*недоступен*	*доступен*	*доступен*
Внешний класс в пакете	*доступен*	*недоступен*	*доступен*	*доступен*
Подкласс вне пакета	*доступен*	*недоступен*	*доступен*	*недоступен*
Внешний класс вне пакета	*доступен*	*недоступен*	*недоступен*	*недоступен*

Чтобы запомнить, где какой член доступен, можно воспользоваться несколькими простыми правилами.

1. Члены класса, описанные с ключевым словом `public`, доступны везде, в том числе и во внешних классах других пакетов.
2. Члены, описанные без ключевого слова уровня доступа, доступны везде в пределах пакета, но не доступны за его пределами, вне зависимости от того, о внешнем классе или подклассе идет речь.
3. Член класса, описанный с ключевым словом `private`, доступен только в классе, где он описан.
4. Член класса, описанный с ключевым словом `protected`, доступен везде, кроме случая, когда речь идет о внешнем классе из внешнего пакета.

Хотя такая схема уровней доступа может показаться на первый взгляд несколько запутанной, на практике она обеспечивает высокую гибкость программного кода.

Резюме

1. Наследование — это механизм, который позволяет одним классам получать свойства других классов (наследовать их).
2. Класс, который создается на основе уже существующего класса, называется подклассом. Исходный класс называется суперклассом.
3. При описании подкласса после его названия через ключевое слово `extends` указывается имя суперкласса. У подкласса может быть только один суперкласс.
4. Подкласс может быть суперклассом для другого класса.
5. При описании конструктора подкласса необходимо предусмотреть механизм передачи аргументов конструктору суперкласса.
6. Наследуемые из суперкласса методы можно переопределять в подклассе.

7. Объектная переменная суперкласса может ссылаться на объект подкласса. При этом существует доступ только к тем членам подкласса, которые объявлены в суперклассе.
8. Класс может содержать только объявление метода (без его описания). Такой класс (и метод) описывается с ключевым словом `abstract` и называется абстрактным. У абстрактного класса нет объектов.
9. Интерфейс напоминает абстрактный класс. Интерфейс содержит только объявления методов и статические константы. Описывается с использованием ключевого слова `interface`.
10. На основе интерфейсов создаются классы. Класс, который реализует интерфейс, содержит в своей сигнатуре ключевое слово `implements`, после которого указывается имя интерфейса.
11. Один класс может реализовывать сразу несколько интерфейсов.
12. Описываемый в классе метод интерфейса должен содержать `public`-инструкцию.
13. Один интерфейс может наследовать (расширять) другой интерфейс.
14. Интерфейсная переменная — переменная, в качестве типа которой указано имя интерфейса. Такая переменная может ссылаться на объект класса, реализующего интерфейс. Доступ возможен только к методам класса, объявленным в интерфейсе.
15. В Java классы и интерфейсы группируются по пакетам. Для группировки классов по пакетам используют `package`-инструкцию. Для подключения пакетов используют `import`-инструкцию.

Глава 9
Работа с текстом и другие утилиты

Минуточку! Будьте добры, помедленнее - я записываю...
(Из к/ф "Кавказская пленница")

В этой главе речь пойдет о ряде полезных встроенных классов Java. В первую очередь остановимся на способах и особенностях реализации текста с помощью объектов классов `String` и `StringBuffer`. Также здесь можно найти краткое описание возможностей, предоставляемых пользователю через класс `Math` для выполнения математических расчетов.

Работа с текстом

Для работы с текстом в Java предусмотрены несколько классов. В частности, интерес представляет класс `String`. Это один из основных классов, через которые реализуются текстовые значения. Особенность объектов этого класса состоит в том, что значения (текстовые) объектов класса `String` неизменны. Другими словами, если в объект класса `String` записан текст, в последующем текст, хранящийся в том объекте, изменить не получится. На первый взгляд может показаться, что это делает нецелесообразным использование объектов данного класса. Тем не менее, это не так. Чтобы понять, почему использование даже таких объектов представляется эффективным, следует вспомнить, что объект и объектная переменная - это далеко не одно и то же. При работе с текстом мы имеем дело с объектной переменной класса `String`, которая ссылается на объект с текстом. Если необходимо внести изменения в существующий текст, то просто на основе уже существующего текстового объекта создается новый, с "правильным" текстом, а ссылка в объектной переменной перебрасывается на новый объект. В результате — полная иллюзия того, что объекты класса `String` можно редактировать (изменять). Этой чудесной иллюзией мы неоднократно пользовались, когда эксплуатировали по полной программе текстовые поля и локальные текстовые переменные (переменные класса `String`).

У класса `String` есть несколько конструкторов, которые позволяют создавать объекты на основе уже существующего текста (объекта класса `String` или текстового литерала), символьного массива (массива, состоящего из символов — значений типа `char`), числового массива (массив значений типа `byte`, которые определяют коды символов, из которых формируется текст) или использовать конструктор без аргументов (в этом случае создается объект с пустой текстовой строкой). Такое разнообразие конструкто-

ров весьма полезно, поскольку сложно бывает наперед предугадать, когда, как и основываясь на чем придется создавать текст. В листинге 9.1 приведены примеры создания текста разными способами.

Листинг 9.1. Создание текста

```
public class MakeString{
   public static void main(String[] args){
      // Текст на основе литерала:
      String str1=new String("Текстовый литерал");
      System.out.println(str1);
      // Текст на основе объекта:
      String str2=new String(str1);
      System.out.println(str2);
      // Пустая строка:
      String str3=new String();
      System.out.println(str3);
      // Текст на основе символьного массива:
      char[] symbs=new char[]{'М','а','с','с','и','в'};
      String str4=new String(symbs);
      System.out.println(str4);
      // Текст на основе числового массива:
      byte[] nums=new byte[]{78,97,109,98,101,114,115};
      String str5=new String(nums);
      System.out.println(str5);
   }}
```

При создании объектов класса `String` на основе массивов (символьного или числового) можно указывать второй и третий числовые аргументы. В этом случае для создания текстового объекта используется не весь массив, а только его часть (подмассив). Второй аргумент определяет индекс первого элемента подмассива, а третий аргумент - количество элементов в подмассиве. Например, если для создания текста использовать команды `char[] symbs=new char[]{'М','а','с','с','и','в'}` и `String str=new String(symbs,1,3)`, то в объект `str` будет записан текст `"acc"`.

На заметку:

Здесь нелишне напомнить, что индексация элементов массива начинается с нуля.

Для работы с объектами класса `String` есть ряд специальных встроенных методов. Их достаточно много, поэтому остановимся на основных - тех, которые представляют наибольший интерес. Эти интересные методы представлены в табл. 9.1.

Табл. 9.1. Методы для работы с текстом

Метод	Описание
`charAt()`	Методом возвращается символ (значение типа `char`) из строки. Индекс символа указывается аргументом метода
`compareTo()`	Метод для сравнения строк. Аргументом методу передается объект класса `String`. Он сравнивается с объектом, из которого вызывается метод. Результат метода (целое число) вычисляется по следующим правилам. Выполняется посимвольное сравнение строк до первого отличного символа (или окончания одной из строк). Если встретились несовпадающие символы, то в качестве значения возвращается разность кодов несовпадающих символов (код символа строки объекта вызова минус код символа строки-аргумента метода). Если при сравнении строк дело закончилось окончанием одной из строк, то в качестве результата метода возвращается разность длин строк (длина строки объекта вызова минус длина строки объекта-аргумента метода). Если строки одинаковые, методом в качестве значения возвращается нулевое значение
`compareToIgnoreCase()`	Методом производится сравнение строк так же, как и в случае метода `compareTo()`, но только без учета состояния регистра
`concat()`	Метод используется для объединения строк. К строке из объекта вызова добавляется строка, указанная в качестве аргумента. В результате возвращается *новый* текстовый объект, который получается объединением указанных строк
`endsWith()`	Если методу в качестве аргумента передана подстрока, то результатом является логическое значение `true`, если строка объекта вызова заканчивается данной подстрокой. В противном случае возвращается значение `false`
`equals()`	Методом сравниваются строки: та, что записана в объект вызова, и та, что передана аргументом методу. Возвращается значение `true`, если оба текста совпадают (с учетом регистра символов). Иначе возвращается значение `false`
`equalsIgnoreCase()`	Метод аналогичен методу `equals()`, но сравнение выполняется без учета состояния регистра символов

Метод	Описание
`getBytes()`	Действие метода аналогично действию метода `getChars()`, но только символы преобразуются в числовые значения типа `byte`. То есть метод предназначен для преобразования подстроки из текста объекта вызова в `byte`-массив. Аргументы метода: начальный индекс подстроки, конечный индекс подстроки плюс один, `byte`-массив для записи результата преобразования и индекс позиции, начиная с которой заполняется массив
`getChars()`	Метод не возвращает результат и вызывается из текстового объекта. Метод предназначен для преобразования подстроки из текста объекта вызова в символьный массив. Аргументами метода указываются: начальный индекс подстроки, конечный индекс подстроки плюс один, символьный массив для записи результата преобразования и индекс позиции, начиная с которой заполняется массив
`indexOf()`	Если методу передается символьный аргумент (или текстовая подстрока), то в качестве значения возвращается индекс первого вхождения символа (подстроки) в строку. Если такого символа (подстроки) в строке нет, возвращается значение `-1`. Можно указать второй целочисленный аргумент. Он определяет индекс начала поиска
`lastIndexOf()`	Методу передается символьный аргумент (или текстовая подстрока). В качестве значения возвращается индекс последнего вхождения символа (подстроки) в строку. Если такого символа (подстроки) в строке нет, возвращается значение `-1`. Второй необязательный целочисленный аргумент определяет индекс начала поиска
`length()`	Методом в качестве значения возвращается длина текстовой строки (в символах), записанная в объекте, из которого вызывается метод
`replace()`	Методу передаются два символьных аргумента. В качестве результата методом возвращается строка, которая получается из строки объекта вызова заменой первого символа-аргумента на второй символ-аргумент
`startsWith()`	Если методу в качестве аргумента передана подстрока, то результатом является логическое значение `true`, если строка объекта вызова начинается данной подстрокой. В противном случае возвращается значение `false`

Метод	Описание
substring()	Методом возвращается текстовая подстрока (объект класса String) текста, записанного в объект вызова. У метода два целочисленных аргумента - например, a и b. Возвращается подстрока из символов с индексами от a до b-1. Если второй аргумент не указан, возвращается подстрока до конца текста
toCharArray()	Метод для преобразования текстовой строки из объекта вызова в символьный массив, который возвращается в качестве результата метода
toLowerCase()	Методом все символы текстовой строки из объекта вызова переводятся в верхний регистр
toString()	Один из основных методов. Переопределение этого метода в классе задает способ приведения объектов соответствующего класса к текстовому значению
toUpperCase()	Методом все символы текстовой строки из объекта вызова переводятся в верхний регистр
trim()	Методом в качестве значения возвращается текстовая строка, которая получается удалением из строки объекта вызова всех ведущих и завершающих пробелов

Хочется выделить несколько моментов, связанных с использованием методов для работы с текстом. Во-первых, объединение текстовых строк может выполняться как методом concat(), так и с помощью оператора сложения + (или сокращенной формы оператора присваивания +=). Такой способ объединения строк мы уже использовали. При этом нужно иметь в виду, что каждый раз, когда изменяется текстовая строка (объект класса String), на самом деле создается новый String-объект. Например, рассмотрим фрагмент программного кода:

```
String str;
str=new String("Текст");
str=str+" и текст";
```

В результате переменная str ссылается на объект с текстовым значением "Текст и текст", хотя происходит это в несколько этапов. Сначала командой String str объявляется объектная переменная класса String. Затем командой str=new String("Текст") создается объект класса String со значением "Текст". Ссылка на этот объект присваивается переменной str в качестве значения. Команда str=str+" и текст" выполняется так. Создается *новый* текстовый объект класса String, значение которого (тек-

стовая строка) получается объединением строки, записанной в объект, на который ссылается переменная `str`, и текстового литерала `" и текст"`. Ссылка на этот *новый* объект записывается в переменную `str`.

Также важно понимать разницу между сравнением строк, например, с помощью метода `equals()` и оператора ==. Кратко разница такая. Методом `equals()` сравнивается текстовое содержимое *разных* объектов. Значение `true` возвращается, если текст в объектах одинаков. При этом сами объекты физически различны. Оператором == значение `true` возвращается, если соответствующие объектные переменные ссылаются на один и тот же объект.

Текстовые литералы реализуются в Java в виде объектов класса `String`. Поэтому из текстового литерала, как и из обычного объекта, можно вызывать перечисленные выше методы. Например, в результате выполнения команды `"Текстовый литерал".length()` возвращается длина (в символах) текстового литерала, из которого вызывается метод (в данном случае в литерале 17 символов).

Для преобразования объектов к текстовым значениям используется метод `toString()`. Этот метод вызывается явно или неявно из преобразуемого объекта. Явно метод вызывается так же, как и другие методы — через точечный синтаксис, в формате `объект.toString()`. Аргументов у метода нет. Результатом является объект класса `String`. Неявно метод вызывается в тех случаях, когда по контексту выражения в том месте, где указан объект, должно быть выражение текстового типа — например, если объект указан аргументом метода `println()`. Метод объявлен в суперклассе `Object`, который находится в вершине иерархии классов в Java и переопределен для всех встроенных классов. В классах, создаваемых пользователем (программистом), этот метод можно переопределять. Не забывайте при этом, что в сигнатуре наследуемых методов при переопределении указывается ключевое слово `public`. В листинге 9.2 приведен пример переопределения метода `toString()`.

Листинг 9.2. Переопределение метода `toString()`

```
class MyClass{
   // Поле объекта - символьный массив:
   char[] code;
   // Поле объекта - целочисленный массив:
   int count;
   // Поле объекта - текст:
   String name;
   // Конструктор класса:
   MyClass(char[] code,int count,String name){
      int i;
```

```
      // Создание массива:
      this.code=new char[code.length];
      // Заполнение массива:
      for(i=0;i<code.length;i++){
         this.code[i]=code[i];}
      // Значение числового поля:
      this.count=count;
      // Создание текстового объекта:
      this.name=new String(name);}
// Переопределение метода toString():
public String toString(){
   // Локальная текстовая переменная:
   String text="Значения полей объекта.\n";
   // Формирование значения локальной текстовой переменной:
   text+="Поле name: "+name+"\n";
   text+="Поле count: "+count+"\n";
   text+="Поле code: |";
   for(int i=0;i<code.length;i++){
      text+=" "+code[i]+" |";}
   // Результат метода:
   return text;}
   }
public class toStringDemo{
   public static void main(String[] args){
      // Создание объекта:
   MyClass obj=new MyClass(new char[]{'Z','A','R','Q','W'},100,"НОВЫЙ");
      // "Отображение" объекта:
      System.out.println(obj);}
   }
```

В программе описывается класс `MyClass`, в котором переопределяется метод `toString()`. Кроме переопределения метода, в классе есть полесимвольный массив `code`, целочисленное поле `count`, текстовое поле `name`. В конструкторе класса этим полям присваиваются значения. У конструктора три аргумента: символьный массив, целое число и текстовая переменная.

На заметку:

Существует по меньшей мере два способа на основе символьного массива-аргумента конструктора присвоить значение полю-символьному массиву. Первый подразумевает присваивание переменной-полю значения переменной-аргумента конструктора. В этом случае и поле, и аргумент конструктора ссылаются на один и тот же объект. Второй подразумевает создание копии массива. Именно такой подход использован выше. В конструкторе создается символьный массив того же размера, что и массив, переданный аргументом конструктору. Далее выполняется поэлементное копирование массивов.

Аналогично обстоят дела с текстовым аргументом. Можно просто "перебросить" на текстовое поле ссылку, записанную в текстовый аргумент конструктора. Как и в случае с массивами, такой финт приводит к тому, что текстовое поле ссылается на тот же объект класса `String`, что не очень удобно. Чтобы создать новый текстовый объект, но с таким же текстовым значением, что и аргумент конструктора, используем конструктор класса `String`, указав аргументом текстовую строку, переданную аргументом конструктору.

Метод `toString()` переопределяется с сигнатурой `public String toString()`. В методе формируется текстовое значение (локальная переменная `text`), и это текстовое значение возвращается в качестве результата метода.

В главном методе программы объект класса `MyClass` создается с помощью команды `MyClass obj=new MyClass(new char[]{'Z','A','R','Q','W'},100,"НОВЫЙ")`.

На заметку:

Для передачи первого аргумента конструктору класса `MyClass` использована инструкция `new char[]{'Z','A','R','Q','W'}`. Это инструкция создания и инициализации символьного массива. Ссылка не записывается в переменную массива, поэтому в данном случае речь идет об анонимном массиве.

Затем выполняется команда `System.out.println(obj)`. В результате выполнения программы в консольном окне появляется такой текст:

```
Значения полей объекта.
Поле name: НОВЫЙ
Поле count: 100
Поле code: | Z | A | R | Q | W |
```

Это результат выполнения команды `System.out.println(obj)` в главном методе программы. В этой команде аргументом метода `println()` указан объект `obj` класса `MyClass`, в котором переопределен метод `toString()`. Именно благодаря переопределению метода `toString()` в классе `MyClass` становится возможным использование команды подобного вида.

Помимо класса `String`, для работы с текстом могут использоваться, например, классы `StringBuffer` и `StringTokenizer`. Поскольку много эти классы мы использовать не будем (точнее, не будем совсем), то ограничимся лишь краткой их характеристикой.

Класс `StringBuffer` принципиально отличается от класса `String` тем, что объекты, созданные на основе класса `StringBuffer`, могут изменяться. Другими словами, если текст реализован с помощью объекта класса `StringBuffer`, то редакторские правки в текст могут вноситься прямо в этом объекте. Достигается это за счет того, что при создании объектов класса `StringBuffer` выделяется дополнительное место, чтобы в значе-

ние объекта можно было вносить изменения. По умолчанию при создании объекта класса StringBuffer, кроме фактически текста, выделяется место еще для 16-ти символов. Именно за счет этого дополнительного места редактирование текста выполняется прямо в объекте.

У класса StringBuffer есть три конструктора: без аргумента, с целочисленным аргументом и с текстовым аргументом. В первом и втором случаях создается пустая строка, реализованная в виде объекта класса StringBuffer. Различие в том, что если в первом случае (конструктор без аргумента) создается объект с выделением в объекте места для шестнадцати символов, то второй способ (с вызовом конструктора с числовым аргументом) подразумевает явное определение размера буфера для записи дополнительных символов. Если конструктор класса вызывается с текстовым аргументом, то этот аргумент определяет значение создаваемого объекта класса StringBuffer.

Помимо стандартных (в основном таких же, как для класса String) методов, в классе StringBuffer есть несколько "особенных". Особенность их связана с тем, что объекты класса StringBuffer могут редактироваться. В качестве примера можно привести метод capacity(). Методом возвращается в качестве значения размер текста в символах, который может быть записан в объект, из которого вызывается метод. Результат метода capacity() (полный размер объекта) отличается от результата метода length() (размер текста) на величину буфера объекта. Прочие полезные методы класса StringBuffer перечислены в табл. 9.2.

Табл. 9.2. Некоторые методы класса StringBuffer

Метод	Описание
append()	Методом добавляется текстовое представление аргумента в строку из объекта вызова
charAt()	Метод возвращает значение символа с указанным индексом (аргумент метода)
delete()	Методом удаляется подстрока из строки (объект вызова). Удаляемая подстрока определяется аргументами метода - целыми числами. Первый аргумент определяет индекс первого удаляемого из строки символа, а второй аргумент - индекс последнего удаляемого символа плюс единица
deleteCharAt()	Удаление из строки символа с указанным индексом (аргумент метода)
ensureCapacity()	Методом задается размер буфера (указывается аргументом метода) после создания объекта

Метод	Описание
`getChars()`	Метод используется для записи текстовой строки в символьный массив (как и в классе `String`)
`insert()`	Методом выполняется вставка подстроки (второй аргумент метода) в строку (объект вызова). Первым аргументом указывается индекс начала вставки подстроки
`replace()`	Методом выполняется вставка в строку подстроки с одновременным удалением фрагмента. У метода три аргумента: два индекса, которые определяют позицию удаляемого фрагмента (индекс первого удаляемого символа и индекс последнего удаляемого индекса плюс единица), и текст, который вставляется вместо него
`reverse()`	Методом изменяется порядок следования символов в тексте объекта вызова
`setCharAt()`	Методом задается новое значение символа в указанной позиции. Индекс символа и его новое значение указываются аргументами функции
`setLength()`	Методом задается длина буфера (указывается аргументом метода)
`substring()`	Методом возвращается подстрока строки (объект вызова). Аргументы - индекс первого символа подстроки и индекс последнего индекса подстроки плюс единица

Есть также полезный класс `StringTokenizer`, который находится в пакете `java.util`. В общем и целом класс предназначен для работы с текстом, предназначенным для синтаксического анализа (например, для разбивки текста на слова). У класса есть несколько конструкторов. Например, аргументом конструктору можно передать текстовую строку, текстовую строку и строку форматирования или текстовую строку, строку форматирования и логическое значение — опцию для режима обработки разделителей. Первый аргумент формирует собственно текстовое значение. Второй аргумент (если он есть) представляет собой последовательное перечисление разделителей (символ возврата каретки, символ перехода к новой строке, запятую, пробел и тому подобное). Эти символы при подсчете количества слов в основном тексте в расчет не принимаются (как слова не учитываются). Если же указать в качестве третьего аргумента значение `true`, то любые разделители будут включаться в число слов (по умолчанию значение считается равным `false`).

Среди полезных методов класса `StringTokenizer` имеет смысл обратить внимание на те, что перечислены в табл. 9.3.

Табл. 9.3. Некоторые методы класса `StringTokenizer`

Метод	Описание
`nextToken()`	В качестве результата методом возвращается следующее слово (лексема) в виде строки (объект класса `String`). Метод вызывается без аргументов или с текстовым аргументом, который задает строку разделителей
`countTokens()`	Методом в качестве значения возвращается количество слов в тексте
`hasMoreElements()`	Методом возвращается логическое значение, определяющее, есть ли в тексте еще слова
`hasMoreTokens()`	Методом возвращается значение `true`, если в тексте осталось хотя бы одно слово и `false`, в противном случае
`nextElement()`	Методом в качестве результата возвращается объект следующего слова

Пример использования класса `StringTokenizer` приведен в листинге 9.3.

Листинг 9.3. Работа с классом `StringTokenizer`

```
import java.util.*;
public class StringTokenizerDemo{
   public static void main(String[] args){
   // Базовый текст:
   String text="Очень простой текст: состоит из слов, и без цифр!";
   // Объект класса StringTokenizer:
   StringTokenizer str=new StringTokenizer(text);
   // Текстовая переменная:
   String s;
   while(str.hasMoreTokens()){
      // "Считывание" очередного слова:
      s=str.nextToken();
      //s=str.nextToken(" :,!");
      //s=str.nextToken(":,!");
      // Вывод слова на экран:
      System.out.println(s);}
   }
}
```

Программа достаточно простая. Состоит она из единственного класса `StringTokenizerDemo`, который, в свою очередь, состоит из одного (главного) метода. В методе `main()` объявляется и инициализируется тестовая (класс `String`) переменная `text`. Затем с помощью команды `StringTokenizer str=new StringTokenizer(text)` создается

объект str класса StringTokenizer на основе текста из переменной text. Кроме этого, объявляется текстовая (объект класса String) переменная s. Она используется в операторе цикла while(). В этом операторе проверяется условие str.hasMoreTokens(). Это выражение возвращает значение true, если в объекте str еще есть непрочитанные слова, и false в противном случае. В самом операторе цикла выполняется две команды. Командой s=str.nextToken() из объекта str считывается очередное слово и записывается в переменную s. Затем командой System.out.println(s) слово выводится в одну строку в консоль. В результате выполнения программы получаем такой результат:

```
Очень
простой
текст:
состоит
из
слов,
и
без
цифр!
```

При считывании слов из текста в качестве разделителя по умолчанию считается пробел. Знаки препинания относятся к словам и отображаются вместе со словами. Если вместо команды s=str.nextToken() использовать команду s=str.nextToken(" :,!"), получим такой результат:

```
Очень
простой
текст
состоит
из
слов
и
без
цифр
```

В данном случае аргументом метода nextToken() указывается текстовая строка с пробелом, двоеточием, запятой и восклицательным знаком. В результате при разбивке на слова эти символы рассматриваются как разделители слов. Если в текстовой строке в аргументе метода nextToken() не указать пробел (воспользоваться командой s=str.nextToken(":,!")), получим следующий результат:

```
Очень простой текст
 состоит из слов
 и без цифр
```

В данном случае разделителем слов считаются двоеточие, запятая и восклицательный знак (пробел - нет).

Работа с датой и временем

Для работы с датой и временем используют класс `Date` из пакета `java.util`. Объекты этого класса позволяют хранить и обрабатывать значения даты/времени. У класса `Date` есть конструктор без аргумента, а также конструктор с целочисленным (тип `long`) аргументом. Для создания объекта с текущей датой и временем используют конструктор без аргумента. Если использован конструктор с аргументом, то этот аргумент определяет дату и время, заносимые в объект. Используется следующее правило: аргумент конструктора класса равен количеству миллисекунд, прошедших с *полуночи 1 января 1970 года*. Некоторые методы класса `Date`, полезные в работе, приведены в табл. 9.4.

Табл. 9.4. Методы класса `Date`

Метод	Описание
`after()`	Методу в качестве аргумента передается объект класса `Date`. Результатом возвращается логическое значение, равное `true`, если дата в объекте вызова более поздняя, чем дата в объекте-аргументе, и `false` в противном случае
`befor()`	Методу в качестве аргумента передается объект класса `Date`. Результатом возвращается логическое значение, равное `true`, если дата в объекте вызова более ранняя, чем дата в объекте-аргументе, и `false` в противном случае
`clone()`	Метод не имеет аргументов и создает копию объекта вызова
`compareTo()`	Методом сравниваются объект вызова и объект класса `Date`, переданный аргументом методу. Если дата в объекте вызова больше даты в объекте-аргументе, возвращается значение `1`, если меньше - значение `-1`, а если даты равны, возвращается `0`
`equals()`	Метод возвращает логическое значение `true`, если объект вызова и объект-аргумент содержат одну и ту же дату, и `false` - в противном случае
`getTime()`	Методом возвращается количество миллисекунд, прошедших с полуночи 1 января 1970 года
`setTime()`	Метод используется для присваивания значения даты в объекте вызова. Аргументом методу передается количество миллисекунд, прошедших с полуночи 1 января 1970 года. Это значение задает устанавливаемую дату и время

Пример использования класса `Date` приведен в листинге 9.4.

Листинг 9.4. Работа с классом `Date`

```
import java.util.*;
public class DateDemo {
   public static void main(String[] args){
```

```
   Date now=new Date();
   Date old=new Date(1000000000000L);
   System.out.println("Текущая дата и время: "+now);
   System.out.println("Еще одна дата: "+old);
   }
}
```

В программе создается два объекта (`now` и `old`) класса `Date`. Первый создается с помощью конструктора без аргумента. Это объект `now`, и он содержит текущую (на момент запуска программы) дату и время. Объект `old` создается конструктором с аргументом. Аргументом конструктору передано число `1000000000000L`.

На заметку:

По умолчанию целочисленный литерал относится к типу `int`. Такого целого `int`-числа, как `1000000000000`, нет, потому что банально не хватит памяти в 32 бита. Чтобы было видно, что это `long`-число, используем суффикс `L`.

Поскольку в классе `Date` имеется переопределенный метод `toString()`, объекты этого класса в случае необходимости автоматически преобразуются к текстовому формату. В частности, в программе объекты `old` и `now` передаются аргументами методу `println()`. Результат выполнения программы может выглядеть так:

```
Текущая дата и время: Sun Oct 03 23:30:53 EEST 2010
Еще одна дата: Sun Sep 09 04:46:40 EEST 2001
```

Класс `Date` не единственный, предназначенный для работы с датой и временем. В Java есть абстрактный класс `Calendar`, содержащий описание ряда методов, полезных при обработке данных типа дата/время. Реализацией абстрактного класса `Calendar` является класс `GregorianCalendar`, предназначенный для обработки системы дат (и времени) в рамках григорианского календаря. Класс `TimeZone` и его подкласс `SimpleTimeZone` предназначен для работы с часовыми поясами. Задачи, связанные с интернационализацией, решаются через класс `Locale`.

Математические утилиты

Математические функции доступны через методы (статические) класса `Math`. Для большинства методов их названия совпадают с названиями соответствующих математических функций. Также в классе определены две статические константы `PI` (значение $\pi \approx 3.141592$) и `E` (значение $e \approx 2.718282$). Основные методы класса `Math` перечислены в табл. 9.5.

Табл. 9.5. Методы класса `Math`

Метод	Краткое описание
`abs()`	Модуль числа
`acos()`	Арккосинус
`asin()`	Арксинус
`atan()`	Арктангенс
`atan2()`	Угол на точку на плоскости. Координаты передаются аргументами методу
`cbrt()`	Корень кубический
`ceil()`	Округление до наименьшего целого числа, которое больше или равно аргументу метода
`cos()`	Косинус
`cosh()`	Косинус гиперболический
`exp()`	Экспонента
`floor()`	Округление до наибольшего целого, не превышающего аргумент метода
`IEEEremainder()`	Вычисляется остаток от деления аргументов метода
`log()`	Натуральный логарифм
`log10()`	Десятичный логарифм
`max()`	Максимальное из значений аргументов метода
`min()`	Минимальное из значений аргументов метода
`pow()`	Возведение в степень (показатель степени - второй аргумент метода)
`random()`	Генерирование случайного числа в диапазоне от 0 до 1
`rint()`	Округление до ближайшего `int`-значения
`round()`	Округление (с увеличением) до ближайшего целого `long`-значения
`signum()`	Знак числа - аргумента метода
`sin()`	Синус
`sinh()`	Синус гиперболический
`sqrt()`	Корень квадратный
`tan()`	Тангенс
`tanh()`	Тангенс гиперболический
`toDegrees()`	Преобразование радиан в градусы
`toRadians()`	Преобразование градусов в радианы

В прикладных задачах нередко приходится генерировать случайные числа. Для этого можно воспользоваться методом `random()` из класса `Math`, а можно воспользоваться возможностями, предоставляемыми классом `Random`. Объекты класса `Random` создаются с помощью конструктора без аргументов или с целочисленным аргументом. В первом случае генератор

случайных чисел инициализируется на основе системной даты и времени, а во втором случае для инициализации генератора используется целочисленный аргумент конструктора.

У класса есть несколько полезных методов, которые перечислены в табл. 9.6.

Табл. 9.6. Некоторые методы класса `Random`

Метод	Краткое описание
`nextBoolean()`	Генерирование случайного логического значения
`nextBytes()`	Методом случайными числами заполняется `byte`-массив, указанный аргументом метода
`nextDouble()`	Генерирование случайного числа типа `double` в диапазоне от 0 до 1
`nextFloat()`	Генерирование случайного числа типа `float` в диапазоне от 0 до 1
`nextGaussian()`	Возвращается случайное число для стандартного нормального распределения
`nextInt()`	Генерирование случайного числа типа `int`. Аргументом метода можно указать верхнюю границу для генерируемых чисел
`nextLong()`	Генерирование случайного числа типа `long`
`setSeed()`	Методом, через аргумент, задается новое значение для инициализации генератора случайных чисел

Разумеется, перечисленные классы даже в малой степени не охватывают всю мощь и красоту встроенных классов Java. Тем не менее, на эти классы стоит обратить внимание.

Резюме

1. В Java текст — это, как правило, объект класса `String` (или `StringBuffer`).
2. "Текстовые" классы имеют несколько конструкторов и большой арсенал методов для работы с текстом.
3. Переопределяя метод `toString()`, можно задавать закон преобразования объектов классов пользователя к текстовому формату.
4. В Java имеется набор встроенных классов для работы с датой и временем: `Date`, `Calendar`, `GregorianCalendar`, `TimeZone` и другие.
5. Полезным при математических вычислениях будет класс `Math`. В нем описаны методы для основных (элементарных) математических функций.

Глава 10
Обработка исключительных ситуаций

Да гранаты у них не той системы!
(Из к/ф "Белое солнце пустыни")

Хочешь - не хочешь, а в программе рано или поздно могут возникнуть ошибки. Имеются в виду логические ошибки, которые связаны с алгоритмом выполнения программы, а не синтаксисом программного кода. Разумеется, лучше отслеживать возможные ошибки еще на стадии составления программного кода. Однако бывают такие ситуации, когда предусмотреть возможность появления ошибки практически нереально. То есть нужно быть морально готовым к тому, что в процессе выполнения программы может случиться неприятность. Как программа будет реагировать на такие неприятности, можно определить на программном уровне. Если этого не сделать, то при возникновении ошибки работа программы будет автоматически завершена. Это плохо. Иногда просто катастрофично. Задача любого, маломальски приличного программиста состоит в том, чтобы таких ситуаций не допускать.

📖 На заметку:

Читатель, помни! Программа должна завершать работу, когда это нужно программисту, а не когда она этого хочет!

Тому, как обрабатывать на программном уровне возникающие в программе ошибки, посвящена данная глава.

Исключительные ситуации и их типы

Чтобы понять, как обрабатывать ошибки, необходимо составить хотя бы самое общее представление о том, что происходит, когда в программе возникает ошибка.

📖 На заметку:

Ошибки, которые возникают в процессе выполнения программы, называются ошибками времени выполнения. Существуют также ошибки времени компиляции, которые возникают при компиляции программы. Эти ошибки нас не интересуют.

При возникновении ошибки выполнение программы приостанавливается и в соответствии с типом ошибки, которая возникла, создается объект,

который описывает эту самую ошибку. Другими словами, объект содержит информацию о том, какая именно возникла ошибка. Такой объект обычно называют *исключением*. Далее исключение (объект, описывающий ошибку) передается для обработки тому методу, который вызывал эту ошибку. Другими словами, исключение *вбрасывается* в метод, в котором возникла ошибка. Метод может или обработать ошибку, или не обработать. Если метод обрабатывает исключение, то дальнейшее выполнение программы определяется конкретным способом обработки возникшей ошибочной ситуации. Если метод не обрабатывает исключение, то оно передается внешнему методу (то есть методу, в котором вызывался тот, который вызвал ошибку). Если и этот метод не обрабатывает исключение, оно передается дальше, по цепочке, пока не будет обработано. Если исключение так и не будет обработано, то, в конце концов, вызывается так называемый обработчик по умолчанию, одним из последствий вызова которого является досрочное завершение программы.

Даже неподготовленному пользователю понятно, что далеко не каждая ошибка может быть обработана программными методами. Поэтому обрабатывать мы будем не все, а только некоторые ошибки. С другой стороны, для работы с исключениями в Java предлагается система встроенных классов. В вершине иерархии находится класс `Throwable`. У этого класса имеется два подкласса `Exception` и `Error`. Подкласс `Error` соответствует как раз тем ошибкам, которые обрабатывать программными методами бесполезно - это фатальные ошибки, наподобие переполнения стека. А вот с ошибками подкласса `Exception` можем иметь дело. В частности, особый интерес представляет подкласс `RuntimeException` класса `Exception`. Класс `RuntimeException`, в свою очередь, имеет подклассы, которые соответствуют стандартным ошибкам, которые могут произойти при выполнении программы - например, деление на ноль или выход за пределы массива.

Ситуация достаточно простая. Каждой стандартной ошибке соответствует определенный подкласс класса `RuntimeException`. При возникновении ошибки автоматически создается объект соответствующего класса. Классы наиболее распространенных ошибок перечислены в табл. 10.1.

Табл. 10.1. Некоторые классы исключений

Класс исключения	Описание ошибки
`ArithmeticException`	Ошибка, связанная с выполнением арифметических операций (например, деление на ноль на множестве целых чисел)

Класс исключения	Описание ошибки
`ArrayIndexOutOfBoundsException`	Ошибка возникает в том случае, если индекс массива выходит за допустимые границы
`ArrayStoreException`	Ошибка возникает при присваивании элементу массива значения несовместимого типа
`ClassCastException`	Ошибка возникает при попытке выполнить приведение несовместимых типов
`IllegalArgumentException`	Ошибка возникает, если методу указан неправильный аргумент
`IllegalMonitorStateException`	Ошибка возникает при работе с *монитором* (относится к многопоточному программированию)
`IllegalStateException`	Ошибка возникает, если ресурс находится в некорректном состоянии
`IllegalThreadStateException`	Ошибка возникает, если предпринята попытка выполнить некорректную операцию на *потоке*
`IndexOutOfBoundsException`	Ошибка связана с тем, что индекс выходит за допустимый диапазон значений
`NegativeArraySizeException`	Ошибка возникает при попытке создать массив отрицательного размера
`NullPointerException`	Ошибка возникает при некорректном использовании ссылок (обычно когда объектная переменная содержит пустую ссылку)
`NumberFormatException`	Ошибка, которая возникает при преобразовании строки к числовому значению (когда в число преобразуется строка, содержащая некорректное текстовое представление числа)
`SecurityException`	Ошибка возникает при попытке нарушить режим защиты
`StringIndexOutOfBoundsException`	Ошибка индексирования при работе с текстовой строкой
`UnsupportedOperationException`	Ошибка возникает при попытке выполнить некорректную операцию

Это так называемые *неконтролируемые* исключения Java. Кроме неконтролируемых, существуют еще и *контролируемые* исключения, примеры которых можно найти в табл. 10.2.

Табл. 10.2. Контролируемые исключения Java

Класс исключения	Описание ошибки
ClassNotFoundException	Ошибка связана с тем, что невозможно найти нужный класс
CloneNotSupportedException	Некорректная попытка клонировать объект
IllegalAccessException	Ошибка доступа к ресурсу
InstantiationException	Ошибка, связанная с попыткой создать объект абстрактного класса или интерфейса
InterruptedException	Ошибка, связанная с прерыванием одного потока другим
NoSuchFieldException	Ошибка связана с тем, что отсутствует нужное поле
NoSuchMethodException	Ошибка связана с тем, что отсутствует нужный метод

Контролируемые исключения от неконтролируемых отличаются мало, но одна важная особенность все же есть. Состоит она в следующем. Если метод может выбрасывать (генерировать) контролируемое исключение, и обработка этого исключения в методе не предусмотрена, то в сигнатуре этого метода необходимо указать ключевое слово `throws` и имя класса контролируемого исключения.

Обработка исключений

Для обработки исключительных ситуаций используют оператор `try-catch`. Идея использования этого оператора достаточно проста. Блок программного кода, который, как мы подозреваем, может вызвать ошибку при выполнении, заключается в блок `try`: после ключевого слова `try` в фигурных скобках указывается этот блок кода. После `try`-блока следует `catch`-блок. Этот `catch`-блок - это тот блок, в котором собственно и обрабатывается исключение. Точнее, программный код в этом блока выполняется в случае, когда в `try`-блоке возникла ошибка. Причем не любая ошибка, а ошибка определенного типа. Для того, чтобы указать, какого типа ошибки обрабатываются в `catch`-блоке, после ключевого слова `catch` в круглых скобках указывается аргумент - объектная ссылка одного из классов исключений (см. табл. 10.1). Таким образом, шаблон использования оператора `try-catch` может иметь такой вид:

```
try{
   // код, который может вызвать исключение
   }catch(класс_исключения объект_исключения){
   // код для обработки исключения
   }
```

Данный блок программного кода выполняется следующим образом. Выполняется программный код в блоке try. Если там все нормально (ошибка не возникает), то блок catch игнорируется и после выполнения кода в try-блоке выполняется следующий оператор, после оператора try-catch. Если при выполнении кода в try-блоке возникла ошибка, дальнейшее выполнение try-блока прекращается и создается объект исключения и выполняется проверка на предмет того, совпадает ли тип исключения с классом ошибки в catch-блоке (проверяется имя класса в круглых скобках после ключевого слова catch). Если совпадение есть, то выполняется программный код в catch-блоке, а затем управление передается оператору, следующему после оператора try-catch. Если совпадения нет (то есть catch-блок не предназначен для обработки возникшего исключения), исключение передается "выше" - внешнему catch-блоку (если имеют место вложенные операторы try-catch) или внешнему методу. Рассмотрим пример (листинг 10.1).

Листинг 10.1. Обработка исключительной ситуации

```
// Пакет для использования класса Random:
import java.util.*;
public class TryCatchDemo{
   public static void main(String[] args){
      // Общее количество ошибок:
      int n=3;
      // Счетчик ошибок и счетчик циклов:
      int count=0,nums=0;
      // Случайные числа:
      int a,b;
      // Объект для генерирования случайных чисел:
      Random rnd=new Random();
      // Оператор цикла:
      while(count<n){
         // Изменение значения счетчика циклов:
         nums++;
      // Отслеживаемый на предмет ошибки код:
      try{
         // Целое случайное число от 0 до 9 включительно:
         a=rnd.nextInt(10);
         // Целое случайное число от 0 до 5 включительно:
         b=rnd.nextInt(6);
         // Вывод в консоль текстового сообщения:
      System.out.print(nums+") Результат деления нацело: "+a+"/"+b+" = ");
      // Попытка вычислить отношение.
      // Здесь возможна ошибка деления на ноль:
      System.out.println(a/b);
      }catch(ArithmeticException eObj){//Обработка арифметической ошибки
```

```
            // При обработке ошибки в консоль выводится сообщение:
            System.out.println("бесконечность");
            // Использование объекта ошибки (исключения):
            System.out.println("Внимание! Произошла ошибка: "+eObj);
            // Изменение значения счетчика ошибок:
            count++;}
        }
        // Текстовое сообщение перед завершением работы программы:
        System.out.println("Количество ошибок: "+n+". Работа программы
        завершена!");}
    }
```

Поскольку в программе предполагается генерировать случайные числа с помощью объекта класса `Random`, то командой `import java.util.*` подключаем пакет `java.util`. Все остальное происходит в методе `main()` класса `TryCatchDemo`. Идея очень проста: в программе генерируются случайные целые числа (попарно) и выполняется целочисленное деление с выведением результата в консольное окно. При этом возможна ошибка, связанная с делением на ноль.

На заметку:

Ошибка деления на ноль относится к классу арифметических ошибок, то есть к классу `ArithmeticException`. Имейте в виду, что исключение генерируется только при целочисленном делении. Если результат вычисляется в виде `double`-значения, результатом является значение `Infinity` и исключение не генерируется.

В начале метода объявляется несколько целочисленных переменных. Назначение их таково: переменная `n` определяет количество ошибок, при превышении которого завершается генерирование случайных чисел. Счетчики ошибок и счетчик циклов реализуются через переменные `count` и `nums` соответственно. Переменные `a` и `b` нужны для записи сгенерированных случайных чисел.

Командой `Random rnd=new Random()` создается объект `rnd` класса `Random`, через который и будут генерироваться случайные числа. Числа генерируются в операторе цикла `while`. В цикле проверяется условие `count<n`. Пока это условие выполняется, выполняется и оператор цикла. В операторе цикла командой `nums++` увеличивается на единицу счетчик циклов. Дальнейший код подозревается на возможное наличие ошибки, поэтому он заключается в блок `try`. В этом блоке целое случайное число в диапазоне значений от `0` до `9` включительно генерируется и присваивается переменной `a` с помощью команды `a=rnd.nextInt(10)`. Для генерирования случайного значения переменной `b` используем команду `b=rnd.nextInt(6)` (возможное значение от `0` до `5` включительно). Затем командой `System.out.print(nums+") Результат деления наце-`

ло: "+a+"/"+b+" = ") на консоль выводится текстовое сообщение со значениями сгенерированных чисел. Попытка вычислить отношение переменных выполняется в команде System.out.println(a/b). Попытка - потому что здесь возможна ошибка. Если переменная b получила нулевое значение (а вероятность этого события составляет величину 0.2), то возникнет ошибка деления на ноль. Так вот, если ошибка не возникла, то один цикл заканчивается и начинается второй. Если ошибка возникла, то в игру вступает код, указанный в блоке catch.

На заметку:

При возникновении ошибки код из блока catch используется не потому, что это блок catch, а потому, что параметром catch-блока указан объект класса ArithmeticException. Ошибка деления на ноль относится именно к классу ArithmeticException. Если бы параметром в catch-блоке был указан объект иного класса, данная ошибка в catch-блоке не обрабатывалась.

Первой командой в catch-блоке указана команда System.out.println("бесконечность"), которой дописывается слово "бесконечность" после выражения со значением случайных чисел (ноль в знаменателе). Именно при вычислении результата такого отношения возникла ошибка. Следующей командой System.out.println("Внимание! Произошла ошибка: "+eObj) также выводится текстовая строка. Ее особенность состоит в том, что в аргументе метода println() указан объект eObj ошибки. Здесь использовано то обстоятельство, что в классах ошибок переопределен метод toString(), благодаря чему, указав, например, объект ошибки аргументом метода println(), получаем стандартное текстовое описание ошибки. Этим мы и воспользовались в данном случае. Наконец, командой count++ на единицу увеличивается счетчик ошибок.

После завершения оператора цикла выводится сообщение с информацией о том, что работа программы завершается. Результат выполнения программы имеет следующий вид:

```
1) Результат деления нацело: 0/2 = 0
2) Результат деления нацело: 8/2 = 4
3) Результат деления нацело: 1/0 = бесконечность
Внимание! Произошла ошибка: java.lang.ArithmeticException: / by zero
4) Результат деления нацело: 9/4 = 2
5) Результат деления нацело: 3/1 = 3
6) Результат деления нацело: 4/5 = 0
7) Результат деления нацело: 3/3 = 1
8) Результат деления нацело: 9/5 = 1
9) Результат деления нацело: 0/1 = 0
10) Результат деления нацело: 7/5 = 1
11) Результат деления нацело: 9/1 = 9
```

```
12) Результат деления нацело: 2/3 = 0
13) Результат деления нацело: 5/0 = бесконечность
Внимание! Произошла ошибка: java.lang.ArithmeticException: / by zero
14) Результат деления нацело: 8/4 = 2
15) Результат деления нацело: 8/4 = 2
16) Результат деления нацело: 3/0 = бесконечность
Внимание! Произошла ошибка: java.lang.ArithmeticException: / by zero
Количество ошибок: 3. Работа программы завершена!
```

Имейте в виду, что от запуска к запуску результат может быть разным, поскольку числа мы генерируем случайные, и выйти может все что угодно. Но неизменно два обстоятельства: ошибки обрабатываются, а работа программы завершается после возникновения ровно трех ошибок.

В этом примере мы видели, что в `catch`-блоке обрабатываются именно те ошибки, которые соответствуют классу, указанному в параметре блока `catch`. К счастью, для обработки ошибок разных типов можно одновременно использовать сразу несколько `catch`-блоков. Рассмотрим пример в листинге 10.2.

Листинг 10.2. Использование нескольких `catch`-блоков

```
// Подключение пакета:
import java.util.*;
public class UsingCatchsDemo{
   public static void main(String[] args){
      // Объект для генерирования случайных чисел:
      Random rnd=new Random();
      // Массив чисел:
      int[] array=new int[]{0,1,2,3,4,5,0,8,10,12,15};
      // Числитель отношения и счетчик ошибок:
      int n=120,count=0;
      // Оператор цикла:
      while(count<5){
         // Отслеживаемый код:
      try{
      // Возможна ошибка (деление на ноль и неверный индекс):
   System.out.print(n/array[rnd.nextInt(array.length+2)-1]+" ");}
      // Обработка ошибки "деление на ноль":
      catch(ArithmeticException eObj){
System.out.println("\nОшибка: деление на ноль! Описание: "+eObj);
count++;}
// Обработка ошибки "неверный индекс":
catch(ArrayIndexOutOfBoundsException eObj){
System.out.println("\nОшибка: неверный индекс! Описание: "+eObj);
count++;}
}}
}
```

Эта программа очень похожа на предыдущую. Остановимся только на основных моментах. В программе объявляется и инициализируется числовой массив, среди значений элементов которого имеется несколько нулей. Программой в строку выводятся результаты деления фиксированного числа на случайно выбранный элемент массива. Элемент массива выбирается путем генерирования индекса элемента массива. При этом сгенерированный индекс может и не попадать в допустимый диапазон значений. Поэтому при выполнении этой части кода могут возникать ошибки двух типов: *деление на ноль*, если случайно выбран элемент с нулевым значением, и *выход за пределы массива*, если сгенерирован индекс со значением вне допустимого диапазона. В программе эти две возможные ошибки отслеживаются и обрабатываются. Обратимся к программному коду.

Программа состоит из одного класса с главным методом программы. В этом методе создается объект класса `Random` для генерирования случайных чисел. Также объявляется и инициализируется массив array с целочисленными значениями, среди которых два нуля. В операторе цикла последовательно вычисляется результат деления переменной `n` (значение `120`) на случайно выбранный элемент массива `array`. Индекс элемента генерируется так, что, кроме допустимых значений, может принимать значение `-1` и на единицу большее, чем самый последний индекс массива. Код с выбором элемента и вычислением отношения указан в аргументе метода `println()`, который, в свою очередь, помещен в блок `try`. После этого блока идут два блока `catch`. Первому `catch`-блоку параметром передается объект класса `ArithmeticException`, а второму `catch`-блоку параметром передается объект класса `ArrayIndexOutOfBoundsException`.

📖 На заметку:

Поскольку область доступности параметров `catch`-блоков ограничивается этими блоками, то названия для объектов исключений в разных `catch`-блоках могут совпадать. При этом означают они разные объекты.

В каждом из `catch`-блоков выводится сообщение соответствующего содержания и на единицу увеличивается значение счетчика `count`. Оператор цикла выполняется до тех пор, пока общее количество ошибок не станет равным пяти. После этого работа программы заканчивается. Результат выполнения программы может иметь следующий вид:

```
8
Ошибка: неверный индекс! Описание: java.lang.ArrayIndexOutOfBoundsException: -1
10 40 8 24 8 10
Ошибка: деление на ноль! Описание: java.lang.ArithmeticException: / by zero
15
Ошибка: деление на ноль! Описание: java.lang.ArithmeticException: / by zero
```

```
120
Ошибка: неверный индекс! Описание: java.lang.ArrayIndexOutOfBoundsException:
11
60 40 40 40 24 30
Ошибка: деление на ноль! Описание: java.lang.ArithmeticException: / by zero
```

Фактически в строку выводятся числа, пока не возникнет ошибка. В этом случае в новой строке появляется сообщение соответствующего содержания и дальше с новой строки продолжается вывод чисел, пока снова не возникнет ошибка, и так далее.

На заметку:

В операторе `try-catch` может использоваться еще и необязательный блок `finally`. Программный код, помещенный в этот блок, выполняется в обязательном порядке, вне зависимости от того, возникла ошибка или нет.

Также нужно иметь в виду общую схему обработки ошибок, если используется несколько `catch`-блоков. Если возникла ошибка, по очереди перебираются `catch`-блоки до первого совпадения. Неприятность может крыться вот в чем. Мы знаем, что переменная суперкласса может ссылаться на объект подкласса. Поэтому совпадением при переборе ошибок и типов параметров в `catch`-блоках считается и ситуация, когда класс исключения является подклассом класса объекта, указанного параметром `catch`-блока. Поэтому, например, чтобы отслеживать и обрабатывать ошибки всех типов, можно для `catch`-блока указать параметр класса `Exception`.

Создание пользовательских классов исключений

Хотя это может показаться странным, в Java есть возможность генерировать исключения вручную - то есть создать иллюзию (очень достоверную) того, что в программе возникла ошибка. Для генерирования исключительной ситуации в программном коде используют инструкцию `throw`, после которого указывается имя объекта исключительной ситуации. Что касается последнего (то есть объекта исключительной ситуации), то есть два стандартных способа заполучить такой объект. Первый состоит в том, чтобы воспользоваться стандартным конструктором стандартного класса ошибки. Второй базируется на "отлавливании" объекта исключительной ситуации, которая возникает в силу каких-то причин, запоминании этого объекта и вбрасывании его в нужном месте программы. Вбрасывание объекта исключения означает, фактически, создание исключительной ситуации соответствующего типа. Особенно хорошо такой подход работает при определении пользовательского класса исключения, то есть создания "собственной" ошибки.

На заметку:

Ошибки (исключения) пользовательских классов относятся к контролируемым, и их нужно либо обрабатывать, либо объявлять в сигнатуре соответствующего метода (через ключевое слово `throws`).

Некоторые из этих моментов обсуждаются дальше. Рассмотрим пример в листинге 10.3.

Листинг 10.3. Генерирование исключения

```
// Класс "пользовательской" ошибки:
class MyException extends Exception{
   // Счетчик ошибок:
   static int count=0;
   // Текст для описания ошибки:
   String name;
   // Конструктор класса:
   MyException(String name){
      count++;
      this.name=name;
      }
   // Переопределение метода toString():
   public String toString(){
      String text="Возникла ошибка!\n";
      text+="Описание: "+name+"\n";
      text+="Номер ошибки: "+count;
      return text;}
   }
// Просто класс:
class MyClass{
   // Числовое поле:
   private int number;
   // Метод для отображения значения поля:
   void show(){
      System.out.println("Значение поля: "+number);}
   // Конструктор класса:
   MyClass(int number){
      try{
         if(number>10){
            number=10;
            // Генерирование ошибки:
            throw new MyException("Слишком большое число!");}
         if(number<0){
            number=0;
            // Генерирование ошибки:
            throw new MyException("Отрицательное число!");}
```

```
      }catch(MyException obj){ // Обработка ошибки
         System.out.println(obj);}
      this.number=number;
      show();}
   // Метод не обрабатывает ошибку:
   void set(int number) throws MyException{
      // Генерирование ошибки:
   if(number>10||number<0) throw new MyException("Неверный аргумент!");
      this.number=number;
      System.out.println("Все в порядке!");
      show();}
   }
// Главный класс программы:
public class UsingThrowDemo{
   public static void main(String[] args){
      // Создание объектов класса MyClass (генерируется ошибка):
      MyClass objA=new MyClass(15);
      MyClass objB=new MyClass(-1);
      // Отслеживается ошибка:
      try{
         objA.set(100);
      }catch(MyException e){ // Обработка ошибки
         System.out.println(e);
         objA.show();}
      // Создание объекта ошибки:
      MyException objE=new MyException("Несуществующая ошибка!");
      try{
         // Генерирование ошибки:
         throw objE;
         }catch(MyException e){ // Обработка ошибки
         System.out.println(e);}
   }}
```

В программе описаны три класса. Класс `MyException` наследует класс `Exception`. В классе объявляется статическое целочисленное поле `count` с начальным нулевым значением. Текстовое поле `name` предназначено для записи текста, описывающего суть ошибки.

Конструктор класса принимает один текстовый аргумент, который определяет значение текстового поля `name`. Также в конструкторе на единицу увеличивается значение статического поля `count`. Кроме этого, в классе переопределяется метод `toString()` так, что результатом преобразования объекта класса `MyException` к текстовому формату является текстовое сообщение с полной информацией об объекте ошибки: сообщение о том, что возникла ошибка, описание ошибки (значение поля `name`) и номер ошибки (значение статического поля `count`).

Класс MyClass предназначен для иллюстрации того, как может использоваться исключение пользовательского типа. В классе объявляется закрытое целочисленное поле number. Для отображения значения поля предназначен метод show().

Конструктору класса передается один целочисленный аргумент. Этот аргумент присваивается в качестве значения полю number. Но не просто присваивается, а с возможными вариантами. Основная часть кода конструктора заключается в блок try. В этом try-блоке сначала проверяется условие number>10 (здесь number - это аргумент конструктора). Если условие выполнено, то полю number присваивается значение 10, а затем инструкцией throw new MyException("Слишком большое число!") генерируется исключительная ситуация.

На заметку:

Для генерирования ошибки используем инструкцию throw. Объект ошибки создается командой new MyException("Слишком большое число!"). Этой командой создается анонимный объект класса MyClass. Именно этот анонимный объект вбрасывается при генерировании ошибки.

Затем еще одним условным оператором проверяется условие number<0 (number — аргумент конструктора). Дело до проверки этого условия доходит, только если не была сгенерирована ошибка при выполнении предыдущего условного оператора. Если *не выполняется то* условие, но *выполняется это*, полю number присваивается нулевое значение, а затем инструкцией throw new MyException("Отрицательное число!") генерируется другая ошибка (но все того же класса MyException). Обработка обеих ошибок (они одного класса) выполняется в catch-блоке. В этом блоке вызывается метод println() с передачей объекта ошибки в качестве аргумента. В результате в дело вступает метод toString(), и все получается довольно симпатично.

А если ни одно из двух условий (number>10 и number<0) не выполняется, полю number присваивается значение аргумента конструктора, а затем методом show() значение этого поля выводится в консольное окно.

Также в классе описан метод set(), предназначенный для присваивания значения полю number. Этот метод может выбрасывать исключение класса MyException. Причем это исключение в методе не отслеживается и не обрабатывается. Поэтому в сигнатуре метода указано ключевое слово throws и имя класса ошибки, которая может выбрасываться методом (то есть throws MyException).

В самом методе, которому аргументом передается переменная number, если выполняется условие number>10||number<0, инструкцией

throw new MyException("Неверный аргумент!"). Другими словами, если аргумент метода больше десяти или меньше нуля, генерируется исключительная ситуация. Если исключительная ситуация по причине приличного поведения аргумента не генерируется, выполняются команды this.number=number (полю присваивается значение аргумента), System.out.println("Все в порядке!") (выводится текстовое сообщение) и show() (отображается значение поля number).

В классе UsingThrowDemo есть метод main(), в котором есть весьма интересные команды. Так, командами MyClass objA=new MyClass(15) и MyClass objB=new MyClass(-1) создаются два объекта класса MyClass, причем аргументы конструктору в обоих случаях передаются такие, что генерируется исключительная ситуация. Обрабатываются эти исключительные ситуации в теле конструктора.

Следующая часть кода помещена в блок try. В этом блоке вызывается команда objA.set(100), которая, как несложно догадаться, приводит к ошибке. Обработка ошибки происходит в блоке catch. В блоке catch командой System.out.println(e) выводится описание ошибки (здесь e - параметр блока catch). Затем вызывается команда objA.show(), благодаря чему узнаем значение поля number объекта objA.

Командой MyException objE=new MyException("Несуществующая ошибка!") создается объект objE ошибки класса MyException.

На заметку:

Создание объекта ошибки совсем *не означает* генерирования исключительной ситуации!

После этого в блоке try с помощью команды throw objE генерируем ошибочную ситуацию. Обрабатываем ее в блоке catch. При обработке исключения описание ошибки командой System.out.println(e) выводится в консольное окно (здесь e - параметр блока catch).

Результат выполнения программы имеет следующий вид:

```
Возникла ошибка!
Описание: Слишком большое число!
Номер ошибки: 1
Значение поля: 10
Возникла ошибка!
Описание: Отрицательное число!
Номер ошибки: 2
Значение поля: 0
Возникла ошибка!
Описание: Неверный аргумент!
```

```
Номер ошибки: 3
Значение поля: 10
Возникла ошибка!
Описание: Несуществующая ошибка!
Номер ошибки: 4
```

Всего в программе "происходит" четыре ошибки. Первые две возникают в результате создания объектов `objA` и `objB` и обрабатываются, как отмечалось, в конструкторе. Третья ошибка возникает при вызове метода `set()` из объекта `objA` в команде `objA.set(100)`. Эта ошибка обрабатывается вне метода `set()`. Наконец, описание четвертой ошибки появляется вследствие создания объекта `objE`.

Резюме

1. При возникновении ошибки в процессе выполнения программы автоматически создается объект, описывающий эту ошибку, — объект называется *исключение*, а сложившаяся ситуация — *исключительной*.
2. В Java существует иерархия классов, предназначенных для обработки исключительных ситуаций.
3. Если исключительную ситуацию не обработать программными методами, она приведет к "аварийному" завершению работы программы.
4. Обработка исключительных ситуаций выполняется в блоке `try-catch`. В `try`-блоке размещается контролируемый программный код, а в `catch`-блок помещается код, предназначенный для обработки исключительной ситуации.
5. Помимо использования встроенных классов, для обработки исключительных ситуаций пользователь/программист может описывать собственные классы исключений.

Глава 11
Многопоточное программирование

Таковы суровые законы жизни. Или, говоря короче, жизнь диктует нам свои суровые законы...
(Из к/ф "Золотой теленок")

В Java можно создавать программы, состоящие из разных частей программного кода, которые выполняются одновременно. Такие части программы, которые выполняются параллельно, называются потоками, а сам подход, базирующийся на использовании потоков, называется многопоточным программированием. В Java есть встроенная система классов для реализации многопоточной модели программирования.

📖 На заметку:

Существует некоторая неоднозначность, связанная с терминологией. То, что здесь мы называем *потоком*, в английской версии называется *thread*. Дословное значение этого термина означает *нить*. Иногда именно этот термин и используют. В ходу также термины *процесс*, *подпроцесс* и *легковесный процесс*. Понятно, что такое изобилие появилось не от хорошей жизни. Пожалуй, термин *поток* в данном случае наиболее точный. Дело в том, что *процесс*, в классическом понимании, подразумевает выделение операционной системой отдельной области оперативной памяти для каждого процесса. В данном случае память используется всеми потоками совместно. С другой стороны, в Java есть так называемые *потоки данных* (термин *stream*), которые обсуждаются в следующей главе. Следует учесть, что в этой и следующей главах описываются разные потоки.

Реализация потоков в Java

Через потоковую модель Java представлены наиболее общие свойства, связанные с многопоточным программированием. Вся (или почти вся) прелесть работы с потоками реализуется через класс `Thread`. У класса есть несколько статических методов для работы с потоками. Эти методы перечислены в табл. 11.1.

Табл. 11.1. Методы класса `Thread`

Метод	Описание
`activeCount()`	Метод в качестве результата возвращает количество активных потоков в текущей группе потоков
`currentThread()`	Методом в качестве результата возвращается ссылка на текущий объект потока (ссылка на текущий поток)

Метод	Описание
`getId()`	Методом в качестве результата возвращается идентификатор потока - уникальное целое число, генерируемое при создании потока
`getName()`	Метод в качестве результата возвращает имя потока, из которого вызывается этот метод
`getPriority()`	Методом в качестве результата возвращает приоритет потока
`getThreadGroup()`	Методом в качестве значения возвращается группа, к которой принадлежит текущий поток
`holdsLock()`	Метод в качестве значения возвращает логическое значение. Значение равно `true`, если метод удерживает *монитор* объекта, переданного в качестве аргумента методу. В противном случае возвращается значение `false`. Монитор - объект, который используется для блокировки ресурса потоком и недопущения одновременного обращения к ресурсу разных потоков. Концепция монитора используется для синхронизации работы потоков
`interrupt()`	Методом выполняется прерывание потока
`isAlive()`	Метод используется для того, чтобы определить, используется ли поток
`join()`	Метод-инструкция ожидания завершения потока
`run()`	Метод определяет точку входа в поток. Программный код потока задается путем переопределения этого метода
`setName()`	Методом задается имя потока (имя потока указывается в виде текстовой строки аргументом метода)
`setPriority()`	Методом задается приоритет потока (приоритет передается аргументом методу)
`sleep()`	Метод используется для приостановки выполнения потока. Время (в миллисекундах), на которое выполняется приостановка в работе потока, указывается аргументом метода
`start()`	Метод для запуска потока. При вызове этого метода автоматически запускается на выполнение метод `run()`, объявленный в интерфейсе `Runnable`
`wait()`	Метод переводит поток в режим ожидания
`yield()`	Метод временно приостанавливает текущий поток и позволяет выполнение других потоков

Здесь следует дать некоторые пояснения относительно выполнения и невыполнения потоков. Поскольку потоки в общем случае выполняются параллельно (одновременно), то, как говорится, возможны варианты. В том случае, если разные потоки в процессе выполнения вступают в конфликт, например, в части доступа к системным ресурсам, в расчет принимается

такой немаловажный показатель, как *приоритет потока*. Приоритет потока — это число, которое определяет "важность" этого потока. Само по себе значение приоритета не принципиально, важно только, у какого потока приоритет больше. Тот поток, у которого поток выше, в случае "конфликта интересов" имеет преимущество. Поток с более высоким приоритетом может прервать выполнение менее приоритетного потока.

Работу потока, кроме окончательного и бесповоротного завершения, можно приостановить. Выполнение потока приостанавливается на определенное время, после чего выполнение потока возобновляется. Отдельной темой является *синхронизация* потоков. Ведь в процессе выполнения потоки могут (и, как правило, так и происходит) обращаться к одним и тем же ресурсам. Проблемы, которые при этом возникают, намного серьезнее, чем может показаться на первый взгляд. Решаются эти проблемы как раз через процедуру синхронизации потоков.

Главный поток

При запуске программы автоматически создается главный поток - поток, который выполняет метод `main()`, то есть главный метод программы. В рамках главного потока могут создаваться (запускаться) дочерние потоки (подпотоки), в которых, в свою очередь, также могут запускаться потоки, и так далее.

На заметку:

Главный поток от прочих потоков отличается тем, что создается первым.

Поскольку до этого, кроме главного потока, мы с другими потоками дела не имели, то необходимости получать прямой доступ к потоку и настраивать его параметры у нас не было. Теперь у нас такая потребность неожиданно появилась. Начнем с простых вещей — с получения прямого доступа к главному потоку.

Поток с точки зрения объектной модели Java — это объект класса, наследующего класс `Thread` или реализующего интерфейс `Runnable`. Поскольку это объект, то на него можно и нужно сделать ссылку. Ссылка записывается в объектную переменную. Однозначно эта объектная переменная может относиться к классу `Thread`.

У класса `Thread` есть статический метод `currentThread()`, которым в качестве результата возвращается ссылка на поток, из которого вызывался метод. Поэтому если метод вызвать в главном методе программы (инструкция вида `Thread.currentThread()`), получим ссылку на главный поток. Этим чудесным обстоятельством воспользуемся в примере, представленном в листинге 11.1.

Листинг 11.1. Главный поток программы

```
public class MainThreadDemo{
// Главный метод может выбрасывать исключение InterruptedException:
public static void main(String[] args) throws InterruptedException{
// Объектная переменная для записи ссылки на главный поток:
Thread t;
// Время (в миллисекундах) задержки в выполнении потока:
long time=2500;
// Получение ссылки на объект главного потока:
t=Thread.currentThread();
// Вывод сведений о потоке в консольное окно:
System.out.println(t);
// Изменение имени потока:
t.setName("Это самый главный поток!");
// Считывание приоритета потока:
int p=t.getPriority();
// Изменение приоритета потока:
t.setPriority(++p);
// Вывод сведений о потоке в консольное окно:
System.out.println(t);
// Текстовое сообщение о приостановке выполнения потока:
System.out.println("Выполнение потока приостановлено на "+(double)
time/1000+" секунд.");
// Приостановка выполнения потока:
Thread.sleep(time);
// Завершение работы программы:
System.out.println("Работа программы завершена!");}
}
```

В результате выполнения программы получаем такую последовательность сообщений в консольном окне:

```
Thread[main,5,main]
Thread[Это самый главный поток!,6,main]
Выполнение потока приостановлено на 2.5 секунд.
Работа программы завершена!
```

Теперь рассмотрим и проанализируем программный код. Неприятности начинаются прямо с сигнатуры метода `main()`. При описании этого метода мы использовали инструкцию `throws InterruptedException`, которая означает, всего-навсего, что главный метод может выбрасывать необрабатываемое исключение класса `InterruptedException` (прерывание потока). Необходимость в такой инструкции в сигнатуре метода вызвана тем, что в теле метода вызывается статический метод `sleep()` класса `Thread`.

Командой `Thread t` в теле метода объявляется объектная переменная, в которую затем запишем ссылку на главный поток программы (точнее, на объект главного потока). Время задержки в выполнении главного потока (в миллисекундах) записывается в переменную `time`. Получение ссылки на объект главного потока выполняется командой `t=Thread.currentThread()`. После этого выводим информацию о главном потоке в консольное окно с помощью команды `System.out.println(t)`, в которой аргументом методу `println()` передается ссылка на поток. Это возможно благодаря переопределенному в классе `Thread` методу `toString()`.

На заметку:

В результате "вывода на экран" объекта потока появляется такое сообщение: ключевое слово `Thread` и затем в квадратных скобках три параметра: имя потока, его приоритет и имя группы потока. По умолчанию имя главного потока `main`, приоритет равен `5`, а имя группы потока тоже `main`. То есть для главного потока сообщение будет иметь вид `Thread[main,5,main]`.

Группы потоков - группы, в которые объединяются потоки для повышения уровня управляемости и степени безопасности. Некоторые действия можно выполнять сразу для всех потоков группы.

Командой `t.setName("Это самый главный поток!")` изменяется имя потока. Считывание приоритета потока выполняется инструкцией `int p=t.getPriority()`. Затем с помощью команды `t.setPriority(++p)` на единицу увеличивается значение приоритета процесса. Затем снова выполняется вывод сведений о потоке, для чего используем команду `System.out.println(t)`.

После всех этих действий на некоторое время приостанавливаем выполнение главного потока (выполнение программы). Используем при этом команду `Thread.sleep(time)`, предварительно выведя на экран текстовое сообщение о том, что поток будет приостановлен. Перед завершением работы программы выводим текстовое сообщение `"Работа программы завершена!"`.

На заметку:

Статический метод `sleep()` вызывался через имя класса `Thread`. То же можно было сделать через объектную переменную `t`. Обращаем также внимание на то, что между выводом последнего и предпоследнего сообщений в консольном окне проходит определенное время (точнее, 2500 миллисекунд, или 2.5 секунды).

Создание дочернего потока

Выполнение потока — это выполнение метода `run()`, который объявлен в интерфейсе `Runnable` (причем это единственный метод, объявленный в интерфейсе). То есть, чтобы создать поток, как минимум где-то нужно описать метод `run()`. Удобно это сделать в классе, который реализует интерфейс `Runnable`. Класс `Thread` реализует интерфейс `Runnable`. Однако в классе `Thread` метод `run()` имеет пустую реализацию (то есть описан без рабочего кода, с пустым телом). Поэтому напрямую воспользоваться классом `Thread` не получится, как и обойтись без него — запускается поток методом `start()`, который вызывается из объекта класса `Thread`. Поэтому у нас вырисовываются следующие перспективы относительно создания нового потока:

1. Реализовать в классе пользователя интерфейс `Runnable`, описав в классе метод `run()`. Это тот самый код, который выполняется в рамках нового потока. В этом классе также необходимо предусмотреть создание объекта класса `Thread` и вызов из этого объекта метода `start()`. Обычно все это реализуется в конструкторе класса.
2. Сделать все точно то же самое, но создать класс пользователя не расширением интерфейса `Runnable`, а наследуя класс `Thread`. В этом случае в качестве объекта, из которого вызывается метод `start()`, может использоваться экземпляр класса пользователя.

Рассмотрим оба способа создания потоков. Обратимся к листингу 11.2.

Листинг 11.2. Создание дочернего потока

```
// Импорт пакета (для генерирования случайных чисел):
import java.util.*;
// Класс реализует интерфейс Runnable:
class MyThread implements Runnable{
// Описание метода run():
public void run(){
System.out.println("Дочерний поток начал работу: "+new Date());
Random rnd=new Random();
for(int i=1;i<=10;i++){
try{
Thread.sleep(1000+rnd.nextInt(4000));
System.out.println("Дочерний поток: "+new Date());
}catch(InterruptedException e){ // Обработка исключения
System.out.println(e);}
}
System.out.println("Дочерний поток завершил работу: "+new Date());
}
```

```
// Конструктор класса:
MyThread(){
// Создание и запуск потока:
new Thread(this).start();}
}
public class SimpleThreadDemo{
// Главный метод может выбрасывать исключение:
public static void main(String[] args) throws InterruptedException{
System.out.println("Главный поток начал работу: "+new Date());
Random rnd=new Random();
// Запуск дочернего потока:
new MyThread();
for(int i=1;i<5;i++){
Thread.sleep(5000+rnd.nextInt(5000));
System.out.println("Главный поток: "+new Date());}
System.out.println("Главный поток завершил работу: "+new Date());}
}
```

Интерфейс `Runnable` реализуется в классе `MyThread`. В классе описывается метод `run()`. Код этого метода определяет действия, выполняемые при запуске дочернего потока. В частности, в методе командой `System.out.println("Дочерний поток начал работу: "+new Date())` выводится сообщение о начале выполнения дочернего потока и системная дата и время.

На заметку:

Обращаем внимание, что получить системную дату и время можно с помощью инструкции `new Date()`, которой создается анонимный объект класса `Date` с текущей системной датой и временем.

Командой `Random rnd=new Random()` создается объект `rnd` для генерирования случайных чисел. Далее в операторе цикла (десять итераций) выполняется приостановка выполнения потока. Для приостановки потока используется инструкция `Thread.sleep(1000+rnd.nextInt(4000))`. Аргументом метода `sleep()` указана инструкция `1000+rnd.nextInt(4000)`. Это случайное число в диапазоне от `1000` до `5000`. Другими словами, дочерний поток прерывается на время от 1 до 5 секунд. Затем, после окончания ожидания потоком с помощью команды `System.out.println("Дочерний поток: "+new Date())` выводится сообщение с текущей системной датой и временем. Поскольку метод `sleep()` может вызывать исключение класса `InterruptedException`, это исключение обрабатывается с помощью блока `try-catch`. После завершения оператора цикла командой `System.out.println("Дочерний поток завершил работу: "+new Date())` выводится сообщение о завершении выполнения дочернего потока.

Конструктор класса MyThread состоит всего из одной команды new Thread(this).start(). Этой командой создается анонимный объект класса Thread (инструкция new Thread(this)), и из этого анонимного объекта вызывается метод start(), что, собственно, и приводит к запуску потока на выполнение.

На заметку:

У класса Thread есть несколько конструкторов. В данном случае использован тот вариант конструктора, когда аргументом указывается объект класса (реализующего интерфейс Runnable), метод run() которого будет выполняться при запуске потока. Здесь нужно использовать метод run(), описанный в классе MyThread. Поэтому аргументом конструктору Thread() указана ссылка this на объект класса.

В главном методе программы в классе SimpleThreadDemo командой System.out.println("Главный поток начал работу: "+new Date()) выводится сообщение (с текущей системной датой) о том, что главный поток начал работу. Затем командой Random rnd=new Random() создается объект rnd для генерирования случайных чисел.

Важной командой является инструкция new MyThread(), которой создается анонимный объект класса MyThread. На самом деле, поскольку в конструкторе класса MyThread запускается дочерний поток, выполнение инструкции new MyThread() приводит к запуску дочернего потока. После этого запускается оператор цикла (пять итераций). Командой Thread.sleep(5000+rnd.nextInt(5000)) выполняется задержка в выполнении основного потока на время от 5 до 10 секунд. Командой System.out.println("Главный поток: "+new Date()) в теле оператора цикла выводится сообщение с системной датой и временем. После завершения оператора цикла командой System.out.println("Главный поток завершил работу: "+new Date()) выводится сообщение о завершении работы основного потока.

На заметку:

Поскольку в методе main() используется метод sleep() и возможное исключение класса InterruptedException не обрабатывается в методе, сигнатура метода содержит инструкцию throws InterruptedException.

Результат выполнения программы может иметь следующий вид:

```
Главный поток начал работу: Sat Oct 09 14:06:23 EEST 2010
Дочерний поток начал работу: Sat Oct 09 14:06:23 EEST 2010
Дочерний поток: Sat Oct 09 14:06:28 EEST 2010
Главный поток: Sat Oct 09 14:06:30 EEST 2010
Дочерний поток: Sat Oct 09 14:06:30 EEST 2010
```

```
Дочерний поток: Sat Oct 09 14:06:32 EEST 2010
Дочерний поток: Sat Oct 09 14:06:34 EEST 2010
Главный поток: Sat Oct 09 14:06:36 EEST 2010
Дочерний поток: Sat Oct 09 14:06:36 EEST 2010
Дочерний поток: Sat Oct 09 14:06:38 EEST 2010
Дочерний поток: Sat Oct 09 14:06:40 EEST 2010
Дочерний поток: Sat Oct 09 14:06:41 EEST 2010
Дочерний поток: Sat Oct 09 14:06:45 EEST 2010
Главный поток: Sat Oct 09 14:06:46 EEST 2010
Дочерний поток: Sat Oct 09 14:06:49 EEST 2010
Дочерний поток завершил работу: Sat Oct 09 14:06:49 EEST 2010
Главный поток: Sat Oct 09 14:06:55 EEST 2010
Главный поток завершил работу: Sat Oct 09 14:06:55 EEST 2010
```

Теперь рассмотрим аналогичный пример, но только теперь класс пользователя создается наследованием класса `Thread`. Программный код этого примера представлен в листинге 11.3.

Листинг 11.3. Создание еще одного потока

```
// Импорт пакета (для генерирования случайных чисел):
import java.util.*;
// Класс наследует класс Thread:
class MyThread extends Thread{
// Описание метода run():
public void run(){
System.out.println("Дочерний поток начал работу: "+new Date());
Random rnd=new Random();
for(int i=1;i<=10;i++){
try{
sleep(1000+rnd.nextInt(4000));
System.out.println("Дочерний поток: "+new Date());
}catch(InterruptedException e){ // Обработка исключения
System.out.println(e);}
}
System.out.println("Дочерний поток завершил работу: "+new Date());
}
// Конструктор класса:
MyThread(){
// Создание и запуск потока:
start();}
}
public class OneMoreThreadDemo{
// Главный метод может выбрасывать исключение:
public static void main(String[] args) throws InterruptedException{
System.out.println("Главный поток начал работу: "+new Date());
Random rnd=new Random();
```

```
// Запуск дочернего потока:
new MyThread();
for(int i=1;i<5;i++){
Thread.sleep(5000+rnd.nextInt(5000));
System.out.println("Главный поток: "+new Date());}
System.out.println("Главный поток завершил работу: "+new Date());}
}
```

Результат выполнения программы может быть таким:

```
Главный поток начал работу: Sat Oct 09 16:51:46 EEST 2010
Дочерний поток начал работу: Sat Oct 09 16:51:46 EEST 2010
Дочерний поток: Sat Oct 09 16:51:48 EEST 2010
Дочерний поток: Sat Oct 09 16:51:51 EEST 2010
Главный поток: Sat Oct 09 16:51:54 EEST 2010
Дочерний поток: Sat Oct 09 16:51:56 EEST 2010
Дочерний поток: Sat Oct 09 16:51:58 EEST 2010
Дочерний поток: Sat Oct 09 16:52:02 EEST 2010
Дочерний поток: Sat Oct 09 16:52:03 EEST 2010
Главный поток: Sat Oct 09 16:52:03 EEST 2010
Дочерний поток: Sat Oct 09 16:52:05 EEST 2010
Дочерний поток: Sat Oct 09 16:52:08 EEST 2010
Главный поток: Sat Oct 09 16:52:09 EEST 2010
Дочерний поток: Sat Oct 09 16:52:09 EEST 2010
Дочерний поток: Sat Oct 09 16:52:14 EEST 2010
Дочерний поток завершил работу: Sat Oct 09 16:52:14 EEST 2010
Главный поток: Sat Oct 09 16:52:16 EEST 2010
Главный поток завершил работу: Sat Oct 09 16:52:16 EEST 2010
```

Поскольку этот пример очень похож на предыдущий, остановимся только на их принципиальных различиях. Итак, класс `MyThread` теперь наследует класс `Thread`. Описание метода `run()` не изменилось. Единственно, по-другому вызывается метод `sleep()`. Раньше он вызывался в формате `Thread.sleep()`, теперь просто указано имя метода. Здесь мы воспользовались тем, что класс `MyThread` является подклассом класса `Thread` и поэтому метод `sleep()` можно вызвать из объекта класса `MyThread`.

В конструкторе класса `MyThread` вызывается метод `start()`, которым запускается дочерний поток. Здесь создавать объект класса `Thread` необходимости нет, и метод вызывается из создаваемого объекта класса `MyThread`. Это возможно, поскольку класс `MyThread` является подклассом класса `Thread`. Главный метод программы при этом не изменился, равно как и логика выполнения программы.

Синхронизация потоков

В этой главе также кратко рассмотрим задачу по синхронизации потоков. Чтобы понять суть проблемы, сначала рассмотрим простой иллюстративный пример, в котором синхронизация не выполняется.

Листинг 11.4. Пример без синхронизации

```
// Подключение пакета (для генерирования случайных чисел):
import java.util.*;
// Класс для создания объекта - общего ресурса для потоков:
class MyClass{
   // Поле класса:
   private int count;
   // Конструктор класса:
   MyClass(){
      count=0;}
   // Метод для присваивания значения полю:
   void set(int count,String name){
      this.count=count;
      System.out.println(name+"Установлено значение поля "+count+": "+new Date());}
   // Метод для считывания значения поля:
   int get(String name){
      System.out.println(name+"Считано значение поля "+count+": "+new Date());
      return count;}
   }
// Класс для создания потоков:
class MyThread extends Thread{
   // Поле - тип потока:
   private boolean UpDown;
   // Поле - имя потока:
   private String name;
   // Объект - ресурс для потока:
   private MyClass obj;
   // Переопределение метода run():
   public void run(){
   // Объект для генерирования случайных чисел:
   Random rnd=new Random();
   // Локальная переменная:
   int number;
   for(int i=1;i<=3;i++){
      try{
         // Считывание поля:
         number=obj.get(name);
         // Изменение значения поля:
         if(UpDown) number++;
            else number--;
         // Задержка в выполнении метода:
```

```
            sleep(1000+rnd.nextInt(9000));
            // Изменение значения поля:
            obj.set(number,name);}
            // Обработка исключения:
         catch(InterruptedException e){
         System.out.println(e);}
   }}
// Конструктор класса:
MyThread(boolean UpDown,MyClass obj){
   this.UpDown=UpDown;
   if(UpDown) name="Up-поток. ";
   else name="Down-поток. ";
   this.obj=obj;
   // Запуск потока:
   start();}
}
public class SynchThreadDemo{
   public static void main(String[] args) throws InterruptedException{
      // Объект - ресурс:
      MyClass obj=new MyClass();
      // Поток, увеличивающий значение поля:
      MyThread thA=new MyThread(true,obj);
      // Поток, уменьшающий значение поля:
      MyThread thB=new MyThread(false,obj);
      // Главный поток будет ожидать окончания первого потока:
      thA.join();
      // Главный поток будет ожидать окончания второго потока:
      thB.join();
      // Окончание главного потока:
      System.out.print("Окончательно: ");
      obj.get("");}
   }
```

Программа работает так. В ней (в главном методе) создается объект класса пользователя с числовым полем, которое при создании объекта получает нулевое значение. В главном методе программы запускаются два дочерних потока, каждый из которых обращается к одному и тому же объекту, который в данном случае выступает в роли общего ресурса. Каждый из потоков считывает значение поля объекта-ресурса и изменяет значение этого поля. В каждом потоке выполняется три итерации. Первый поток за каждую итерацию на единицу увеличивает значение поля, а второй поток на единицу уменьшает значение поля. Особенность ситуации состоит в том, что, во-первых, потоки выполняются одновременно. Но дело даже не в этом. Важно то, что между считыванием значения поля и присваиванием нового значения каждый из потоков приостанавливает свое выполнение на время от 1 до 10 секунд. В программе контролируется значение поля объекта в разные моменты времени.

В принципе, можно было бы ожидать, что после выполнения программы значение поля должно остаться нулевым - трижды поле увеличивается на единицу и трижды уменьшается на единицу. Суммарное изменение равно нулю. Но это в теории. На практике все не совсем так. Представим, что может произойти. Например, первый поток считал значение. Для определенности пускай это будет значение `0`. Затем в первом потоке полю должно быть присвоено значение `1`. Но между считыванием и записью есть время ожидания потока. За это время второй поток также может считать нулевое значение поля. О том, что поле уже считано первым потоком, второму потку неизвестно. Поэтому второй поток попытается присвоить полю значение `-1`. В результате поле получит значение `1` или `-1` (но никак не `0`) в зависимости от того, какой из потоков будет последним выполнять присваивание значения полю. Учитывая, что интервал между считыванием и присваиванием значения полю объекта является случайным, то окончательное значение поля объекта после выполнения обоих потоков является случайным. Теперь детально проанализируем программный код.

В классе `MyClass` объявляется закрытое целочисленное поле `count`. В конструкторе класса (без аргументов) полю присваивается нулевое значение. Метод для присваивания значения полю называется `set()`, и методу передаются два аргумента. Первый целочисленный аргумент присваивается полю `count` в качестве значения. Второй текстовый аргумент `name` используется в команде `System.out.println(name+"Установлено значение поля "+count+": "+new Date())` и служит для индикации потока, из которого будет вызываться метод. Также для удобства в этом сообщении выводится системная дата и время.

Метод для считывания значения поля `count` называется `get()` и имеет текстовый аргумент, назначение которого такое же, как и назначение текстового аргумента в методе `set()`, то есть для индикации потока. В методе, кроме того, что он возвращает значение поля count, командой `System.out.println(name+"Считано значение поля "+count+": "+new Date())` выводится сообщение о считанном значении поля.

Потоковая модель программы реализуется через класс `MyThread`, который наследует класс `Thread`. У класса есть три закрытых поля. Логическое поле `UpDown` используется в качестве индикатора для определения того, какой поток создается — поток, увеличивающий значение объекта-ресурса, или поток, уменьшающий значение объекта-ресурса. Ссылка на объект-ресурс также является полем класса. Это поле называется `obj` и является объектной ссылкой класса `MyClass`. Также у класса есть текстовое поле `name`, которое мы используем в качестве имени потока. Это поле будет использовано при вызовах методов `set()` и `get()` объекта `obj`.

На заметку:

Вообще-то можно было воспользоваться методами присваивания и считывания имени потока, но мы ими не воспользовались и пошли более простым путем.

При переопределении метода `run()` командой `Random rnd=new Random()` создается объект `rnd` для генерирования случайных чисел (необходимо при определении интервала задержки в выполнении потока). Целочисленная переменная `number` нужна для запоминания значения поля `count`.

В операторе цикла на три итерации командой `number=obj.get(name)` считывается значение поля count объекта-ресурса `obj` и записывается в переменную `number`. Затем выполняется условный оператор:

```
if(UpDown) number++;
else number--;
```

В данном случае, в зависимости от значения поля `UpDown`, значение переменной `number` увеличивается или уменьшается на единицу. После этого командой `sleep(1000+rnd.nextInt(9000))` выполняется приостановка в выполнении объекта на время от 1 до 10 секунд. После того, как пауза выдержана, командой `obj.set(number,name)` присваивается новое значение полю `count` объекта `obj`.

На заметку:

В операторе цикла также выполняется обработка исключительной ситуации `InterruptedException`.

Конструктор класса `MyThread` принимает два аргумента: логическое для определения значения поля `UpDown` и ссылку на объект-ресурс (объектная переменная класса `MyClass`). В конструкторе, в зависимости от значения поля `UpDown`, полю `name` присваивается значение `"Up-поток. "` или `"Down-поток. "`. Командой `start()` выполняется запуск потока.

В главном методе программы командой `MyClass obj=new MyClass()` создается объект класса `MyClass`. Это ресурс, который будет использоваться двумя дочерними потоками. Объекты потоков создаются соответственно командами `MyThread thA=new MyThread(true,obj)` и `MyThread thB=new MyThread(false,obj)`. Первый поток (с логическим аргументом `true`) будет увеличивать значение поля `count` объекта `obj`, который указан вторым аргументом конструктора класса `MyThread`. Второй поток (с логическим аргументом `false`) будет уменьшать значение поля `count` объекта `obj` (второй аргумент конструктора класса `MyThread`). Следующие две команды необходимы для того, чтобы главный поток программы не завершался до того, как будут завершены дочерние методы. Например, команда `thA.join()` означает, что главный метод (команда вызывается

из главного метода) будет ожидать завершения потока thA. Аналогично команда thB.join() означает, что главный метод будет ожидать завершения потока thB. После того, как выполнение дочерних потоков завершено, выполняются команды System.out.print("Окончательно: ") и obj.get(""). В последнем случае метод get() вызывается с формальным аргументом - пустым текстовым значением.

Результат выполнения программы может иметь такой вид:

```
Up-поток. Считано значение поля 0: Sat Oct 09 19:00:41 EEST 2010
Down-поток. Считано значение поля 0: Sat Oct 09 19:00:41 EEST 2010
Down-поток. Установлено значение поля -1: Sat Oct 09 19:00:42 EEST 2010
Down-поток. Считано значение поля -1: Sat Oct 09 19:00:42 EEST 2010
Up-поток. Установлено значение поля 1: Sat Oct 09 19:00:43 EEST 2010
Up-поток. Считано значение поля 1: Sat Oct 09 19:00:43 EEST 2010
Down-поток. Установлено значение поля -2: Sat Oct 09 19:00:43 EEST 2010
Down-поток. Считано значение поля -2: Sat Oct 09 19:00:43 EEST 2010
Up-поток. Установлено значение поля 2: Sat Oct 09 19:00:48 EEST 2010
Up-поток. Считано значение поля 2: Sat Oct 09 19:00:48 EEST 2010
Down-поток. Установлено значение поля -3: Sat Oct 09 19:00:49 EEST 2010
Up-поток. Установлено значение поля 3: Sat Oct 09 19:00:58 EEST 2010
Окончательно: Считано значение поля 3: Sat Oct 09 19:00:58 EEST 2010
```

Несложно убедиться, что финальный результат далек от идеального. Чтобы исправить ситуацию, выполняем *синхронизацию* потоков.

Есть два способа синхронизировать потоки. В обоих случаях используется ключевое слово synchronized. Чтобы выполнить синхронизацию метода (в этом случае доступ к нему может иметь только один поток), в сигнатуре метода указывается ключевое слово synchronized. Чтобы выполнить синхронизацию ресурса (объекта), используем такой шаблон:

```
synchronized(объект_синхронизации){
   код_синхронизации
   }
```

После ключевого слова synchronized в круглых скобках указывается объект, доступ к которому потоки должны выполнять синхронно, то есть по очереди. В фигурных скобках указывается код, который выполняется в режиме синхронной работы потоков. В данном конкретном случае достаточно изменить описание метода run() в классе MyThread следующим образом (добавленный код выделен для удобства жирным шрифтом и отмечен комментарием):

```
// Переопределение метода run():
public void run(){
   // Объект для генерирования случайных чисел:
   Random rnd=new Random();
```

```
// Локальная переменная:
int number;
for(int i=1;i<=3;i++){
   // Синхронизация потоков при обращении к ресурсу:
   synchronized(obj){
   try{
      // Считывание поля:
      number=obj.get(name);
      // Изменение значения поля:
      if(UpDown) number++;
        else number--;
      // Задержка в выполнении метода:
      sleep(1000+rnd.nextInt(9000));
      // Изменение значения поля:
      obj.set(number,name);}
   // Обработка исключения:
   catch(InterruptedException e){
      System.out.println(e);}
   }
}}
```

Если описать метод `run()` так, как указано выше, оставив неизменным прочий код рассмотренного примера, то результат выполнения программы может быть следующим:

```
Up-поток. Считано значение поля 0: Sat Oct 09 18:56:03 EEST 2010
Up-поток. Установлено значение поля 1: Sat Oct 09 18:56:09 EEST 2010
Up-поток. Считано значение поля 1: Sat Oct 09 18:56:09 EEST 2010
Up-поток. Установлено значение поля 2: Sat Oct 09 18:56:19 EEST 2010
Down-поток. Считано значение поля 2: Sat Oct 09 18:56:19 EEST 2010
Down-поток. Установлено значение поля 1: Sat Oct 09 18:56:28 EEST 2010
Down-поток. Считано значение поля 1: Sat Oct 09 18:56:28 EEST 2010
Down-поток. Установлено значение поля 0: Sat Oct 09 18:56:38 EEST 2010
Down-поток. Считано значение поля 0: Sat Oct 09 18:56:38 EEST 2010
Down-поток. Установлено значение поля -1: Sat Oct 09 18:56:42 EEST 2010
Up-поток. Считано значение поля -1: Sat Oct 09 18:56:42 EEST 2010
Up-поток. Установлено значение поля 0: Sat Oct 09 18:56:49 EEST 2010
Окончательно: Считано значение поля 0: Sat Oct 09 18:56:49 EEST 2010
```

В этом случае окончательное значение поля равно 0, как и ожидалось первоначально. Это следствие выполненной синхронизации программного кода в теле метода `run()`. Если какой-то поток считал значение поля count, то другой поток не получит доступа к объекту, пока первый не внесет изменение в значение этого поля.

Тема работы с потоками в Java достаточно обширна и, очевидно, выходит за рамки тематики книги. Но хочется верить, что читатель уловил основную

идею в использовании многопоточной модели и в случае необходимости сможет ею воспользоваться.

Резюме

1. Несколько фрагментов кода программы могут выполняться одновременно (параллельно). В этом случае говорят о потоках выполнения.
2. В Java поддерживается встроенная многопоточная модель.
3. Арсенал средств многопоточного программирования реализуется через класс `Thread` и интерфейс `Runnable`.
4. Поток в Java — это объект класса `Thread`, его подкласса или класса, реализующего интерфейс `Runnable`.
5. Для получения ссылки на поток используют статический метод `currentThread()` класса `Thread`.
6. Для создания нового потока необходимо описать метод `run()`, объявленный в интерфейсе `Runnable` (в классе `Thread` метод `run()` имеет пустую реализацию). Этот метод определяет набор действий, выполняемых в рамках потока.
7. Для запуска потока из `Thread`-объекта вызывается метод `start()`.
8. В некоторых случаях необходимо выполнять синхронизацию - регулировать совместный доступ потоков к общим ресурсам.

Глава 12
Система ввода/вывода

Нет, это не Рио-де-Жанейро.
Это гораздо хуже...
(Из к/ф "Золотой теленок")

Система ввода/вывода - это набор утилит и программных механизмов, применяемых для ввода информации (данных) в программу и вывода результатов выполнения программы в наглядном и понятном для пользователя виде. Что касается вывода, то с ним мы работали достаточно много - в каждой программе так или иначе информация выводилась или в консольное окно, или посредством диалогового окна. С системой ввода мы сталкивались меньше. По большому счету, единственный способ передачи данных программе, с которым мы имели дело, состоял в использовании стандартного диалогового окна с полем ввода, создаваемого средствами класса `JOptionPane` библиотеки Swing. Как реализуется консольный ввод, в книге еще не описывалось. И это не случайно. Дело в том, что в Java система консольного ввода достаточно нетривиальна, особенно в начальных версиях JDK. С другой стороны, обычно программы на Java пишут не для того, чтобы вводить данные в консольном окне. Обычно приложения Java имеют графический интерфейс. Собственно, и технология Java создавалась не для того, чтобы писать консольные приложения. Поэтому в отношении консольного ввода мы поступим мудро и просто. Рассмотрим только наиболее простой способ реализации консольного ввода, который доступен начиная с версии JDK 5. Кроме этого, рассмотрим еще один очень важный способ обмена данными — через файловую систему, путем записи данных в файлы и считывания данных из файлов.

Потоки данных и консольный ввод

Ввод и вывод данных в Java реализуется через *потоки*. И это не те потоки, что обсуждались в предыдущей главе. В данном случае под потоками подразумеваются *потоки данных*, которые в англоязычной литературе обозначаются термином *stream*. Потоки данных обычно делят на *символьные потоки* и *байтовые потоки*. Исторически первыми появились байтовые потоки. Они используются для записи и считывания данных в числовом (двоичном) коде. Символьные потоки предназначены, как несложно догадаться, для обработки символов. Проблема в том, что символ в Java - это два байта. То есть для обработки символов нужно записывать и считывать по два байта. Поэтому символьный поток данных представляет собой своео-

бразную "упаковку" для байтового потока. В подробности того, что, куда и как упаковывается, здесь вдаваться не будем. Ограничимся лишь тем, что рассмотрим наиболее простые механизмы считывания данных с консоли.

В Java есть класс `System`, у которого три статических поля `in`, `out` и `err`, которые являются потоками данных (потоковыми переменными) соответственно ввода, вывода и потока ошибок. С потоком `System.out`, который называется *потоком стандартного вывода*, мы много раз сталкивались - в основном, вызывая из него метод `println()`. Если никакие дополнительные действия не предпринимались, то поток стандартного вывода связан с консольным устройством. Именно в консоль по умолчанию выполняется вывод. С объектом `System.err`, который называется *потоком ошибок*, мы дела не имели и иметь не будем. Этот поток также связан с консолью. Наконец, объект `System.in` называется *стандартным потоком ввода*. Стандартный поток ввода связан с клавиатурой. К сожалению, так же просто считать данные с клавиатуры с помощью объекта `System.in`, как вывести данные на консоль с помощью объекта `System.out`, не получится. Тем не менее, существует простой путь.

На заметку:

При желании стандартные потоки ввода/вывода (то есть `System.out`, `System.err` и `System.in`) можно перенаправить на другие устройства ввода/вывода, отличные от консоли и клавиатуры.

В пакете `java.util` есть класс `Scanner`. С помощью этого класса можно достаточно просто организовать ввод данных с клавиатуры. Общая рекомендация состоит в том, чтобы создать объект класса `Scanner` и связать этот объект со стандартным потоком ввода `System.in`. Процедура банальная: при вызове конструктора класса `Scanner` объект стандартного ввода `System.in` указывается аргументом конструктора. После этого с помощью методов созданного таким образом объекта класса `Scanner` можно выполнять считывание данных, вводимых с клавиатуры. Некоторые методы класса `Scanner` перечислены в табл. 12.1.

Табл. 12.1. Некоторые методы класса `Scanner`

Метод	Описание
`hasNext()`	Методом возвращается логическое значение `true`, если в строке ввода еще есть слова, и `false` в противном случае
`hasNextInt()`	Методом возвращается логическое значение `true`, если в строке ввода еще есть `int`-значение, и `false` в противном случае

Метод	Описание
`hasNextLong()`	Методом возвращается логическое значение `true`, если в строке ввода еще есть `long`-значение, и `false` в противном случае
`hasNextDouble()`	Методом возвращается логическое значение `true`, если в строке ввода еще есть `double`-значение, и `false` в противном случае
`hasNextBoolean()`	Методом возвращается логическое значение `true`, если в строке ввода еще есть логическое значение, и `false` в противном случае
`hasNextByte()`	Методом возвращается логическое значение `true`, если в строке ввода еще есть `byte`-значение, и `false` в противном случае
`hasNextShort()`	Методом возвращается логическое значение `true`, если в строке ввода еще есть `short`-значение, и `false` в противном случае
`hasNextLine()`	Методом возвращается логическое значение `true`, при наличии строки ввода, и `false` в противном случае
`hasNextFloat()`	Методом возвращается логическое значение `true`, если в строке ввода еще есть `float`-значение, и `false` в противном случае
`next()`	Методом возвращается следующее слово в строке ввода
`nextInt()`	Методом возвращается следующее `int`-значение в строке ввода
`nextLong()`	Методом возвращается следующее `long`-значение в строке ввода
`nextDouble()`	Методом возвращается следующее `double`-значение в строке ввода
`nextBoolean()`	Методом возвращается следующее логическое значение в строке ввода
`nextByte()`	Методом возвращается следующее `byte`-значение в строке ввода
`nextShort()`	Методом возвращается следующее `short`-значение в строке ввода
`nextLine()`	Методом возвращается текстовая строка ввода
`nextFloat()`	Методом возвращается следующее `float`-значение в строке ввода

Пример простой программы, в которой используется консольный ввод (ввод данных с клавиатуры), приведен в листинге 12.1.

Листинг 12.1. Пример ввода текста с клавиатуры

```
// Импорт пакета для использования класса Scanner:
import java.util.*;
public class ScannerDemo{
   public static void main(String[] args){
      // Объект класса Scanner:
      Scanner input=new Scanner(System.in);
      // Текстовая переменная (имя):
      String name;
      // Целочисленная переменная (возраст):
      int age;
      // double-переменная (вес):
      double weight;
      // Как Вас зовут?
      System.out.print("Как Вас зовут? ");
      // Считывание полного имени пользователя:
      name=input.nextLine();
      // Приветствие (с именем):
      System.out.println("Добрый день, "+name+"!");
      // Сколько Вам лет?
      System.out.print("Сколько Вам лет? ");
      // Считывание возраста:
      age=input.nextInt();
      // Сообщение (с указанием возраста):
      System.out.println("Вам "+age+" лет!");
      // Укажите вес (в кг):
      System.out.print("Укажите Ваш вес (в кг): ");
      // Считывание веса:
      weight=input.nextDouble();
      // Сообщение (с указанием веса):
      System.out.println("Ваш вес составляет "+weight+" кг.");
   }}
```

Для использования класса `Scanner` инструкцией `import java.util.*` подключается соответствующий пакет. В классе `ScannerDemo` в методе `main()` командой `Scanner input=new Scanner(System.in)` создается объект класса `Scanner`. Также объявляются переменные: текстовая `name` для записи в нее имени пользователя, целочисленная `age` для записи в нее возраста пользователя и переменная `weight` типа `double` для записи веса пользователя.

Командой `System.out.print("Как Вас зовут?")` выводится сообщение `"Как Вас зовут?"`. После этого командой `name=input.nextLine()` выполняется считывание в формате текстовой строки того, что ввел пользователь с клавиатуры. Результат считывания записывается в переменную `name`. Значение этой переменной использовано в команде `System.out.`

println("Добрый день, "+name+"!"), которой выводится приветствие с указанием имени пользователя. Еще одной командой System.out.print("Сколько Вам лет? ") выводится приглашение указать свой возраст. Введенное пользователем значение считывается как целое число с помощью команды age=input.nextInt(), причем результат записывается в переменную age. Считанное значение для возраста пользователя отображается в консольном окне с помощью команды System.out.println("Вам "+age+" лет!"). Наконец, командой System.out.print("Укажите Ваш вес (в кг): ") выводится сообщение с просьбой указать свой вес. Командой weight=input.nextDouble() введенное значение считывается как число типа double, результат записывается в переменную weight, и командой System.out.println("Ваш вес составляет "+weight+" кг.") выводится на экран сообщение об указанном весе пользователя.

На заметку:

Вообще-то в килограммах измеряется *масса*, а не *вес*. Но в данном случае это неважно.

Результат выполнения программы может выглядеть следующим образом (жирным шрифтом выделены вводимые пользователем в процессе выполнения программы данные):

```
Как Вас зовут? Иван Петров
Добрый день, Иван Петров!
Сколько Вам лет? 59
Вам 59 лет!
Укажите Ваш вес (в кг): 78,325
Ваш вес составляет 78.325 кг.
```

В принципе все достаточно просто. Создаем объект input класса Scanner, а затем через этот объект вызываются, в зависимости от потребностей, методы (например, те, что представлены в табл. 12.1).

Форматированный вывод

Здесь кратко опишем способы форматированного вывода. **Под форматированным выводом будем подразумевать явное определение параметров представления данных в консольном окне — таких, например, как количество цифр в дробной части действительных чисел, наличие или отсутствие знака у положительных чисел, параметры данных типа дата/время и многое другое.**

Основу форматированного консольного вывода составляет метод printf(), который вызывается из объекта стандартного вывода System.out. Метод появился в версии JDK 5.

На заметку:

Метод `printf()` аналогичен одноименному методу из языка C.

При вызове метода `printf()` его первым аргументом указывается текстовая строка, которая и содержит информацию о применяемом форматировании. Непосредственно выводимые на экран (консоль) значения указываются вторым, третьим (и так далее) аргументами метода. Эта очень важная самая первая текстовая строка формируется по определенным правилам. При этом используются так называемые *символы форматирования*. Следует помнить несколько важных правил.

Методом `printf()` может выводиться на экран сразу несколько значений (в том порядке, как они указаны в аргументах метода). Эти значения указываются, как уже отмечалось, начиная со второго аргумента метода. Для каждого из отображаемых значений можно предусмотреть собственный формат вывода. Это немаловажное обстоятельство, особенно если учесть, что выводимые на экран данные могут относиться к разным типам. Текстовая строка, определяющая формат данных, содержит фрагменты для форматирования каждого из выводимых на экран аргументов. В качестве разделителя фрагментов используется символ `%`. С этого символа начинается текстовая строка форматирования. Некоторые другие символы, которые могут использоваться в строке форматирования, приведены в табл. 12.2.

Табл. 12.2. Некоторые символы форматирования для метода `printf()`

Символ форматирования	Описание
`d`	Целое (десятичное) число
`x`	Целое шестнадцатеричное число
`o`	Целое восьмеричное число
`f`	Действительное число с плавающей (или фиксированной) точкой
`e`	Действительное число с плавающей точкой в экспоненциальном формате
`g`	Число с плавающей точкой в общем формате
`a`	Действительное число с плавающей точкой в шестнадцатеричном представлении
`s`	Текстовая строка
`c`	Символ
`b`	Логическое значение
`h`	Хэш-код

Символ форматирования	Описание
%	Символ процента. Разделитель блоков в строке форматирования
+	Инструкция выводить знак для положительных чисел
пробел	Инструкция добавлять пробел перед положительными числами
0	Инструкция выводить ведущие нули
(	Инструкция заключать отрицательные числа в скобки
,	Использование *системного* разделителя групп (тысячных разрядов)
#	Инструкция для действительных чисел всегда добавлять десятичную точку. Для шестнадцатеричных и восьмеричных чисел означает добавление префикса 0x и 0 соответственно
$	Инструкция используется для индексирования аргументов в строке форматирования. Индекс указывается перед символом $
<	Инструкция используется в качестве ссылки на аргумент, который обрабатывался в предыдущем блоке форматирования

Общий шаблон блока текстовой строки, определяющей формат вывода аргумента, имеет следующий вид:

`%[`*`индекс_аргумента`*`$][`*`индикатор`*`][`*`размер_поля`*`][.`*`знаков_после_запятой`*`]`*`символ_типа_данных`*

В этом шаблоне квадратные скобки означают необязательный фрагмент. Как отмечалось, блок начинается с символа %. Затем может быть указан индекс аргумента, для которого применяется формат. Единица означает аргумент, первый после текстовой строки форматирования. Второй индекс означает второй, после текстовой строки формата, аргумент метода `printf()` и так далее. Далее может быть указан *индикатор* (который иногда еще называют *флагом*) - один из символов, определяющий особенности отображения данных (один из символов +, 0, (, ,, < или *пробел*). Затем идет комбинация *число-точка-число*. Первое число указывает количество знаков (позиций) поля, в котором представляется число (то есть общее количество символов в представлении числа), а второе число - количество знаков после десятичной запятой. Заканчивается блок форматирования символом, обозначающим тип отображаемых данных (один из символов d, x, o, f, e, g, a, s, c, b, h или tx).

На заметку:

В блоке *число-точка-число* первое *число* определяет минимальное количество символов, отображаемых при представлении числа.

Простые примеры использования разных способов вызова метода `printf()` с разными аргументами приведены в листинге 12.2.

Листинг 12.2. Форматированный вывод

```
import java.util.*;
public class PrintFDemo{
public static void main(String[] args){
String text="Текстовая строка";
double x=100.0/7.0;
double z=130;
int n=-1234567;
int k=7654321;
int m=0xABC;
int l=0123;
Date now=new Date();
System.out.printf("%s\t%d\t%f\n",text,k,z);
System.out.printf("Десятичные числа:\n%1$g\t%2$e\t%1$07.2f\n",x,z);
System.out.printf("Отрицательное число: %,(d\n",n);
System.out.printf("Положительное число: %+,d\n",k);
System.out.printf("16-е значение %x соответствует 10-му числу %<d\n",m);
System.out.printf("10-е значение %d соответствует 16-му числу %<X\n",k);
System.out.printf("8-е значение %o соответствует 10-му числу %<d\n",l);
System.out.printf("Месяц: %tB\n",now);
System.out.printf("Число: %te\n",now);
System.out.printf("День недели: %tA\n",now);
System.out.printf("Время: %tT\n",now);}
}
```

Результат выполнения этой программы имеет следующий вид:

```
Текстовая строка 7654321 130,000000
Десятичные числа:
14.2857 1.300000e+02 0014,29
Отрицательное число: (1 234 567)
Положительное число: +7 654 321
16-е значение abc соответствует 10-му числу 2748
10-е значение 7654321 соответствует 16-му числу 74CBB1
8-е значение 123 соответствует 10-му числу 83
Месяц: Октябрь
Число: 11
День недели: понедельник
Время: 13:33:55
```

Разберем детальнее программный код метода `main()`, который описан в классе `PrintFDemo`. В методе объявлено и инициализировано несколько переменных, которые затем отображаются, в разных комбинациях, с помощью метода `printf()`. В частности, в главном методе программы объявлены такие переменные: текстовая переменная `text` со значением

"Текстовая строка", действительные переменные x (результат деления 100.0 на 7.0) и z (значение 130 не имеет дробной части), целочисленные переменные n (отрицательное значение -1234567), k (положительное значение 7654321), m (шестнадцатеричное значение 0xABC) и l (восьмеричное значение 0123), а также объектная переменная now класса Date (значение - текущая системная дата и время).

На заметку:

Шестнадцатеричные литералы в Java начинаются с префикса 0x или 0X. Буквы A, B, C, D, E и F (или их "маленькие" собратья a, b, c, d, e и f) используются в представлении шестнадцатеричного числа для представления значений 10, 11, 12, 13, 14 и 15 соответственно. Восьмеричные литералы в Java начинаются с 0 (то есть начинаются с ведущего нуля).

Если шестнадцатеричный (восьмеричный) литерал или переменную с таким значением указать аргументом метода println(), то на экране будет выведено соответствующее десятичное значение.

Затем в главном методе программы вызывается несколько раз подряд метод printf() с разными аргументами. Фактически, разный способ отображения данных определяется строкой форматирования и списком аргументов метода printf(). Поэтому для удобства разные способы передачи аргументов этому методу, использованные в программе, перечислены и описаны далее.

1. Аргумент "%s\t%d\t%f\n",text,k,z. В данном случае на экран выводится значение текстовой переменной text, целочисленной переменной k и действительной переменной z. Это соответственно второй (первый отображаемый), третий (второй отображаемый) и четвертый (третий отображаемый) аргументы метода printf(). Первым аргументом указана строка форматирования "%s\t%d\t%f\n", которая означает, что первый из отображаемых аргументов является текстом (блок %s). Затем выполняется табуляция (символ табуляции \t между блоками форматов). Второй из отображаемых аргументов является целочисленным (блок %d). Снова табуляция, а третий из отображаемых аргументов является действительным числом (блок %f). Инструкция \n, как и ранее, означает переход к новой строке.

2. Аргумент "Десятичные числа:\n%1$g\t%2$e\t%1$07.2f\n",x,z. Здесь отображается два аргумента — double-переменные x и z. Ссылка на отображаемые аргументы в строке форматирования выполнена в явном виде, с использованием индексов. Инструкция 1$ означает первый отображаемый аргумент, то есть переменную x, а инструкция 2$ означает второй отображаемый аргумент, то есть переменную z. Обратите внимание, что переменная x отображается дважды, но с использованием разных форматов. Согласно строке форматирования сначала отображается

текст "Десятичные числа:" с последующим переходом к новой строке. Затем первый раз отображается переменная x в формате g (общий формат для действительных чисел), затем выполняется табуляция (символ \t). Второй аргумент (переменная z) отображается в формате десятичного числа с мантиссой и показателем степени (научная нотация, ей соответствует инструкция e). Наконец, снова отображается первый аргумент (переменная x), но уже в формате 07.2f, после чего выполняется переход к новой строке (инструкция \n). Формат 07.2f, в свою очередь, означает, что для отображения числа используется не менее 7 позиций, в десятичной части две цифры, а 0 в начале блока формата означает, что в случае необходимости (для заполнения всех позиций поля, в которое выводится число) будут добавляться ведущие незначащие нули. Обратите также внимание, что в форматах g и f используются разные разделители целой и дробной части — соответственно, точка (разделитель среды исполнения) и запятая (системный разделитель).

3. Аргумент "Отрицательное число: %,(d\n",n. Отображается отрицательное целочисленное значение, записанное в переменную n. Перед значением выводится текст "Отрицательное число: ", после чего само число в таком формате: число без знака минус, но в круглых скобках (инструкция (в блоке формата), с использованием системного разделителя тысячных разрядов (для русскоязычной операционной системы это, как правило, пробел). После вывода значения на экран выполняется переход к новой строке.

4. Аргумент "Положительное число: %+,d\n",k. Положительное значение целочисленной переменной k отображается со знаком плюс и с использованием системного разделителя тысячных разрядов. Перед числом отображается текст "Положительное число: ".

5. Аргумент "16-е значение %x соответствует 10-му числу %<d\n",m. Один и тот же аргумент (переменная m с шестнадцатеричным значением) сначала в формате шестнадцатеричного значения, а затем в десятичном формате. Шестнадцатеричное представление вставляется в текстовую фразу вместо инструкции %x, а десятичное - вместо инструкции %<d. Ссылка на аргумент, использовавшийся в предыдущем блоке форматирования, выполнена с помощью символа <. Поскольку в инструкции %x использована маленькая (строчная) буква x, при отображении шестнадцатеричного числа используются маленькие (строчные) буквы a, b, c и так далее.

6. Аргумент "10-е значение %d соответствует 16-му числу %<X\n",k. Значение целочисленной переменной k отображается в десятичном и шестнадцатеричном форматах. В инструкции %<X для шестнадцатеричного формата использована прописная литера X. В этом

случае в представлении шестнадцатеричного числа (при отображении в консольном окне) используются прописные литеры A, B, C и так далее.

7. Аргумент "8-е значение %o соответствует 10-му числу %<d\n",l. Восьмеричное значение переменной l сначала представляется в восьмеричном формате, а затем в десятичном. Восьмеричный формат определяется инструкцией %o.

8. Аргумент "Месяц: %tB\n",now. На основе переменной now со значением текущей даты/времени отображается название месяца (используются системные обозначения). Название месяца вставляется вместо инструкции %tB.

9. Аргумент "Число: %te\n",now. На основе переменной now со значением текущей даты/времени отображается число дня в текущем месяце. Число вставляется в формате без незначащего нуля (то есть если число состоит из одной цифры, одна цифра и отображается, а ноль перед ней не ставится). Число вставляется вместо инструкции %te.

10. Аргумент "День недели: %tA\n",now. На основе переменной now со значением текущей даты/времени отображается день недели (используются системные обозначения). День недели вставляется вместо инструкции %tA.

11. Аргумент "Время: %tT\n",now. На основе переменной now со значением текущей даты/времени текущее время (время, записанное в переменную now). Время вставляется вместо инструкции %tT.

📖 На заметку:

Фактически строка форматирования обрабатывается так. Выводится содержимое строки форматирования, только вместо блоков формата указывается, с учетом инструкций форматирования, значение соответствующего аргумента, переданного методу printf() для отображения.

Что касается отображения данных типа даты/время, то в этом случае для обозначения типа данных в строке формата используется инструкция из буквы t и еще одной буквы. Вторая буква определяет, какую именно информацию и как выводить. Например, B (полная инструкция tB) означает вывод имени месяца, e (полная инструкция te) означает число (без незначащего нуля), A (полная инструкция tA) означает день недели, T (полная инструкция tT) означает время и так далее. Полезными могут быть также следующие символы-инструкции: c (полные сведения о дате и времени), F (дата в формате *год-месяц-число*), D (дата в формате *месяц/число/год*), r (время по 12-часовому циклу), R (время по 24-часовому циклу без секунд), Y (год представлен четырьмя цифрами), y (последние две цифры года), m (представленный двумя цифрами месяц), d (представленный двумя цифрами день месяца), a (сокращенное название дня недели), b (сокращенное название месяца).

Работа с файлами

Здесь рассмотрим две задачи: во-первых, как данные в файл записать и, во-вторых, как данные из файла считать. **Общий рецепт состоит в том, чтобы связать поток ввода или вывода данных с соответствующим файлом**.

Для связывания потоков ввода и вывода данных с файлами используются классы `FileInputStream` и `FileOutputStream`. Эти классы используются для создания байтовых потоков ввода и записи данных в файлы. Чтобы открыть нужный файл для чтения или записи, необходимо создать объект класса `FileInputStream` (чтение из файла) и `FileOutputStream` (запись данных в файл). Аргументом конструктора класса указывается имя открываемого файла. Конструкторы обоих классов могут выбрасывать необрабатываемое исключение класса `FileNotFoundException`.

Таким образом, создание объекта соответствующего класса в случае успеха означает открытие соответствующего файла для чтения/записи. После завершения работы с файлом его необходимо закрыть. Для этого используется метод `close()`, который определен в каждом из классов `FileInputStream` и `FileOutputStream`.

Для побайтового считывания данных из файла используется метод `read()` класса `FileInputStream`. Считанный байт возвращается в виде целого числа. При считывании символа конца файла (обозначается `EOF`) методом в качестве результата возвращается значение `-1`. Для записи данных в файл используется метод `write()` класса `FileOutputStream`. Записываемый в файл байт указывается аргументом метода. Результата метод не возвращает. Методы `close()`, `read()` и `write()` могут выбрасывать необрабатываемое исключение класса `IOException`.

Во всей этой истории все просто и понятно. Плохо одно — через байтовые потоки считывать данные из файла и записывать их в файл крайне неудобно. Поэтому **на практике обычно используют символьные потоки**. Байтовые потоки удобно использовать при работе с бинарными файлами, а также файлами, хранящими изображения и звук. В листинге 12.3 приведен пример кода, в котором показано использование файловых потоков ввода и вывода на основе классов `FileInputStream` и `FileOutputStream`.

Листинг 12.3. Файловый поток ввода/вывода на основе классов `FileInputStream` и `FileOutputStream`

```
import java.io.*;
import javax.swing.*;
public class FileInputOutputDemo{
   public static void main(String[] args) throws IOException{
```

```
    // Базовая директория:
    String FilePath="D:/Books/Files/";
    // Считывание имени файла:
    FilePath+=JOptionPane.showInputDialog("Укажите имя файла");
    try{
       // Создание файлового потока ввода:
       FileInputStream fin=new FileInputStream(FilePath);
       // Создание файлового потока вывода:
       FileOutputStream fout=new FileOutputStream("D:/Books/
       Files/result.txt");
       // Считывание из файла:
       int s=fin.read();
       // Обработка считанного значения:
       while(s!=-1){
          switch(s){
             case 'T':
                s='t';
                break;
             case '.':
                fout.write('<');
                fout.write('*');
                s='>';
                break;
             }
             // Запись в файл:
          fout.write(s);
          // Считывание из файла:
          s=fin.read();}
       // Отображение диалогового окна:
       JOptionPane.showMessageDialog(null,"Изменения в
       файл внесены!"," Программа закончила
       работу",JOptionPane.INFORMATION_MESSAGE);
       // Закрытие файлов:
       fin.close();
       fout.close();
       // Обработка исключительной ситуации:
    }catch(FileNotFoundException e){
    JOptionPane.showMessageDialog(null,"Такого файла нет!","
    Произошла ошибка",JOptionPane.ERROR_MESSAGE);}
}}
```

Командами `import java.io.*` и `import javax.swing.*` подключаются пакеты для использования классов `FileInputStream` и `FileOutputStream` (классы для реализации бинарного файлового ввода/вывода) и класса `JOptionPane` (отображение диалоговых окон) соответственно. Текстовая переменная `FilePath` в главном ме-

тоде программы получает начальное значение "D:/Books/Files/" (полное имя директории, где расположен файл), к которому впоследствии будет добавлено имя файла для считывания информации. Эта операция выполняется с помощью команды FilePath+=JOptionPane.showInputDialog("Укажите имя файла"). В данном случае открывается диалоговое окно с полем ввода, в котором следует указать имя файла, расположенного в соответствующей директории. В принципе, такая попытка может закончиться неудачно, если, например, пользователь введет неправильное имя файла. Возможны и другие неприятности. Поэтому прочий код помещен в try-блок. В частности, там командой FileInputStream fin=new FileInputStream(FilePath) создается объект fin для файлового потока ввода. Конструктору класса FileInputStream в качестве аргумента передается полное имя файла, который открывается для считывания информации. Файловый поток вывода реализуется через объект fout класса FileOutputStream. Объект создается командой FileOutputStream fout=new FileOutputStream("D:/Books/Files/result.txt"). Таким образом, вывод информации выполняется в файл result.txt.

На заметку:

Если при создании файлового потока вывода конструктор класса FileOutputStream вызывается с одним аргументом (имя открываемого файла), то содержимое файла будет автоматически удалено, и запись в файл начинается "с нуля". Для того чтобы текущее содержимое файла не удалялось, а новые данные дописывались в конец файла, необходимо конструктору передать второй аргумент true. Это же замечание относится к классу FileWriter, который обсуждается несколько позже.

Командой int s=fin.read() объявляется целочисленная переменная s, и в качестве значения ей присваивается результат считывания символа и преобразования считанного символа в числовой код. Если считана инструкция окончания файла, значением переменной s будет -1. В операторе цикла проверяется условие на равенство считанного значения (переменная s) значению -1.

Обработка считанного значения выполняется в рамках оператора цикла в операторе switch. Правила такие: большая буква T меняется на маленькую t путем переприсваивания значения переменной s. Точки заменяются комбинацией символов <*>. В последнем случае если считана точка, то командами fout.write('<') и fout.write('*') в файл-результат записываются последовательно два символа, а переменной s командой s='>' присваивается код символа '>' (выполняется автоматическое приведение типов). После этого командой fout.write(s) выполняется запись символа в файл-результат, а затем командой s=fin.read() из исходного файла считывается новое значение. И так продолжается, пока не будет достигнут конец файла.

Если все прошло успешно, командой `JOptionPane.showMessageDialog(null,"Изменения в файл внесены!", " Программа закончила работу", JOptionPane.INFORMATION_MESSAGE)` отображается информационное окно с сообщением о том, что в файл внесены изменения.

После того, как файлы считаны и заполнены, командами `fin.close()` и `fout.close()` закрываем файлы.

В коде обрабатывается исключительная ситуация `FileNotFoundException` (файл не найден). Если такая неприятность произошла, командой `JOptionPane.showMessageDialog(null,"Такого файла нет!", " Произошла ошибка",JOptionPane.ERROR_MESSAGE)` выводится сообщение об ошибке.

В начале выполнения программы появляется окно, представленное на рис. 12.1.

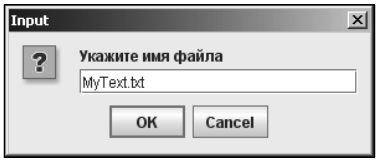


Рис. 12.1. Диалоговое окно для ввода имени файла

В поле ввода этого окна указываем имя базового файла. В данном случае файл называется `MyText.txt`. Если имя файла введено правильно, следующим появится окно, представленное на рис. 12.2.

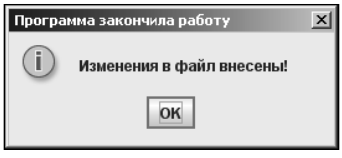


Рис. 12.2. Сообщение о том, что в файл внесены изменения

Если произошла ошибка, связанная с поиском и открытием файла (исходного `MyText.txt` или результирующего файла `result.txt`), появится окно, представленное на рис. 12.3.

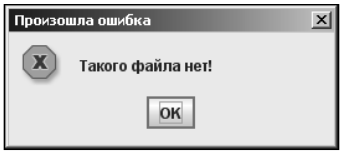


Рис. 12.3. Сообщение о том, что имя файла указано неверно

Для примера на рис. 12.4 показано окно текстового редактора с открытым в нем файлом `MyText.txt`.

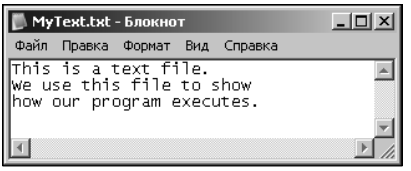


Рис. 12.4. Исходный текстовый файл

На основе этого файла заполняется файл `result.txt`, показанный на рис. 12.5, открытый в текстовом редакторе.

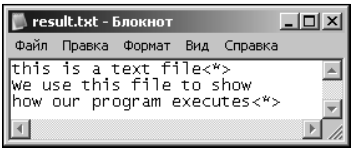


Рис. 12.5. Текстовый файл с результатом

Исходный файл `MyText.txt` содержал текст:

```
This is a text file.
We use this file to show
how our program executes.
```

В результате файл `result.txt` будет иметь следующее содержание:

```
this is a text file<*>
We use this file to show
how our program executes<*>
```

Заглавная прописная литера `T` заменена на строчную литеру `t`. Вместо каждой из двух точек появились комбинации символов `<*>`.

На заметку:

То, что в данном случае текст английский, не случайно. С кириллицей дела обстоят не так просто. В частности, для обработки кириллических символов лучше вместо классов `FileInputStream` и `FileOutputStream` использовать соответственно классы `FileReader` и `FileWriter`.

В отличие от классов `FileInputStream` и `FileOutputStream`, классами `FileReader` и `FileWriter` выполняется одностороннее преобразование файлового потока из байтового в символьный. С классами `FileReader` и `FileWriter` также используют методы `read()`, `write()` и `close()`. Тем не менее, еще более удобным представляется создание *буферизированного* потока. Буферизированный поток позволяет считывать данные не посимвольно, а построчно. Создается буферизированный поток на

основе символьного потока, который, в свою очередь, создается на основе байтового потока. Байтовые потоки можно создать с помощью классов `FileInputStream` и `FileOutputStream`. На их основе с помощью соответственно классов `InputStreamReader` и `OutputStreamWriter` создаются символьные потоки. Буферизированные потоки создаются с помощью классов `BufferedReader` и `BufferedWriter`. Пример использования этих классов приведен в листинге 12.4.

Листинг 12.4. Буферизированный файловый поток ввода/вывода

```
import java.io.*;

public class FileInOutDemo {
   public static void main(String[] args) throws IOException{
   try{
      // Создание байтового потока ввода:
      FileInputStream fis=new FileInputStream("D:/Books/Files/
      base.txt");
      // Создание байтового потока вывода:
      FileOutputStream fos=new FileOutputStream("D:/Books/
      Files/data.txt",true);
      // Создание буферизированного потока ввода:
      BufferedReader br=new BufferedReader(new InputStreamReader(
      fis,"windows-1251"));
      // Создание буферизированного потока вывода:
      BufferedWriter bw=new BufferedWriter(new OutputStreamWriter(fos));
      // Текстовая переменная:
      String str;
      do{
         // Считывание строки из файла:
         str=br.readLine();
         // Вывод считанной строки в консоль:
         System.out.println(str);
         // Переход к новой строке в файле результата:
         bw.newLine();
         // Запись в файл результата преобразованной строки:
         bw.write(str.toLowerCase().replace(' ','_'));
         }while(!str.equalsIgnoreCase("Омар Хайям"));
      // Закрытие потоков:
      br.close();
      bw.close();
      // Обработка ошибки:
   }catch(FileNotFoundException e){
      System.out.println("Файл не найден! "+e);}
   }}
```

В главном методе программы командой `FileInputStream fis=new FileInputStream("D:/Books/Files/base.txt")` создается файловый байтовый поток ввода на основе класса `FileInputStream`. Считывание выполняется с файла `base.txt`, размещенного в директории `D:/Books/Files`. Байтовый поток вывода создается командой `FileOutputStream fos=new FileOutputStream("D:/Books/Files/data.txt",true)` на основе класса `FileOutputStream`. В данном случае конструктору класса передан второй аргумент `true`. Это означает, что файл `data.txt`, переданный первым аргументом конструктору класса (вместе с именем директории `D:/Books/Files`, где размещен файл), открывается для *дописывания* информации, а не для *перезаписи*. На основе этих потоков, через создание символьных потоков, создается буферизированный поток. Для создания буферизированного потока ввода используем команду `BufferedReader br=new BufferedReader(new InputStreamReader(fis,"windows-1251"))`. У конструктора класса `BufferedReader()`, которым создается буферизированный поток ввода, передается один аргумент - анонимный объект класса `InputStreamReader`. Класс используется для символьного потока ввода. Аргументом конструктору класса `InputStreamReader` передается объект байтового файлового потока ввода, а также второй необязательный аргумент `"windows-1251"`, определяющий используемую для преобразования кодировку - в данном случае предполагается работать с русским текстом.

На заметку:

Кроме кодировки `"windows-1251"`, можно, например, использовать кодировки `"CP866"`, `"UTF-8"` или `"KOI8-R"`.

Буферизированный поток вывода создается командой `BufferedWriter bw=new BufferedWriter(new OutputStreamWriter(fos))`. Здесь на основе объекта `fos` байтового файлового потока вывода создается анонимный объект класса `OutputStreamWriter`, что соответствует созданию символьного потока вывода. Этот символьный поток служит базой для создания буферизированного потока вывода. В последнем случае используется класс `BufferedWriter`. На этом создание потоков закончено. Начинается их использование.

Объявляем текстовую переменную `str` и запускаем оператор цикла `do-while`. В этом операторе командой `str=br.readLine()` выполняется считывание текстовой строки из файла, с которым связана переменная `br` буферизированного потока, а результат записывается в переменную `str`.

На заметку:

Для считывания строки используем метод `readLine()`.

Считанную строку командой `System.out.println(str)` выводим в консольное окно. Эту строку будем записывать (с некоторыми преобразованиями) в файл результата - файл, на который "завязан" поток вывода. В этом файле выполняем переход к новой строке с помощью команды `bw.newLine()`.

На заметку:

Для перехода к новой строке используем метод `newLine()`.

Непосредственно запись в файл результата (файл `data.txt`) выполняем командой `bw.write(str.toLowerCase().replace(' ','_'))`.

На заметку:

Для записи строки через буферизированный поток используем метод `write()`.

Аргументом методу `write()` передается текстовая строка, предназначенная для записи через буферизированный поток вывода. В данном случае из строки `str` вызывается метод `toLowerCase()`, а затем из строки-результата вызывается метод `replace()`. Методом `toLowerCase()` возвращается новая строка, в которой выполняется перевод всех символов строки `str` в нижний регистр (при этом строка `str` не меняется). Методом `replace()` возвращается текстовая строка, которая получается из строки символа заменой **символа-первого аргумента метода** на **символ-второй аргумент метода**. В данном случае пробелы меняются символами подчеркивания.

Оператор цикла выполняется до тех пор, пока не будет считана строка `"Омар Хайям"` (проверяется условие `!str.equalsIgnoreCase("Омар Хайям")`). После завершения оператора цикла командами `br.close()` и `bw.close()` выполняется закрытие потоков ввода и вывода. В программе также выполняется обработка ошибки `FileNotFoundException`. Если такая ошибка возникает, то выполняется команда `System.out.println("Файл не найден! "+e)`.

Результат выполнения программы существенно зависит от содержимого файла `base.txt`. Если в этом файле нет фразы `"Омар Хайям"`, будут проблемы. К счастью, в файле `base.txt`, который для выполнения программы используем мы, такая фраза есть. Окно текстового редактора с содержимым файла base.txt представлено на рис. 12.6.

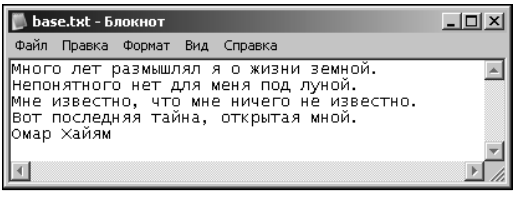


Рис. 12.6. Содержимое файла `base.txt`

Этот же текст выводится в процессе выполнения программы в консольное окно, и его там можно прочитать:

```
Много лет размышлял я о жизни земной.
Непонятного нет для меня под луной.
Мне известно, что мне ничего не известно.
Вот последняя тайна, открытая мной.
Омар Хайям
```

Этот текст берется за основу, в нем пробелы меняются символами подчеркивания, все буквы становятся строчными, и результат записывается (дописывается к уже существующему содержимому) в файл `data.txt`. Содержимое файла `data.txt` перед началом выполнения программы показано на рис. 12.7.

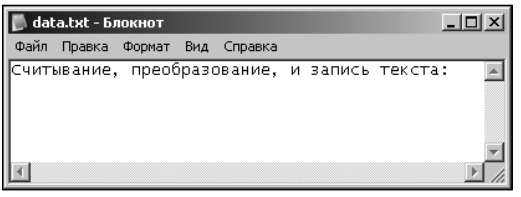


Рис. 12.7. Содержимое файла `data.txt` перед началом выполнения программы

В результате текст будет таким, как показано ниже:

```
Считывание, преобразование, и запись текста:
много_лет_размышлял_я_о_жизни_земной.
непонятного_нет_для_меня_под_луной.
мне_известно,_что_мне_ничего_не_известно.
вот_последняя_тайна,_открытая_мной.
омар_хайям
```

Для удобства жирным шрифтом выделена фраза, которая уже была в файле `data.txt` перед запуском программы. Окно текстового редактора с открытым после выполнения программы файлом `data.txt` показано на рис. 12.8.

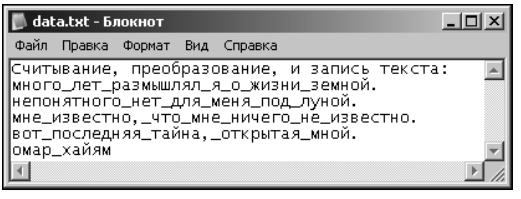


Рис. 12.8. Содержимое файла `data.txt` после выполнения программы

Отметим, что если программу запустить еще раз, новая порция текста будет добавлена к уже существующему содержимому файла `data.txt`. Чтобы при запуске программы выполнялась автоматическая очистка содержимого

файла, достаточно убрать второй логический аргумент `true` (или заменить его на `false`) в команде создания байтового потока вывода.

Резюме

1. Консольный вывод в Java может быть реализован на основе стандартного потока вывода — статического поля-объекта `out` класса `System`. Кроме стандартного потока вывода `System.out`, полезными в работе могут быть стандартный поток ввода `System.in` и поток ошибок `System.err`.
2. Один из наиболее простых способов реализации консольного ввода подразумевает создание объекта класса `Scanner`. Этот объект, в свою очередь, создается на основе потока ввода `System.in`.
3. Для форматированного консольного вывода используем метод `printf()`.
4. Неконсольный ввод/вывод может быть организован средствами класса `JOptionPane` библиотеки Swing. В этом случае ввод и вывод данных осуществляется с помощью диалоговых окон.
5. Для создания файлового ввода/вывода могут, кроме прочего, использоваться классы `FileInputStream` и `FileOutputStream`, `FileReader`, `FileWriter`, `InputStreamReader`, `OutputStreamWriter`, `BufferedReader` и `BufferedWriter`.

Глава 13
Основы библиотеки Swing

И шо я в тебя такой влюбленный?
(Из к/ф "Свадьба в Малиновке")

В этой главе речь пойдет о создании приложений с графическим интерфейсом. Хотя до этого мы уже выводили на экран диалоговые окна с различными сообщениями, утверждать, что это полноценные приложения с графическим интерфейсом, было бы несколько самонадеянно с нашей стороны.

В Java разработчику предоставляются достаточно широкие возможности по созданию полноценного, элегантного и очень функционального графического интерфейса. Средств разработки очень много, они разные и полезные. Но, памятуя о том, что объять необъятное крайне сложно, ограничимся минимально необходимым джентльменским набором. Основу нашего джентльменского набора составят средства библиотеки графических утилит Swing.

Принципы создания приложений с графическим интерфейсом

Есть две базовые библиотеки графических утилит, которые используются при создании графического интерфейса для приложений в Java. Одна – это AWT (аббревиатура от *Abstract Window Toolkit*), **и еще одна библиотека, которая называется Swing.** Исторически первой появилась библиотека AWT. Со временем ее дополнила библиотека Swing. Речь идет именно о том, что дополнила, а не заменила, хотя на сегодня библиотека Swing является, пожалуй, основной. Тем не менее, следует помнить, что основные классы библиотеки Swing созданы наследованием классов библиотеки AWT. В библиотеке Swing используется практически та же система обработки событий, что и в AWT.

Элементы графического интерфейса можно создавать как на основе утилит библиотеки AWT (так называемые *"тяжелые"* компоненты), так и на основе утилит библиотеки Swing (так называемые *"легкие"* компоненты). Что касается отличий между компонентами, созданными на основе классов библиотеки Swing (по сравнению с аналогичными компонентами, созданными на основе классов библиотеки AWT), то Swing-компоненты обладают большей гибкостью и разнообразием свойств.

На заметку:

Желающих познакомиться с чудесным миром компонентов библиотеки AWT, их отличием от компонентов библиотеки Swing, равно как историей и причинами появления последней, отсылаем к специальной литературе. В рамках этой книги останавливаться на таких познавательных подробностях нет абсолютно никакой возможности.

Для того чтобы создать приложение с графическим интерфейсом, необходимо, во-первых, создать сам интерфейс, со всеми необходимыми окнами, кнопками, полями и прочими элементами. Другими словами, необходимо определить, как элементы интерфейса приложения будут отображаться на экране. Во-вторых, следует определить реакцию функциональных элементов интерфейса приложения на действия пользователя, то есть определить способ взаимодействия пользователя с приложением через элементы графического интерфейса.

Понятно, что это задачи концептуально взаимосвязанные, поэтому решать их лучше в комплексе, хотя указанное разделение процесса создания приложения с графическим интерфейсом достаточно четко прослеживается разделение на эти два этапа.

Для создания элементов графического интерфейса, таких как окна, кнопки, переключатели, метки, полосы прокрутки, поля ввода, меню и прочее, используются специальные классы, входящие в библиотеку Swing. Каждому элементу соответствует отдельный класс. Классы библиотеки Swing размещаются в пакете `javax.swing`.

На заметку:

В этой главе мы будем иметь дело с простыми приложениями, точнее, с простым графическим интерфейсом. Более сложные варианты рассматриваются в следующей главе.

Все перечисленные элементы условно можно разделить на два типа: *компоненты* и *контейнеры*. От контейнера компонент отличается тем, что может содержать (и, как правило, содержит) другие компоненты и контейнеры. Компонент, в свою очередь, для отображения на экране должен быть включен в состав контейнера. Поэтому, по крайней мере, один контейнер при создании графического интерфейса придется использовать. Если контейнеров в программе несколько, то один всегда является контейнером *верхнего уровня*, в котором содержатся все прочие компоненты и контейнеры.

Контейнеры, в свою очередь, делятся на два типа или две категории. Одну небольшую группу формируют контейнеры верхнего уровня. Им соответствуют классы `JFrame`, `JWindow`, `JDialog` и `JApplet`. Эти классы, через систему наследования, являются потомками класса `Container` библиотеки AWT.

На заметку:

Система наследования, в той части, которая касается четырех упомянутых классов `JFrame`, `JWindow`, `JDialog` и `JApplet`, следующая. В вершине иерархии находится класс `Object`. Его наследует класс `Component`. Класс `Container` является подклассом класса `Component`. Класс `JApplet` реализуется через схему наследования `Container`▶`Pane`▶`Applet`▶`JApplet`. Три другие схемы наследования: `Container`▶`Window`▶`JWindow`, `Container`▶`Window`▶`Dialog`▶`JDialog`, и `Container`▶`Window`▶`Frame`▶`JFrame`. Классы, не содержащие начальной литеры `J` в названии, относятся к библиотеке AWT, то есть соответствуют "тяжелым" компонентам.

Эти четыре контейнера относятся к "тяжеловесным", и один из них обычно находится в вершине иерархии компонентов программы. Мы будем для этих целей пользоваться классом `JFrame`. В последней главе, где обсуждаются *апплеты*, в качестве контейнера верхнего уровня будет использоваться класс `JApplet`.

Прочие контейнеры, которые образуют многочисленную группу, относятся к "легковесным". Они, как и обычные компоненты библиотеки `Swing`, являются потомками класса `JComponent`. Принципиальное отличие легковесного контейнера от обычного легковесного компонента состоит в том, что в контейнер могут добавляться компоненты (и контейнеры). Поэтому деление на контейнеры и обычные компоненты достаточно условное. Обычно понять, относится ли компонент к контейнерам, достаточно легко исходя из общего назначения компонента. Например, такой компонент, как панель (класс `JPane`), предназначен для размещения на нем компонентов. Понятно, что это контейнер. Кнопка (класс `JButton`) вряд ли будет содержать другие графические компоненты. Наоборот, она сама должна где-то размещаться. Поэтому кнопка к контейнерам не относится.

На заметку:

Класс `JComponent` является непосредственным потомком класса `Container`.

В табл. 13.1 приведен перечень классов библиотеки Swing, используемых для создания различных компонентов (в том числе и контейнеров).

Табл. 13.1. Классы библиотеки Swing

Класс	Описание
`JApplet`	Класс предназначен для работы с апплетами
`JButton`	Класс предназначен для создания кнопок
`JCheckBox`	Класс для создания опций
`JCheckBoxMenuItem`	Класс для создания пункта меню в виде опции
`JColorChooser`	Класс для создания палитры выбора цвета

Класс	Описание
JComboBox	Класс для создания раскрывающихся списков
JComponent	Суперкласс для классов компонентов
JDesktopPane	Контейнер, используемый для создания многодокументного интерфейса
JDialog	Контейнер верхнего уровня для создания диалоговых окон
JEditorPane	Класс для создания панели редактирования с поддержкой стилей
JFileChooser	Класс для создания диалогового окна выбора файлов
JFormattedTextField	Класс для создания поля редактирования с поддержкой форматирования
JFrame	Класс для создания окна
JInternalFrame	Класс для создания внешнего окна, которое может содержать внутренние подокна
JLabel	Класс для создания меток
JLayeredPane	Класс для управления механизмом перекрытия компонентов
JList	Класс для создания списка
JMenu	Класс для создания стандартного меню
JMenuBar	Класс для создания меню верхнего уровня
JMenuItem	Класс для реализации пункта меню
JOptionPane	Класс для работы с основными типами диалоговых окон
JPanel	Класс панели
JPasswordField	Класс для создания поля ввода пароля
JPopupMenu	Класс для создания контекстного меню
JProgressBar	Класс для создания индикатора хода процесса
JRadioButton	Класс для создания переключателя опции
JRadioButtonMenuItem	Класс для создания меню в виде переключателя опций
JRootPane	Класс корневой панели
JScrollBar	Класс для создания полосы прокрутки
JScrollPane	Класс для создания панели с полосой прокрутки
JSeparator	Класс для создания разделителя в меню
JSlider	Класс для создания линейного регулятора
JSpinner	Класс для создания инкрементного регулятора (список с кнопками со стрелками, предназначенными для изменения значения в списке)

Класс	Описание
JSplitPane	Класс для создания специальной панели, состоящей из двух частей с подвижным разделителем
JTabbedPane	Класс для создания панели с вкладками
JTable	Класс для создания таблиц
JTextArea	Класс для создания текстовой области
JTextField	Класс для создания поля редактирования
JTextPane	Класс текстовой панели широкого спектра
JToggleButton	Класс для создания кнопки с двумя состояниями
JToolBar	Класс для создания панели инструментов
JToolTip	Класс предназначен для реализации оперативных подсказок для элементов интерфейса
JTree	Класс для реализации древовидной иерархической структуры данных
JViewport	Класс реализации канала просмотра
JWindow	Контейнер для реализации окон (без строки названия и системных пиктограмм)

Несложно заметить, что названия всех классов начинаются с литеры J. Для создания того или иного компонента в программе и размещения его в нужном месте необходимо фактически создать объект класса соответствующего компонента или его подкласса. Но этого мало. Необходимо также "оживить" компоненты графического интерфейса, то есть предусмотреть реакцию разных компонентов на действия пользователя. Для этих целей используется *система обработки событий*. Здесь отметим только общие принципы, а конкретную реализацию программного кода, в соответствии с этими принципами, будем изучать на примерах.

В принципе, для каждого компонента существует стандартный набор действий, на которые компонент может реагировать. В Java используется модель *обработки событий с делегированием*. Реализуется она по следующей схеме. В компонентах, с которыми что-то может происходить, регистрируются специальные *обработчики событий*. Когда в компоненте происходит событие, управление передается обработчику этого события, зарегистрированному в компоненте. Обработчик выполняет нужные действия и возвращает управление компоненту. В роли обработчика событий выступают объекты специальных классов. Эти классы создаются путем расширения интерфейсов, предназначенных для обработки того или иного события. Другими словами, для тех событий, которые могут происходить в компоненте, определены встроенные интерфейсы. Чтобы создать обработчик события, необходимо реализовать этот интерфейс в классе и создать объект

этого класса. Такой объект может использоваться в качестве обработчика события. Все это в теории. Посмотрим, как такая схема реализуется на практике. Но для начала разберем наиболее простой пример, в котором обработка событий как таковая вообще отсутствует.

Создание простого окна

Рассмотрим пример, в котором создается очень простое окно. Окно не содержит ничего, кроме строки названия с именем окна и стандартными системными пиктограммами. Для создания окна используем класс контейнера `JFrame`. Обратимся к программному коду, представленному в листинге 13.1.

Листинг 13.1. Создание простого окна

```
import javax.swing.*;
// Класс окна:
class NewFrame{
   // Конструктор класса:
   NewFrame(){
      // Создание окна - объекта класса JFrame.
      // Текстовый аргумент конструктора задает имя окна:
      JFrame MyFrame=new JFrame("Новое окно");
      // Установка размеров окна (в пикселях по ширине и высоте):
      MyFrame.setSize(300,200);
      // Реакция окна на щелчок на системной пиктограмме закрытия окна:
      MyFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      // Отображение окна:
      MyFrame.setVisible(true);
   }}
public class NewFrameDemo{
   public static void main(String[] args){
      // Анонимный объект класса NewFrame - объект окна:
      new NewFrame();}
   }
```

Программа состоит из двух классов. Интерес представляет в первую очередь класс `NewFrame`, который является классом окна, которое создается и используется в программе. Этот класс состоит из единственного конструктора (кроме конструктора в классе `NewFrame` больше нет ни методов, ни полей). Поэтому функциональность класса полностью определяется кодом его конструктора. В конструкторе первой командой `JFrame MyFrame =new JFrame("Новое окно")` создается объект `MyFrame` класса `JFrame`. Это и есть окно! Текстовый аргумент конструктора (фраза `"Новое окно"`) задает название окна. Это название отображается в полосе названий окна.

Но создать окно - мало. Необходимо выполнить настройку его параметров и отобразить окно на экране.

Для настройки параметров окна используются методы класса `JFrame()`, которые вызываются из объекта класса (объекта окна – в данном случае через объектную переменную `MyFrame`). Так, командой `MyFrame.setSize(300,200)` задаются размеры окна. Первым аргументом методу `setSize()` передается целое число, определяющее ширину окна в пикселях. Второй аргумент метода `setSize()` – высота окна в пикселях. Кроме этого, нужно предусмотреть возможность убрать окно с экрана. Дело в том, что хотя системные пиктограммы в строке названия окна по умолчанию отображаются, закрытие окна с помощью соответствующей системной пиктограммы к завершению программы не приводит. Исправить ситуацию просто. Достаточно вызывать метод `setDefaultCloseOperation()`, указав аргументом константу из класса `JFrame`. Для того чтобы работа программы при щелчке на пиктограмме завершалась, передаем аргументом константу `JFrame.EXIT_ON_CLOSE`. Вся команда выглядит так: `MyFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)`. В дальнейшем мы будем использовать такого типа команду достаточно часто (с поправкой на имя объекта). Наконец, для отображения окна на экране используем команду `MyFrame.setVisible(true)`.

На заметку:

Создание окна означает, что в памяти выделено место, в которое прописаны все параметры и характеристики этого окна, в соответствии со спецификацией класса окна. Это не означает, что окно отображается на экране. Для отображения окна его необходимо сделать видимым – вызвать из объекта окна метод `setVisible()` с аргументом `true`.

Таким образом, создание объекта класса `NewFrame` приводит к созданию и отображению на экране окна с названием "`Новое окно`" размерами `300` на `200` пикселей. Поэтому чтобы увидеть окно, в главном методе программы создаем анонимный объект класса `NewFrame` (анонимный, поскольку в дальнейшем создаваемый объект использовать не предполагается). Окно по умолчанию отображается в верхнем левом углу экрана. Оно имеет вид, как на рис. 13.1.

Рис. 13.1. Отображение простого окна

Правда, пользы от этого окна не много. Его можно перемещать по экрану, изменить размеры (мышкой переместив края рамки), свернуть, развернуть и закрыть. Поэтому разумно пример усложнить, добавив в окно что-нибудь еще. Самый простой элемент в этом отношении — текстовая метка.

Окно с текстовой меткой

Усложним предыдущий пример, добавив в окно текстовую метку. Метка представляет собой компонент, который используется исключительно для представления информации (в основном текстовой, но возможны варианты). Для создания меток в Swing есть класс `JLabel`. То есть для того, чтобы создать метку, следует, как минимум, создать объект класса `JLabel`. У класса есть несколько конструкторов. В частности, при создании объекта класса `JLabel` аргументами конструктору можно передавать:

1. Текстовую строку (объект класса `String`), которая будет отображаться меткой.
2. Изображение (объект класса, реализующего интерфейс `Icon`), которое отображается меткой.
3. Текстовая строка (объект класса `String`), изображение (объект класса, реализующего интерфейс `Icon`), которые отображаются меткой, и константа (статическое поле класса `JLabel`), определяющая способ выравнивания текста и пиктограммы в области метки.
4. Конструктору могут не передаваться аргументы. В этом случае создается пустая (не содержащая информации для отображения) метка.

Выше приведен неполный список возможных аргументов. В случае надобности мы познакомимся и с другими способами создания меток. Сейчас рассмотрим еще один простой пример. Обратимся к листингу 13.2.

Листинг 13.2. Окно с текстовой меткой

```
import javax.swing.*;
// Класс для реализации окна с меткой:
class MyFrame{
   // Конструктор класса:
      MyFrame(){
         // Создание окна:
         JFrame frm=new JFrame(" Окно с меткой");
         // Положение окна на экране и размеры:
         frm.setBounds(400,300,180,100);
         // Реакция на закрытие окна:
         frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
      // Создание текстовой метки:
      JLabel lbl=new JLabel(" Всем огромный привет!");
      // Добавление метки в окно:
      frm.add(lbl);
      // Отображение окна:
      frm.setVisible(true);}
   }
public class FrameAndLabel{
   public static void main(String[] args){
      // Создание окна (анонимный объект):
      new MyFrame();}
   }
```

В результате выполнения программы появляется окно, представленное на рис. 13.2.

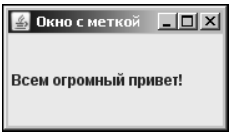


Рис. 13.2. Окно с текстовой меткой

Почему появляется именно это окно, и почему оно вообще появляется, становится ясно, если проанализировать приведенный выше программный код. В частности, основу программного кода составляет код класса `MyFrame`. Основу класса, в свою очередь, составляет конструктор класса, у которого нет аргументов и которым, по большему счету, и создается окно. В конструкторе командой `JFrame frm=new JFrame(" Окно с меткой")` создается объект `frm` класса `JFrame`, который отождествляется с окном. Строка названия этого окна содержит текст `" Окно с меткой"`, переданный аргументом конструктора класса `JFrame`. Следующими командами в конструкторе задаются свойства этого окна. В частности, командой `frm.setBounds(400,300,180,100)` задается положение окна на экране и его размеры. Для этого из объекта окна `frm` вызывается метод `setBounds()`, аргументами которому передаются: координаты (в пикселях) окна при отображении на экране и размеры (в пикселях) по горизонтали и вертикали окна.

На заметку:

При определении координат чего-нибудь в чем-нибудь используется система координат, с началом в верхнем левом углу и координатными осями: горизонтальная направлена слева направо, а вертикальная направлена сверху вниз. Например, инструкция `frm.setBounds(400,300,180,100)` означает, что окно отображается на экране на расстоянии `400` пикселей от левого края, на расстоянии `300` пикселей сверху, ширина окна `180` пикселей и его высота составляет `100` пикселей.

Реакцию на закрытие окна через системную пиктограмму определяем командой `frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)`. Метка создается командой `JLabel lbl=new JLabel(" Всем огромный привет!")`. Здесь создается объект `lbl` класса `JLabel`. Аргументом конструктору класса `JLabel` передается текстовое значение `" Всем огромный привет!"`, которое становится текстовым содержимым метки, которое меткой и отображается. Но само по себе создание метки не означает, что эта метка где-то появится. Чтобы она появилась не просто где-то, а в окне, используем команду `frm.add(lbl)`. Наконец, окно (вместе с меткой) отображается командой `frm.setVisible(true)`.

На заметку:

Практически все контейнеры (то есть объекты, которые могут содержать другие компоненты) имеют метод `add()` для добавления компонентов. Метод вызывается из того объекта, в который компонент добавляется, а аргументом методу передается добавляемый компонент (его объект).

Все, что осталось, — создать объект класса `MyFrame`. Это делаем в главном методе программы с помощью команды `new MyFrame()`, которой создаем анонимный объект класса.

Окно с текстом и пиктограммой

Усложняем (незначительно) пример, добавив в окно не просто текстовую метку, а метку с пиктограммой. Пример похож на предыдущий, но не совсем. В результате мы увидим окно, представленное на рис. 13.3.

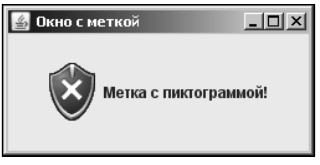


Рис. 13.3. Окно с текстом и пиктограммой

Разумеется, исследуем код, который позволяет создать такое замечательное окно. Исследовать будем код, представленный в листинге 13.3.

Листинг 13.3. Окно с текстом и пиктограммой

```
import javax.swing.*;
// Класс для реализации окна с меткой:
class MyFrame{
   // Конструктор класса.
   // Аргументы конструктора - текст для метки
   // и координаты окна на экране:
```

```
   MyFrame(String text,int x,int y){
      // Создание окна:
      JFrame frm=new JFrame(" Окно с меткой");
      // Положение окна на экране и размеры:
      frm.setBounds(x,y,250,120);
      // Реакция на закрытие окна:
      frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      // Создание пиктограммы:
   ImageIcon icn=new ImageIcon("d:/Books/Java_guide/pictures/redshd.gif");
      // Создание текстовой метки (с пиктограммой):
      JLabel lbl=new JLabel(text,icn,JLabel.CENTER);
      // Добавление метки в окно:
      frm.add(lbl);
      // Отображение окна:
      frm.setVisible(true);}
   }
public class FrameAndLabel2{
   public static void main(String[] args){
      // Создание окна (анонимный объект):
      new MyFrame(" Метка с пиктограммой!",300,400);}
   }
```

Хотя код напоминает тот, что рассматривался в предыдущем примере, но есть изменения. Сосредоточим свое внимание на них.

Во-первых, конструктор класса `MyFrame`, на основе которого реализуется окно, в данном случае принимает три аргумента: первый аргумент текстовый, а второй и третий - целочисленные. Первый аргумент — это тот текст (переменная `text` в сигнатуре конструктора), который отображается меткой. Целочисленные аргументы (переменные `x` и `y` в сигнатуре конструктора) определяют координаты отображаемого на экране окна. Но, как и ранее, все интересное происходит в конструкторе класса `MyFrame`. Объект окна создается `JFrame frm=new JFrame(" Окно с меткой")`, а положение окна на экране и его размеры задаем командой `frm.setBounds(x,y,250,120)`. Здесь в качестве первых двух аргументов метода `setBounds()` указаны аргументы, которые передавались конструктору. Знакома читателю и команда `frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)`, определяющая реакцию на закрытие окна.

Существенно новой является команда по созданию пиктограммы. Эта чудесная команда имеет такой вид: `ImageIcon icn=new ImageIcon("d:/Books/Java_guide/pictures/redshd.gif")`. Командой создается объект `icn` класса `ImageIcon`. Класс `ImageIcon` расширяет интерфейс `Icon`. Поэтому объект `icn` можно указывать там, где по идее должен быть объект класса, расширяющего интерфейс `Icon`. В частности, объект `icn` может передаваться аргументом конструктору класса `JLabel`. Пиктограм-

ма создается на основе изображения, записанного в файл `redshd.gif`. Аргументом конструктору класса `ImageIcon()` передается текстовая строка с полным путем к файлу.

После того, как создана пиктограмма для отображения меткой, создаем метку с помощью команды `JLabel lbl=new JLabel(text,icn,JLabel.CENTER)`. Создается объект `lbl` класса `JLabel()`, а аргументами конструктору класса передаются: переменная `text` (это первый текстовый аргумент класса `MyFrame`, который содержит текстовое значение, предназначенное для отображения меткой), объект пиктограммы `icn`, а также статическая константа `JLabel.CENTER`. Эта константа означает, что текст и пиктограмма в метке выравниваются по центру.

На заметку:

При размещении компонентов автоматически используется так называемый *менеджер компоновки*. В данном случае компонент всего один, и тот менеджер компоновки, который используется по умолчанию, масштабирует метку по размеру окна. Поэтому при размещении текста и пиктограммы в метке важно указать, как они выравниваются в области метки. Кроме константы `JLabel.CENTER`, могут использоваться, например, `JLabel.LEFT` и `JLabel.RIGHT`. Что касается способа отображения пиктограммы и текста в метке, то по умолчанию пиктограмма отображается слева от текста. Этот способ отображения пиктограммы и текста можно изменить.

Добавление метки в окно выполняется командой `frm.add(lbl)`. Отображение окна на экране осуществляется с помощью команды `frm.setVisible(true)`.

Окно создается в главном методе программы путем создания анонимного объекта командой `new MyFrame(" Метка с пиктограммой!",300,400)`. Первым аргументом конструктору передается текст `" Метка с пиктограммой!"`. Этот текст отображается в окне. Аргументы `300` и `400` означают, что окно (его левый верхний угол) при отображении находится на расстоянии `300` пикселей от левого края экрана и `400` пикселей от верхнего края экрана.

Окно с меткой и кнопкой

Следующий шаг переводит нас на качественно новый уровень - мы создадим окно, в котором помимо метки есть еще и кнопка. Добавление кнопки в окно настолько же банально, как и добавление метки. Небанальным в этом примере будет создание обработчика события на кнопке. Начнем с того, что опишем, как работает программа. В частности, при запуске программы открывается окно, представленное на рис. 13.4.

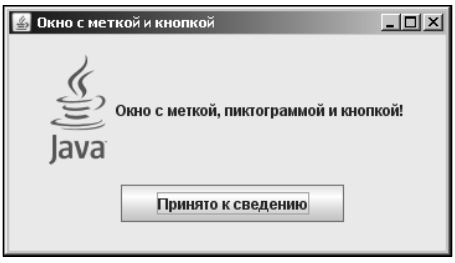


Рис. 13.4. Окно с пиктограммой и кнопкой

В центре окна находится текст и пиктограмма (реализуются в виде метки) и кнопка в нижней части окна с надписью `Принято к сведению`. Щелчок на кнопке приводит к тому, что оно закрывается, а программа завершает свою работу. Код программы представлен в листинге 13.4.

Листинг 13.4. Окно с меткой и кнопкой

```
import javax.swing.*;
import java.awt.event.*;
// Класс для реализации окна с меткой.
// Класс реализует интерфейс ActionListener:
class MyFrame implements ActionListener{
   // Конструктор класса.
   // Аргументы конструктора - текст для метки
   // и координаты окна на экране:
   MyFrame(String text,int x,int y){
      // Создание окна:
      JFrame frm=new JFrame(" Окно с меткой и кнопкой");
      // Положение окна на экране и размеры:
      frm.setBounds(x,y,360,200);
      // Реакция на закрытие окна:
      frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      // Создание пиктограммы:
ImageIcon icn=new ImageIcon("d:/Books/Java_guide/pictures/javalogo.gif");
      // Создание текстовой метки (с пиктограммой):
      JLabel lbl=new JLabel(text,icn,JLabel.CENTER);
      // Положение и размер пиктограммы:
      lbl.setBounds(10,10,330,100);
      // Создание кнопки:
      JButton btn=new JButton("Принято к сведению");
      // Положение и размер кнопки:
      btn.setBounds(90,120,180,30);
      // Регистрация обработчика в кнопке:
      btn.addActionListener(this);
      // Отключение менеджера компоновки:
      frm.setLayout(null);
      // Добавление метки в окно:
```

```
      frm.add(lbl);
      // Добавление кнопки в окно:
      frm.add(btn);
      // Отображение окна:
      frm.setVisible(true);}
   // Обработчик для кнопки.
   // Определение метода actionPerformed()
   // из интерфейса ActionListener:
   public void actionPerformed(ActionEvent AEobj){
      System.exit(0);}
   }
public class FrameAndButton{
public static void main(String[] args){
// Создание окна (анонимный объект):
new MyFrame(" Окно с меткой, пиктограммой и кнопкой!",400,250);}
}
```

Как выше отмечалось, поскольку окно содержит кнопку, эта кнопка должна как-то реагировать на щелчок на ней, в программе используется (в минимальном объеме) обработка событий. Поэтому прежде, чем приступить к описанию и анализу программного кода, кратко охарактеризуем общую ситуацию с обработкой событий в Java.

В Java используется модель обработки событий *с делегированием*, или модель *делегирования событий*. Означает она буквально следующее. Компоненты графического интерфейса могут генерировать события. Это *источники* событий. Для обработки событий используются специальные объекты, которые называются *обработчиками*. В схеме с делегированием обработчики ожидают возникновения того или иного события. Если событие возникло, обработчик получает управление и выполняет обработку события, после чего возвращает управление. Таким образом, обработка событий *делегирована* специальному коду, который отделен от конкретной реализации интерфейса приложения. Это удобно. Однако такой подход подразумевает, что обработчик, который предлагается для обработки событий компонента, должен быть *зарегистрирован* в этом компоненте. Другими словами, когда возникает какое-то событие в компоненте, за веревочки дергаются не все обработчики, а лишь те, что зарегистрированы в компоненте. Поэтому для обработки событий в приложении необходимо, по меньшей мере:

1. Создать нужные обработчики.
2. Зарегистрировать обработчики в соответствующих компонентах.

На заметку:

Существует и другой способ обработки событий, который не предполагает регистрации обработчиков в компонентах. Хотя в принципе это удобно, поскольк-

ку не нужно выполнять нетривиальную процедуру регистрации обработчиков, такой алгоритм обработки событий не очень эффективен. В этой схеме если происходит событие, то подходящий для него обработчик выбирается перебором *всех* доступных обработчиков. А на это, по крайней мере, уходит время, не считая прочих не столь очевидных неудобств.

Теперь вернемся к программному коду. Зачем командой `import javax.swing.*` подключается библиотека Swing, в общем-то понятно. Командой `import java.awt.event.*` подключается пакет event из библиотеки AWT. Этот пакет содержит утилиты для обработки событий. Несмотря на то, что мы используем библиотеку Swing для создания графического интерфейса, система обработки событий у Swing фактически та же, что и у AWT.

Класс для реализации окна с меткой называется `MyFrame`. Но поскольку в данном случае кроме графических компонентов приходится также обрабатывать события (щелчок на кнопке), создаваемый класс еще и реализует интерфейс `ActionListener`. Причина к такому радикальному и непопулярному шагу следующая. Как уже анонсировалось, в окне имеется кнопочка. Для события, которое состоит в щелчке на кнопке, хорошо было бы создать обработчик. Обработчик должен содержать метод `actionPerformed()`, который собственно и вызывается при обработке события щелчка на кнопке. Метод `actionPerformed()` объявлен в интерфейсе `ActionListener`. Обработчик является объектом класса, который реализует этот интерфейс. Мы пошли самым простым путем - интерфейс `ActionListener` реализуется тем же классом, в котором описывается окно приложения. Поэтому в этом классе, кстати говоря, описывается метод `actionPerformed()`.

Конструктор класса `MyFrame` принимает три аргумента: текстовая строка для отображения в метке в области окна и два целочисленных параметра, которые определяют положение на экране отображаемого окна. В конструкторе командой `JFrame frm=new JFrame(" Окно с меткой и кнопкой")` создается окно с заголовком `" Окно с меткой и кнопкой"`. Положение окна на экране и размеры окна задаются командой `frm.setBounds(x,y,360,200)`. Здесь `x` и `y` - второй и третий аргументы конструктора. Реакцию на закрытие окна с помощью системной пиктограммы задаем командой `frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)`.

В окне, кроме метки и кнопки, будет еще и пиктограмма (как составная часть, или содержание, метки). Создаем пиктограмму командой `ImageIcon icn=new ImageIcon("d:/Books/Java_guide/pictures/javalogo.gif")` на основе изображения в файле `javalogo.gif`. Пиктограмма создается как объект `icn` класса `ImageIcon`. После этого командой `JLabel lbl=new JLabel(text,icn,JLabel.CENTER)` создается метка

с текстом (text — первый аргумент конструктора), пиктограммой из объекта icn и способом выравнивания содержимого метки по центру (третий аргумент конструктора класса JLabel - статическая константа JLabel.CENTER). Положение и размер пиктограммы в окне задается командой lbl.setBounds(10,10,330,100). После этого приступаем к созданию кнопки. Создаем ее командой JButton btn=new JButton("Принято к сведению"). Положение и размер кнопки задаем командой btn.setBounds(90,120,180,30).

Дальше - очень важная команда, на которую следует обратить внимание: btn.addActionListener(this). Из объекта кнопки btn вызывается стандартный метод addActionListener(). Этим методом в объект, из которого он вызывается (кнопка), добавляется обработчик события щелчка на кнопке. Объект обработчика указывается аргументом метода addActionListener(). К объекту обработчика применяются достаточно демократичные требования: он должен содержать описание метода actionPerformed(). Поскольку, как увидим далее, описание этого метода есть в классе MyFrame, создаваемый объект этого класса может быть обработчиком щелчка на кнопке. Именно об этом свидетельствует ссылка this на создаваемый объект в аргументе метода addActionListener().

Чтобы компоненты (кнопка и метка) отображались в окне приложения так, как мы указали, а не в автоматическом режиме, отключаем менеджер компоновки командой frm.setLayout(null). Дальше команды простые. Так, командой frm.add(lbl) добавляется метка в окно. Добавление кнопки в окно выполняется командой frm.add(btn). Отображается окно командой frm.setVisible(true). На этом код конструктора заканчивается.

Описание метода actionPerformed() из интерфейса ActionListener, имеет следующий вид:

```
public void actionPerformed(ActionEvent AEobj){
System.exit(0);}
```

Аргументом методу передается объект класса ActionEvent. Вообще это объект события. Другими словами, методу, который обрабатывает событие, аргументом передается объект, который описывает это событие. В принципе, вполне логично. В общем случае объект события может быть очень полезным, особенно если один обработчик используется для обработки событий в нескольких компонентах. Например, по объекту события можно определить компонент, который его вызвал. Но в данном случае аргумент метода actionPerformed() в программном коде тела этого метода не используется. Тело метода состоит всего из одной команды System.exit(0). Командой завершается работа приложения (всех потоков).

Описанием метода `actionPerformed()` заканчивается описание класса `MyFrame`. После этого в методе `main()` класса `FrameAndButton` командой `new MyFrame(" Окно с меткой, пиктограммой и кнопкой!", 400,250)` создается анонимный объект класса `MyFrame`, то есть окно с описанными выше параметрами.

Классы событий

В заключение главы рассмотрим некоторые аспекты, связанные с обработкой событий. Сконцентрируем наше внимание в основном на классах и интерфейсах, которые при этом используются.

Из предыдущего примера мы уже знаем, что событие описывается объектом. Обычно объект события описывает и содержит информацию об основных параметрах, связанных с событием и вызвавшим его компонентом. Если есть объект, то должен быть и класс. Для возможных событий описана иерархия классов, в вершине которой находится класс `EventObject` из пакета `java.util`. Основная часть классов событий описана в пакете `java.awt.event`. Это система классов событий, используемых в библиотеке AWT для обработки событий. Но несколько классов событий все же описаны в библиотеке Swing — в пакете `javax.swing.event`. Некоторые классы событий из пакета `java.awt.event` перечислены в табл. 13.2.

Табл. 13.2. Классы событий пакета `java.awt.event`

Класс события	Описание
`ActionEvent`	Событие происходит при нажатии кнопки, двойном щелчке на элементе списка или выборе пункта меню
`AdjustmentEvent`	Событие происходит при выполнении операций с полосой прокрутки
`FocusEvent`	Событие происходит при получении или потере компонентом фокуса
`ItemEvent`	Событие происходит, если выбран или отмечен элемент списка или меню
`KeyEvent`	Событие генерируется при выполнении ввода с клавиатуры
`MouseEvent`	Событие связано с перемещением мыши, нажатием и отпусканием клавиш мыши, наведением курсора в область компонента и выводом его из области
`MouseWheelEvent`	Событие связано с вращением колесика мыши
`WindowEvent`	Событие происходит при активизации окна, закрытии, деактивации, сворачивании и разворачивании и прочее

Класс события	Описание
ComponentEvent	Событие происходит при изменении компонента, скрытии, отображении, изменении размера и прочее
ContainerEvent	Событие происходит при добавлении (удалении) компонента в контейнер (из контейнера)
InputEvent	Абстрактный суперкласс для классов, описывающих события, связанные с вводом для компонентов
TextEvent	Событие происходит при изменении значения текстового поля или текстовой области

Некоторые классы событий из пакета javax.swing.event (библиотека Swing) представлены в табл. 13.3.

Табл. 13.3. Классы событий пакета javax.swing.event

Класс события	Описание
AncestorEvent	Событие связано с добавлением, удалением или перемещением объекта-предка компонента
CaretEvent	Событие связано с изменением позиции курсора в текстовом компоненте
ChangeEvent	Событие происходит при изменении состояния компонента
HyperlinkEvent	Событие связано с операциями с гиперссылкой
ListDataEvent	Событие происходит при изменении содержимого списка
ListSelectionEvent	Событие происходит при выборе (отмене выбора) элемента списка
MenuEvent	Событие генерируется при выборе или отмене выбора пункта меню
TableModelEvent	Событие связано с изменением модели таблицы
TreeExpansionEvent	Событие связано с разворачиванием или сворачиванием дерева
TreeModelEvent	Событие связано с изменением модели дерева
TreeSelectionEvent	Событие происходит при выборе узла дерева

На каждое из этих событий предусмотрен специальный обработчик, который специальным образом нужно регистрировать в объекте - потенциальном источнике события (то есть в компоненте графического интерфейса). Для регистрации обработчиков применяются разные методы - в зависимости от типа обрабатываемого исключения. Если известно название для класса исключения, то с большой вероятностью можно вос-

становить имя метода, с помощью которого в компоненте регистрируется обработчик для соответствующего события. Имя метода, которым регистрируется обработчик, начинается со слова *add*. Затем следует имя события. Имя события можно узнать по имени класса, если из имени класса события убрать слово *Event*. После этого в имени метода следует слово *Listener*. Если мы обрабатываем событие класса *СобытиеEvent*, то метод для регистрации обработчика такого события в компоненте будет, скорее всего, называться *addСобытиеListener()*. Например, для регистрации обработчика события `ActionListener` мы использовали метод `addActionListener()`. Аргументом методу регистрации обработчика, как уже отмечалось, является объект обработчика (об этом несколько позже).

Регистрацию обработчика в компоненте можно отменить. Для этого также есть специальный метод. Имя метода для отмены регистрации получаем из имени метода для регистрации обработчика, если заменим *add* на *remove*. Другими словами, метод для отмены регистрации имеет структуру *removeСобытиеListener()*. Например - `removeActionListener()`. Аргументом методу передается объект обработчика (регистрация которого отменяется).

На заметку:

Методы регистрации и отмены регистрации обработчика события являются членами класса объекта компонента, в котором регистрируется обработчик или отменяется регистрация обработчика.

Осталась самая малость — разобраться с самими обработчиками событий. Обработчик должен содержать описание методов, вызываемых при обработке события. Эти методы давно определены, но не описаны. Здесь намек на то, что методы, вызываемые при обработке того или иного события, объявлены в специальных интерфейсах. Чтобы создать обработчик, необходимо реализовать соответствующий интерфейс и описать там методы обработки событий. Как правило, таких методов в интерфейсе не много - чаще всего он там один.

На заметку:

Для обработки определенного события предназначен определенный интерфейс - в обработчике *этого* события необходимо реализовать *соответствующий* интерфейс.

Чтобы узнать имя интерфейса, который нужно реализовать для обработки события, необходимо в имени класса события заменить слово *Event* на слово *Listener*.

На заметку:

В этом правиле есть небольшие исключения. Например, для события класса `MouseEvent` есть, кроме интерфейса `MouseListener`, еще и интерфейс `MouseMotionListener`. Но такие исключения погоды, как говорится, не делают.

Кроме непосредственной реализации нужного интерфейса, при создании обработчика события можно воспользоваться классами *адаптеров*. Адаптер — это встроенный класс, который реализует тот или иной интерфейс обработчика событий. Обычно в адаптерах представлены пустые реализации методов интерфейса. Класс обработчика на основе адаптера создается путем наследования адаптера как суперкласса.

На заметку:

Адаптеры предусмотрены для тех интерфейсов, которые содержат объявление больше, чем одного метода. Например, для интерфейса `ActionListener` (и соответствующего ему события `ActionEvent`) адаптера нет, поскольку в интерфейсе объявлен всего один метод `actionPerformed()`.

Хотя адаптеры принципиально ничего не меняют, их применение во многих случаях является удобным, поскольку при наследовании адаптера достаточно переопределить лишь те методы, которые используются в программном коде для обработки конкретного события.

В табл. 13.4 приведены некоторые классы-адаптеры и соответствующие им интерфейсы (то есть интерфейсы, реализованные в адаптере).

Табл. 13.4. Классы-адаптеры и соответствующие им интерфейсы

Класс-адаптер	Интерфейс, реализуемый в классе
`FocusAdapter`	`FocusListener`
`KeyAdapter`	`KeyListener`
`MouseAdapter`	`MouseListener`
`MouseMotionAdapter`	`MouseMotionListener`
`MouseInputAdapter`	`MouseListener` и `MouseMotionListener`
`WindowAdapter`	`WindowListener`
`ComponentAdapter`	`ComponentListener`
`ContainerAdapter`	`ContainerListener`

Несложно заметить, имя адаптера, как правило, получается из имени реализуемого интерфейса заменой слова *Listener* на *Adapter*.

Как все это счастье используется на практике, рассмотрим в следующей главе на конкретных примерах.

Резюме

1. Приложения с графическим интерфейсом в Java создаются на основе классов библиотеки AWT или Swing.
2. Для основных компонентов графического интерфейса существуют встроенные классы. Создание компонента подразумевает создание объекта соответствующего класса.
3. На основе существующих классов можно создавать собственные классы для компонентов с предопределенными свойствами.
4. В Java используется система обработки событий с делегированием. Для обработки того или иного события создается объект-обработчик. Объект-обработчик регистрируется в компоненте, который может вызвать событие.
5. Классы для обработчиков событий создаются путем реализации стандартных интерфейсов или наследованием специальных классов-адаптеров.

Глава 14
Приложения с графическим интерфейсом

Вы его напрасно прелестным ругаете!
(Из к/ф "Собачье сердце")

В этой главе мы продолжим знакомство с методами создания приложений с графическим интерфейсом. В частности, рассмотрим несколько примеров более-менее реальных приложений, в которых этот самый графический интерфейс используется. При этом примеры остаются относительно простыми. Первый из примеров иллюстрирует уже рассмотренные в предыдущей главе приемы создания диалоговых окон с минимальным набором управляющих элементов — метками, кнопками, а также полями ввода. Обработка событий в этом примере используется в минимальном объеме.

Создание окна с текстовым полем ввода

Программа достаточно простая и в ней последовательно отображается несколько диалоговых окон. Первое окно содержит поле ввода, в котором пользователю предлагается указать свое имя. После того, как имя пользователь ввел и подтвердил ввод, появляется новое диалоговое окно, которое внешне достаточно похоже на предыдущее, но работает немножко иначе. В этом новом окне пользователь указывает свой возраст - целое число лет. Особенность ситуации в том, что при вводе выполняется автоматическая проверка того, что пользователь вводит именно число, а не что-то другое.

Когда и эта просьба пользователем выполнена, и он ввел свой возраст, отображается третье и последнее диалоговое окно с сообщением, в котором указано имя пользователя и его возраст.

Для реализации каждого из трех упомянутых выше окон создаются специальные классы. Для удобства и наглядности (хотя и без необходимости) ссылки на компоненты диалогового окна (в том числе и ссылка на объект окна) реализованы в виде полей соответствующего класса. Программный код примера представлен в листинге 14.1.

Листинг 14.1. Создание окна с текстовым полем ввода

```
// Подключение пакетов:
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

```
// Класс для реализации окна ввода текста:
class TextInputFrame implements ActionListener{
   // Ссылка на объект окна:
   private JFrame frame;
   // Ссылка на текстовую метку:
   JLabel enter;
   // Ссылка на текстовое поле:
   JTextField tf;
   // Ссылки на кнопки:
   JButton btYes,btNo;
   private String name;
   // Конструктор класса:
   TextInputFrame(String msg){
      // Размеры окна:
      int width=300,height=150;
      // Создание окна:
      frame=new JFrame(" Окно для ввода текста (имени)...");
      // Определение положения и установка размеров окна:
      frame.setBounds(400,300,width,height);
      // Отключение менеджера компоновки:
      frame.setLayout(null);
      // Текстовая метка для поля текстового ввода:
      enter=new JLabel(msg,JLabel.LEFT);
      // Положение и размер метки:
      enter.setBounds(10,10,width-20,height/6);
      // Текстовое поле ввода:
      tf=new JTextField(frame.getWidth()-20);
      // Положение и размер текстового поля ввода:
      tf.setBounds(10,enter.getHeight()+5,enter.getWidth(),height/6);
      // Кнопка подтверждения ввода:
      btYes=new JButton("Подтверждаю");
      // Положение и размеры кнопки:
      btYes.setBounds(10,height/2+10,(width-30)/2,height/6);
      // Отмена режима отображения рамки фокуса в кнопке:
      btYes.setFocusPainted(false);
      // Кнопка отмены (завершение работы):
      btNo=new JButton("Завершить");
      // Положение и размеры кнопки:
      btNo.setBounds((width-30)/2+18,height/2+10,(width-30)/2,height/6);
      // Отмена режима отображения рамки фокуса в кнопке:
      btNo.setFocusPainted(false);
      // Реакция на закрытие окна щелчком на системной пиктограмме:
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      // Добавление в окно текстовой метки:
      frame.add(enter);
      // Добавление в окно текстового поля:
      frame.add(tf);
      // Добавление в окно кнопки подтверждения ввода:
```

```
      frame.add(btYes);
      // Добавление в окно кнопки отмены:
      frame.add(btNo);
      // Отмена возможности изменять размеры окна:
      frame.setResizable(false);
      // Регистрация обработчика щелчка в кнопке подтверждения:
      btYes.addActionListener(this);
      // Регистрация обработчика щелчка в кнопке отмены:
      btNo.addActionListener(this);
      // Отображение окна:
      frame.setVisible(true);}
   // Обработка щелчка на кнопке:
   public void actionPerformed(ActionEvent ae){
      // Считывание текста нажатой кнопки:
      String btName=ae.getActionCommand();
      // Определение реакции в зависимости от нажатой кнопки:
      if(btName.equalsIgnoreCase(btYes.getText())){
         // Заполнение текстового поля класса:
         name=tf.getText();
         // Окно с текстовым полем убирается с экрана:
         frame.setVisible(false);
         // Отображение нового окна:
         IntInputFrame.show("Сколько Вам лет? Укажите возраст (лет):",name);}
      // Завершение работы:
      else System.exit(0);}
   // Метод для отображения окна:
   static void show(String txt){
      // Создание анонимного объекта:
      new TextInputFrame(txt);}
   }
// Класс для реализации окна для ввода целого числа:
class IntInputFrame implements ActionListener, KeyListener{
   // Ссылка на объект окна:
      private JFrame frame;
   // Ссылка на объект текстовой метки:
   private JLabel enter;
   // Ссылки на кнопки:
   private JButton btYes,btNo;
   // Ссылка на текстовое поле ввода:
   private JTextField tf;
   // Текстовое поле:
   private String name;
   // Целочисленное поле:
   private int age;
   // Конструктор класса:
   IntInputFrame(String msg,String name){
      // Значение текстового поля:
      this.name=new String(name);
```

```
// Размеры окна:
int width=300,height=150;
// Создание окна:
frame=new JFrame(" Окно для ввода числа (возраста)...");
// Определение положения и установка размеров окна:
frame.setBounds(400,300,width,height);
// Отключение менеджера компоновки:
frame.setLayout(null);
// Создание текстовой метки:
enter=new JLabel(msg,JLabel.LEFT);
// Положение и размеры метки:
enter.setBounds(10,10,width-20,height/6);
// Создание текстового поля:
tf=new JTextField(frame.getWidth()-20);
// Положение и размеры текстового поля:
tf.setBounds(10,enter.getHeight()+5,enter.getWidth(),height/6);
// Регистрация обработчика в текстовом поле:
tf.addKeyListener(this);
// Создание кнопки подтверждения ввода:
btYes=new JButton("Подтверждаю");
// Положение и размеры кнопки подтверждения ввода:
btYes.setBounds(10,height/2+10,(width-30)/2,height/6);
// Кнопка подтверждения ввода неактивна:
btYes.setEnabled(false);
// Отмена режима отображения рамки фокуса в кнопке:
btYes.setFocusPainted(false);
// Кнопка завершения работы:
btNo=new JButton("Завершить");
// Положение и размеры кнопки:
btNo.setBounds((width-30)/2+18,height/2+10,(width-30)/2,height/6);
// Отмена режима отображения рамки фокуса в кнопке:
btNo.setFocusPainted(false);
// Реакция на закрытие окна щелчком на системной пиктограмме:
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
// Добавление в окно текстовой метки:
frame.add(enter);
// Добавление в окно текстового поля:
frame.add(tf);
// Добавление в окно кнопки подтверждения ввода:
frame.add(btYes);
// Добавление в окно кнопки завершения работы:
frame.add(btNo);
// Отмена возможности изменять размеры окна:
frame.setResizable(false);
// Регистрация обработчика в кнопке подтверждения ввода:
btYes.addActionListener(this);
// Регистрация обработчика в кнопке завершения работы:
btNo.addActionListener(this);
```

```
      // Отображение окна:
      frame.setVisible(true);}
   // Обработка события отпускания нажатой клавиши:
   public void keyReleased(KeyEvent ke){
      try{
         // Попытка преобразовать текст в число:
         Integer.parseInt(tf.getText());
         // Кнопка подтверждения ввода активна:
         btYes.setEnabled(true);
         // Обработка исключения:
      }catch(Exception e){
         // Кнопка подтверждения ввода неактивна:
         btYes.setEnabled(false);}
         }
   // "Пустая" реализация методов обработки нажатия кнопок:
   public void keyPressed(KeyEvent ke){}
   public void keyTyped(KeyEvent ke){}
   // Обработка нажатия кнопки:
   public void actionPerformed(ActionEvent ae){
      // Определение текста нажатой кнопки:
      String btName=ae.getActionCommand();
      // Реакция зависит от того, какая кнопка нажата:
      if(btName.equalsIgnoreCase(btYes.getText())){
         // Целочисленному полю присваивается значение:
         age=Integer.parseInt(tf.getText());
         // Окно убирается с экрана:
         frame.setVisible(false);
         // Текст для отображения в окне:
         String msg="Добрый день, "+name+"!\n";
         msg+="Ваш возраст - "+age+" лет!";
         // Отображается новое окно:
         MessageFrame.show(msg);
      }
      // Завершение работы:
      else System.exit(0);}
   // Статический метод для отображения окна для ввода числа:
   static void show(String txt,String name){
   // Создание анонимного объекта:
   new IntInputFrame(txt,name);}
   }
// Класс для реализации окна с сообщением:
class MessageFrame implements ActionListener{
   // Ссылка на объект окна:
   private JFrame frame;
   // Ссылка на метку с изображением:
   JLabel il;
   // Ссылка на метку с текстом:
   JLabel message;
```

```
    // Ссылка на кнопку:
    JButton button;
    // Конструктор класса:
    MessageFrame(String msg){
       // Размеры окна:
       int width=350,height=150;
       // Изображение для пиктограммы метки:
       ImageIcon img=new ImageIcon("d:/Books/Files/pict.gif");
       // Создание объекта окна:
       frame=new JFrame(" Сообщение...");
       // Определение положения окна и установка его размеров:
       frame.setBounds(400,300,width,height);
       // Реакция на закрытие через системную пиктограмму:
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       // Отключение менеджера компоновки:
       frame.setLayout(null);
       // Изображение для пиктограммы окна:
       Image icon=frame.getToolkit().getImage("d:/Books/Files/logo.jpg");
       // Создание метки с изображением:
       il=new JLabel(img);
       // Считывание размеров изображения:
       int w=img.getIconWidth();
       int h=img.getIconHeight();
       // Положение и размер метки с изображением:
       il.setBounds(10,10,w,h);
       // Метка с текстом:
message=new JLabel("<html>"+msg.replace("\n","<br>")+"</html>",JLabel.LEFT);
       // Положение и размеры метки с текстом:
       message.setBounds(w+20,10,width-30-w,height/2-10);
       // Добавление метки с изображением в окно:
       frame.add(il);
       // Добавление метки с текстом в окно:
       frame.add(message);
       // Создание кнопки:
       button=new JButton("Прочитано!");
       // Положение и размер кнопки:
       button.setBounds(width/4,3*height/5,width/2,height/6);
       // Отмена отображения рамки фокуса в кнопке:
       button.setFocusPainted(false);
       // Регистрация обработчика щелчка на кнопке:
       button.addActionListener(this);
       // Добавление кнопки в окно:
       frame.add(button);
       // Отмена возможности изменения размеров окна:
       frame.setResizable(false);
       // Применение пиктограммы для окна:
       frame.setIconImage(icon);
       // Отображение окна:
```

```
      frame.setVisible(true);}
   // Обработка щелчка на кнопке:
   public void actionPerformed(ActionEvent ae){
      // Завершение работы:
      System.exit(0);}
   // Статический метод для отображения окна:
   static void show(String txt){
      // Создание анонимного объекта:
      new MessageFrame(txt);}
   }
public class UsingFramesDemo{
   public static void main(String[] args){
      // Отображение окна для ввода имени:
      TextInputFrame.show("Как Вас зовут? Введите имя:");}
   }
```

Командами `import javax.swing.*`, `import java.awt.*` и `import java.awt.event.*` выполняется подключение соответствующих пакетов.

На заметку:

Первой инструкцией подключается библиотека Swing. Второй инструкцией подключается библиотека AWT. Она нам понадобится при работе с изображениями. Последней инструкцией подключается пакет, необходимый при обработке событий.

Класс для реализации окна ввода текста называется `TextInputFrame`. Он реализует интерфейс `ActionListener` - для того, чтобы в этом же классе можно было предусмотреть выполнять обработку событий. При этом среди полей класса имеется ссылка `frame` на объект окна `JFrame`. Предполагается, что окно имеет метку, текстовое поле и две кнопки. Поэтому класс также содержит поля-ссылки: на метку с текстом `enter` (ссылка на объект класса `JLabel`), на текстовое поле `tf` (объект класса `JTextField`) и ссылки на кнопки `btYes` и `btNo` (объекты класса `JButton`). Кроме этих полей, в классе есть текстовое поле `name`, предназначенное для записи текста, вводимого пользователем. Все поля зарытые.

Конструктор класса принимает текстовый аргумент, который используется как текст метки. В теле конструктора размеры создаваемого окна задаются целочисленными переменными `width` (ширина окна) и `height` (высота окна). Окно создается командой `frame=new JFrame(" Окно для ввода текста (имени)...")`. Определение положения и установка размеров окна выполняется командой `frame.setBounds(400,300,width, height)`. Инструкцией `frame.setLayout(null)` отключаем менеджер компоновки окна. Текстовая метка для поля текстового ввода создается командой `enter=new JLabel(msg,JLabel.LEFT)`.

На заметку:

Вторым аргументом конструктору класса `JLabel` передана константа `JLabel.LEFT`. Это означает, что выравнивание текста в области метки выполняется по левому краю.

Положение и размер метки задаем с помощью команды `enter.setBounds(10,10,width-20,height/6)`. Метка занимает почти всю ширину окна - остается по 10 пикселей по краям. Высота метки составляет шестую часть высоты окна.

Текстовое поле ввода создаем командой `tf=new JTextField(frame.getWidth()-20)`. Аргументом конструктору передается инструкция `frame.getWidth()-20`. Здесь для разнообразия мы размер (ширину) окна узнали не через переменную `width`, а вызовом метода `getWidth()` объекта окна `frame`. Положение и размер текстового поля задаются командой `tf.setBounds(10,enter.getHeight()+5,enter.getWidth(), height/6)`. В команде использован вызов методов `getHeight()` и `getWidth()` из объекта метки `enter`, которые соответственно возвращают высоту и ширину метки. Высота поля такая же, как и высота метки и составляет шестую часть высоты окна.

Кнопка для подтверждения ввода текста пользователем создается командой `btYes=new JButton("Подтверждаю")`. Положение и размеры кнопки задаем командой `btYes.setBounds(10,height/2+10,(width-30)/2,height/6)`. Также в коде можно заметить команду `btYes.setFocusPainted(false)`. Данной командой отменяется режим отображения рамки в области текста кнопки при передаче кнопке фокуса. Здесь исключительно эстетический момент, зацикливаться на нем не стоит.

Кнопка для завершения работы программы создается командой `btNo=newJButton("Завершить")`. Положение и размеры кнопки задаются командой `btNo.setBounds((width-30)/2+18,height/2+10,(width-30)/2,height/6)`. Как и для кнопки подтверждения ввода текста, для кнопки завершения работы отменяем для кнопки режим отображения рамки фокуса в кнопке с помощью команды `btNo.setFocusPainted(false)`.

Предполагается, что при закрытии окна щелчком на системной пиктограмме работа программы завершается. Поэтому в программный код добавляем команду `frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)`, определяющую реакцию на щелчок системной пиктограммы закрытия окна. После этого в коде класса следует серия команд по добавлению созданных компонентов в окно. Добавление в окно текстовой метки осуществляется командой `frame.add(enter)`. Текстовое поле добавляем

командой `frame.add(tf)`, а кнопки — командами `frame.add(btYes)` и `frame.add(btNo)`.

Мы создаем окно с фиксированными размерами. То есть такое окно, размеры которого захватом и растягиванием границ изменить нельзя. Для этого в код помещена инструкция `frame.setResizable(false)`, которой отменяется возможность изменения размеров окна. Регистрация обработчика щелчка в кнопке подтверждения выполняется командой `btYes.addActionListener(this)`, а регистрация обработчика щелчка в кнопке отмены - командой `btNo.addActionListener(this)`. Обращаем внимание, что в обоих случаях объектом обработчика указывается создаваемый объект (ссылка `this`). Отображается окно командой `frame.setVisible(true)`. На этом код конструктора заканчивается.

Чтобы объект класса мог использоваться в качестве обработчика щелчка на кнопке, в классе необходимо определить метод `actionPerformed()` из интерфейса `ActionListener`. В данном случае метод описан так, что при его выполнении сначала проверяется, какая кнопка нажата. Если нажата кнопка подтверждения, то значение текстового поля запоминается, окно с экрана убирается, а взамен открывается новое окно для ввода числа. Если нажата кнопка завершения программы, окно закрывается, а работа программы завершается.

Считывание текста нажатой кнопки выполняется командой `String btName=ae.getActionCommand()`. Здесь объявляется текстовая переменная `btName`, а значение переменной определяется в результате вызова метода `getActionCommand()`, который вызывается из объекта `ae`. Объект `ae` относится к классу `ActionEvent` и определен как объект аргумента `actionPerformed()`. Объект `ae` является объектом события и содержит информацию о нем. В частности, метод `getActionCommand()`, будучи вызван из объекта события, возвращает текстовое название для параметра, который называется *команда действия*. Для кнопки по умолчанию название *команды действия* совпадает с текстом на кнопке. Поэтому в следующем условном операторе проверяется условие `btName.equalsIgnoreCase(btYes.getText())`. Здесь проверяется (без учета состояния регистра) текстовое название кнопки подтверждения ввода (инструкция `btYes.getText()`) и текст, записанный в переменную `btName`. Если это так, то есть нажата кнопка подтверждения, командой `name=tf.getText()` значение текстового поля `tf` считывается и запоминается в поле `name`.

📖 На заметку:

И для кнопки, и для поля, метод, который возвращает текст компонента (название кнопки и содержание поля соответственно), называется `getText()`.

После запоминания текста окно закрывается (убирается с экрана командой `frame.setVisible(false)`), а командой `IntInputFrame.show ("Сколько Вам лет? Укажите возраст (лет):",name)` отображается новое окно для ввода числового значения (возраста пользователя). В последнем случае использован статический метод `show()` класса `IntInputFrame`, который описывается несколько позже.

Все эти действия выполняются, если нажата кнопка подтверждения (кнопка `btYes`). Если нажата кнопка завершения работы программы (кнопка `btNo`), то выполняется команда `System.exit(0)`, которой завершается работа программы.

У класса `TextInputFrame` есть статический метод `show()`, предназначенный для отображения окна ввода текста. Аргументом методу передается в качестве аргумента текстовое значение, которое служит значением текстовой метки в создаваемом окне. В частности, в методе создается с передачей аргумента метода аргументу конструктора анонимный объект класса `TextInputFrame()`.

Класс `IntInputFrame` очень похож на класс `TextInputFrame`, но некоторые отличия все же есть. На них сосредоточим свое внимание.

В первую очередь стоит заметить, что класс `IntInputFrame` реализует не только интерфейс `ActionListener`, но еще и интерфейс `KeyListener`. В качестве закрытых полей класс содержит ссылку `frame` на объект класса `JFrame` (окно для ввода числа), ссылку `enter` на объект класса `JLabel` (метка с текстом над полем ввода числа), ссылки `btYes` и `btNo` на объекты класса `JButton` (соответственно, кнопка подтверждения и кнопка завершения работы), ссылка `tf` на объект класса `JTextField` (текстовое поле ввода). Есть еще два поля: текстовое поле `name` и целочисленное поле `age`. В поле `name` запоминается имя пользователя, а в поле `age` запоминается его возраст.

На заметку:

Общая схема такая. Сначала создается объект класса `TextInputFrame` с окном для ввода имени пользователя. Это имя записывается в поле `name` объекта. Затем первое окно убирается с экрана и создается объект класса `IntInputFrame` с окном для ввода возраста. Значение поля `name` из объекта класса `TextInputFrame` переписывается в одноименное поле класса `IntInputFrame`. Поле `age` класса `IntInputFrame` заполняется на основе значения, введенного в поле ввода второго окна. Значения полей `name` и `age` передаются в объект класса `MessageFrame`, который создается для отображения третьего, последнего окна с сообщением с именем и возрастом пользователя.

Конструктору класса IntInputFrame в качестве аргументов передаются два текстовых аргумента. Первый аргумент определяет надпись над полем ввода. Второй аргумент (его копия) записывается в поле name. Этот второй аргумент предназначен для запоминания текстового значения с именем пользователя, которое передается объекту класса IntInputFrame при создании. Помимо уже знакомых нам по коду предыдущего класса команд, стоит обратить внимание на инструкцию tf.addKeyListener(this), которой в текстовом поле регистрируется обработчик для события, связанного с нажатием клавиши на клавиатуре.

На заметку:

В конструкторе для кнопки подтверждения ввода btYes установлен режим, при котором кнопка неактивна (выделена серым цветом и недоступна для нажатия). Делается это для того, чтобы предотвратить возможную ошибку ввода, когда пользователь щелкает кнопку подтверждения при некорректном (нечисловом) значении в поле ввода. Достигается такой чудесный эффект с помощью команды btYes.setEnabled(false).

Все остальное достаточно просто. Банально создаются две кнопки, задается реакция окна на закрытие через щелчок на системной пиктограмме, банально добавляется в окно поле и метка, а также кнопки. Также отменяется режим отображения рамки фокуса вокруг текста в кнопке при передаче ей фокуса. Регистрация обработчиков в кнопках выполняется командами btYes.addActionListener(this) и btNo.addActionListener(this).

На заметку:

В данном случае обработчики событий регистрируются в трех компонентах. Это две кнопки и текстовое поле ввода. В каждом из трех случаев в качестве обработчика событий указывается объект класса. Для кнопок обработчик регистрируется методом addActionListener(), поэтому обрабатывается событие ActionEvent и для обработки вызывается метод actionPerformed() (интерфейс ActionListener). Для поля обработчик регистрируется с помощью метода addKeyListener() из интерфейса KeyListener. В этом интерфейсе объявлены методы keyReleased() (вызывается при отпускании клавиши на клавиатуре), keyPressed() (вызывается при нажимании клавиши на клавиатуре) и keyTyped() (вызывается как реакция на ввод символа с клавиатуры). Эти методы необходимо описать в классе IntInputFrame, поскольку объект этого класса регистрируется как обработчик события в текстовом поле. Обрабатываемое событие относится к классу KeyEvent.

При отпускании клавиши на клавиатуре вызывается метод keyReleased(). Аргументом метода при описании указывается объект события класса KeyEvent.

На заметку:

Метод `keyReleased()` вызывается каждый раз, когда пользователь отпускает нажатую клавишу. Мы воспользуемся переопределением этого метода для проверки, является ли текущее содержимое текстового поля ввода числом или нет. Проверка выполняется так: пытаемся преобразовать содержимое поля в целое число. Если ошибки при этом не возникает, делаем кнопку подтверждения активной (доступной для нажатия). Если возникла ошибка, кнопка недоступна для нажатия. Все просто.

В теле метода в блоке `try` делается попытка преобразовать текст в поле ввода в число, для чего используем команду `Integer.parseInt(tf.getText())`. В этой команде инструкцией `tf.getText()` считывается значение в текстовом поле ввода `tf` и результат передается аргументом статическому методу `parseInt()` класса `Integer`.

На заметку:

Напомним, метод `parseInt()` преобразует текстовое представление целого числа в целое число.

Если при этом не возникает ошибки, то выполняется следующая команда `btYes.setEnabled(true)`, благодаря чему кнопка `btYes` становится активной (доступной для нажатия). Если при попытке преобразования текста в число включается обработка исключения в catch-блоке, в этом блоке выполняется команда `btYes.setEnabled(false)`, и кнопка `btYes` становится неактивной (недоступной для нажатия).

На заметку:

В `catch`-блоке аргументом указан объект класса `Exception`. Это суперкласс для классов обрабатываемых исключений. Поэтому перехватываться в блоке будут все исключения, какие только можно перехватить.

Два других метода `keyPressed()` и `keyTyped()` нам для работы не нужны, но их все равно нужно описать. Здесь мы используем "пустое" описание: после сигнатуры каждого из методов указана пустая пара фигурных скобок.

На заметку:

В данном случае становится очевидным преимущество использования классов-адаптеров по сравнению с использованием интерфейсов. Хотя нам нужен только один метод интерфейса `KeyListener`, описывать приходится все методы этого интерфейса. В классе-адаптере методы интерфейса имеют пустую реализацию. При наследовании класса-адаптера достаточно переопределить только явно используемые методы.

Обработка нажатия кнопок выполняется методом `actionPerformed()`. В методе сначала определяется, какая кнопка нажата. Если это кнопка

завершения работы, работа программы завершается (команда `System.exit(0)`). Для обработки нажатия кнопки подтверждения последовательность действий такая. Командой `age=Integer.parseInt(tf.getText())` в целочисленное поле `age` записывается результат преобразования в число текстового значения поля `tf`. Затем командой `frame.setVisible(false)` окно убирается с экрана. После этого двумя командами `String msg="Добрый день, "+name+"!\n"` и `msg+="Ваш возраст - "+age+" лет!"` в текстовую переменную `msg` записывается текстовое сообщение, предназначенное для отображения в следующем окне. Это следующее окно отображается командой `MessageFrame.show(msg)`. Здесь для отображения третьего окна использован статический метод `show()` класса `MessageFrame`. Методу в качестве аргумента передается текстовая переменная `msg`. Класс `MessageFrame` и его методы (в том числе и `show()`) описываются далее. У класса `IntInputFrame` также есть статический метод `show()`. Он принимает два аргумента, которые передаются аргументами конструктору класса `IntInputFrame` при создании анонимного экземпляра этого класса в теле метода.

Класс `MessageFrame` для реализации окна с сообщением реализует интерфейс `ActionListener`. Он содержит две метки (одна с изображением, другая с текстом) и одну кнопку. У класса есть такие закрытые поля: ссылка `frame` класса `JFrame` (объект окна сообщения), ссылки `il` и `message` класса `JLabel` (метки с изображением и метка с текстом), ссылка `button` класса `JButton` (объект кнопки).

У конструктора класса один текстовый аргумент. Записанный в нем текст предназначен для отображения в окне. В частности на основе аргумента конструктора формируется текстовое значение одной из меток. В частности, текстовая метка создается командой `message=new JLabel("<html>"+msg.replace("\n","<br>")+"</html>",JLabel.LEFT)`. Здесь конструктору первым аргументом передается текстовая строка `"<html>"+msg.replace("\n","<br>")+"</html>"`. Формируется она так. В текстовом параметре `msg` (аргумент конструктора) все инструкции перехода к новой строке `"\n"` заменяются на последовательность `"<br>"` (инструкция начала новой строки в html-формате). Полученный в результате текст вставляется между тегами `<html>` и `</html>`. Это позволяет использовать инструкцию `"\n"` для начала новой строки в тексте, передаваемом для отображения в окне. Мелочь, а приятно.

В конструкторе командой `ImageIcon img=new ImageIcon("d:/Books/Files/pict.gif")` создается объект изображения. В команде `il=new JLabel(img)` этот объект использован для создания метки с графическим изображением в качестве значения. Это изображение поя-

вится в главной панели окна. Командами `int w=img.getIconWidth()` и `int h=img.getIconHeight()` выполняется считывание размеров изображения. Положение и размер метки с изображением определяется командой `il.setBounds(10,10,w,h)` с учетом считанных ранее размеров изображения.

Еще одно изображение, которое отображается в строке названия окна (в качестве пиктограммы), создается командой `Image icon=frame.getToolkit().getImage("d:/Books/Files/logo.jpg")`. Здесь для создания объекта пиктограммы вызывается метод `getImage()`. Метод вызывается через объект *инструментария* для окна. Доступ к последнему получаем путем вызова метода `getToolkit()` из объекта окна `frame`. Применение пиктограммы для окна выполняется командой `frame.setIconImage(icon)`.

Обработка щелчка на кнопке (метод `actionPerformed()`) состоит в завершении программы. Еще в классе `MessageFrame` есть статический метод `show()`. Этот метод мы уже упоминали выше при обсуждении кода предыдущего класса. Метод принимает текстовый аргумент. Аргумент передается конструктору класса `MessageFrame` при создании анонимного объекта класса.

В классе `UsingFramesDemo` в методе `main()` командой `TextInputFrame.show("Как Вас зовут? Введите имя:")` отображается окно с полем для ввода имени пользователя. Все остальное зависит от пользователя. На рис. 14.1 показано окно, которое появляется первым.

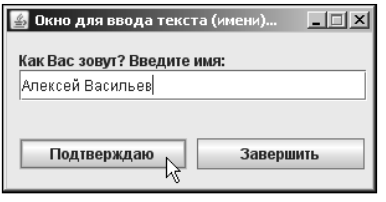


Рис. 14.1. В текстовом поле первого окна вводится имя пользователя

Если ввести (или не вводить) в поле имя (или просто текст) и щелкнуть кнопку **Подтверждаю**, появится следующее окно, показанное на рис. 14.2.

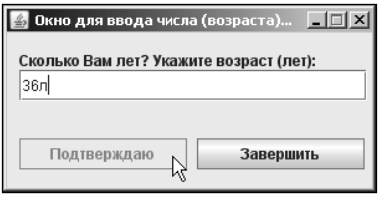


Рис. 14.2. Во втором окне при некорректном значении в поле ввода кнопка подтверждения неактивна

Особенность этого окна состоит в том, что если в текстовом поле указано нечисловое значение, то кнопка **Подтверждаю** неактивна. Именно такая ситуация показана на рис. 14.2. Но стоит только исправить значение в поле, статус кнопки меняется, как показано на рис. 14.3.

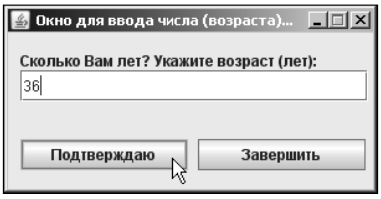


Рис. 14.3. Во втором окне в поле ввода вводится возраст пользователя

После щелчка на кнопке **Подтверждаю** появляется окно с сообщением, в котором указано имя и возраст пользователя, как на рис. 14.4.

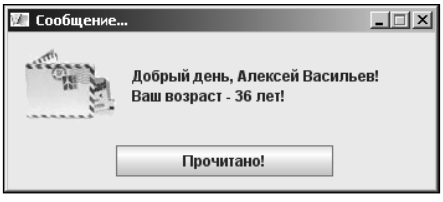


Рис. 14.4. Окно сообщения содержит информацию об имени пользователя и его возрасте

После щелчка кнопки **Прочитано** окно закрывается, а работа программы завершается.

На заметку:

Обращаем внимание, что щелчок кнопки **Завершить** любого из первых двух окон или системной пиктограммы закрытия окна любого из трех окон приводит к завершению работы программы.

Наследование классов компонентов

Не секрет, что графический интерфейс можно создавать по-разному. Один из способов подразумевает создание собственных классов как для отдельных компонентов, так и для окон. Такой способ реализации структуры классов проиллюстрируем на примере. Перед рассмотрением программного кода предварительно очертим основные моменты, связанные с функциональными возможностями создаваемого приложения.

При запуске программы отображается окно, представленное на рис. 14.5.

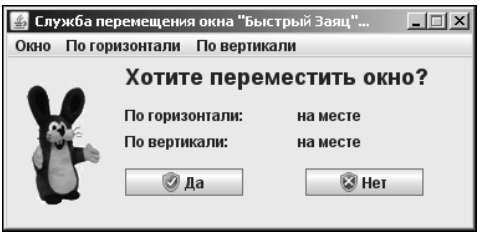


Рис. 14.5. При запуске программы отображается окно с двумя кнопками

У окна несколько функциональных элементов, а именно панель меню (состоит из трех меню **Окно**, **По горизонтали** и **По вертикали**) и две кнопки с названиями **Да** и **Нет** (обе кнопки с пиктограммами). В области окна отображается пиктограмма (с изображением зайца), статический текст **Хотите переместить окно?** (текст отображается большими буквами) и меньшими буквами отображается текст **По горизонтали** и, через пробел, **на месте**, а также в новой строке **По вертикали** и, через пробел, **на месте**. Текст в том месте, где при запуске программы отображается **на месте**, в процессе выполнения программы может меняться.

Меню **Окно** содержит два пункта: **Подтверждаю** и **Выход** (рис. 14.6).

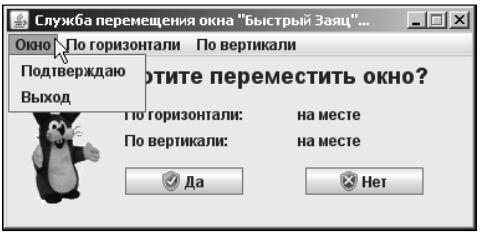


*Рис. 14.6. Содержимое меню **Окно***

В меню **По горизонтали** три пункта: **Влево**, **Вправо** и **На месте** (рис. 14.7).

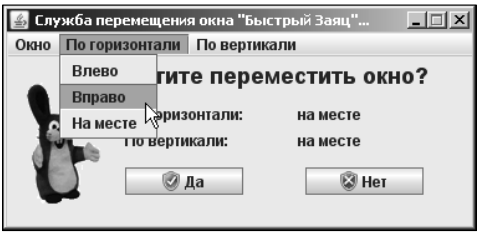


*Рис. 14.7. Содержимое меню **По горизонтали***

В меню **По вертикали** также три пункта: **Вверх**, **Вниз** и **На месте** (рис. 14.8).

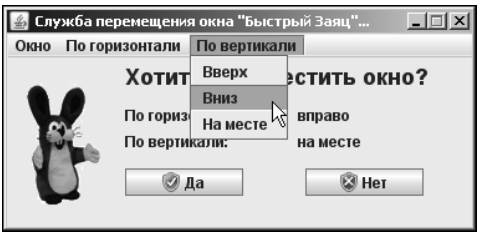


*Рис. 14.8. Содержимое меню **По вертикали***

При щелчке на кнопке **Да** или выборе пункта меню **Окно ▶ Подтверждаю** происходит перемещение окна. Окно может перемещаться, независимо по горизонтали и вертикали, на 50 пикселей влево/вправо и вверх/вниз, а также может оставаться на месте. Направление перемещения выбирается с помощью пунктов меню **По горизонтали** и **По вертикали**. При выборе одного из пунктов в области окна меняется текстовое значение для обозначения соответствующего направления перемещения. На рис. 14.9 предварительно выбраны пункты **По горизонтали ▶ Вправо** и **По вертикали ▶ Вниз**, поэтому в области окна указаны именно эти направления.

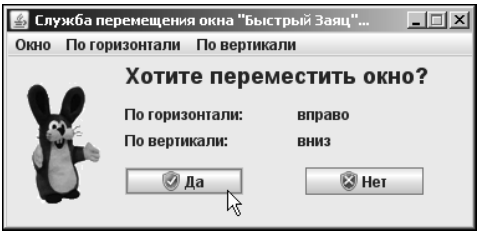


Рис. 14.9. Окно с выполненными настройками перед перемещением

После щелчка на кнопке **Да** (или выборе пункта меню **Окно ▶ Подтверждаю**) окно переместится на 50 пикселей вправо и 50 пикселей вниз. Щелчок на кнопке **Нет**, выбор пункта меню **Окно ▶ Выход** или закрытие окна с помощью системной пиктограммы приводит к завершению работы программы. Теперь обратимся к коду. Он представлен в листинге 14.2.

Листинг 14.2. Наследование классов компонентов

```
// Подключение пакетов:
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
// Класс для кнопки:
class MyButton extends JButton{
   // Ссылка на окно, в которое добавляется кнопка:
   private MyFrame frame;
   // Конструктор класса.
```

```
   // Аргументы - окно, в которое добавляется кнопка и
   // логическое значение, определяющее тип кнопки:
   MyButton(MyFrame frame,boolean state){
      // Вызов конструктора суперкласса:
      super();
      // Ссылка на окно для добавления в него кнопки:
      this.frame=frame;
      // Текст для записи имени файла изображения:
      String fileName;
      // Текст для записи имени кнопки:
      String bName;
      // Определение имени файла и названия кнопки:
      if(state){fileName="Yes.png";
           bName="Да";
           addActionListener(new HandlerYes(frame));}
      else{fileName="No.png";
      bName="Нет";
      addActionListener(new HandlerNo());}
   // Формирование полного имени файла с изображением:
   fileName="d:\\books\\files\\"+fileName;
   // Создание объекта изображения (пиктограммы):
   ImageIcon icn=new ImageIcon(fileName);
   // Применение пиктограммы для кнопки:
   setIcon(icn);
   // Применение названия для кнопки:
   setText(bName);
   // Отмена режима отображения рамки
   // вокруг текста кнопки при передаче ей фокуса:
   setFocusPainted(false);
   // Размеры кнопки:
   int w=frame.getWidth()/4;
   int h=frame.getHeight()/8;
   // Применение размеров для кнопки:
   setSize(w,h);
   // Координаты кнопки:
   int x=frame.getWidth()/4,y=frame.getHeight()-h-50;
   if(!state) x+=w+50;
   // Положение кнопки в окне:
   setLocation(x,y);
   // Добавление кнопки в окно:
   frame.add(this);}
   }
// Класс метки с изображением:
class MyIconLabel extends JLabel{
   // Ссылка на окно, в которое добавляется метка:
   private MyFrame frame;
   // Конструктор класса.
   // Аргумент - окно, в которое добавляется метка:
   MyIconLabel(MyFrame frame){
```

```
      // Вызов конструктора суперкласса:
      super();
      // Заполнение поля класса:
      this.frame=frame;
      // Создание изображения:
      ImageIcon icn=new ImageIcon("d:\\books\\files\\rabbit.gif");
      // Добавление изображения в метку:
      setIcon(icn);
      // Положение и размер метки:
      setBounds(5,15,frame.getWidth()/4-10,frame.getHeight()-30);
      // Добавление метки в окно:
      frame.add(this);}
   }
// Класс для панели меню:
class MyMenuBar extends JMenuBar{
   // Ссылка на окно, в которое добавляется панель:
   private MyFrame frame;
   // Конструктор класса.
   // Аргумент - ссылка на окно, в которое добавляется панель:
   MyMenuBar(MyFrame frame){
      // Вызов конструктора суперкласса:
      super();
      // Присваивание значения ссылке на объект окна:
      this.frame=frame;
      // Первое меню - "Окно":
      JMenu wind=new JMenu("Окно");
      // Пункты для меню "Окно":
      JMenuItem apply=new JMenuItem("Подтверждаю");
      JMenuItem exit=new JMenuItem("Выход");
      // Регистрация обработчиков событий:
      apply.addActionListener(new HandlerYes(frame));
      exit.addActionListener(new HandlerNo());
      // Добавление пунктов в меню "Окно":
      wind.add(apply);
      wind.add(exit);
      // Добавление меню "Окно" в панель меню:
      add(wind);
      // Второе меню - "По горизонтали":
      JMenu horiz=new JMenu("По горизонтали");
      // Пункты для меню "По горизонтали":
      JMenuItem left=new JMenuItem("Влево");
      JMenuItem right=new JMenuItem("Вправо");
      JMenuItem hnone=new JMenuItem("На месте");
      // Регистрация обработчиков событий:
      left.addActionListener(new MenuHandler(frame,true));
      right.addActionListener(new MenuHandler(frame,true));
      hnone.addActionListener(new MenuHandler(frame,true));
      // Добавление пунктов в меню "По горизонтали":
      horiz.add(left);
```

```
      horiz.add(right);
      horiz.add(hnone);
      // Добавление меню "По горизонтали" в панель меню:
      add(horiz);
      // Третье меню - "По вертикали":
      JMenu vert=new JMenu("По вертикали");
      // Пункты для меню "По вертикали":
      JMenuItem up=new JMenuItem("Вверх");
      JMenuItem down=new JMenuItem("Вниз");
      JMenuItem vnone=new JMenuItem("На месте");
      // Регистрация обработчиков событий:
      up.addActionListener(new MenuHandler(frame,false));
      down.addActionListener(new MenuHandler(frame,false));
      vnone.addActionListener(new MenuHandler(frame,false));
      // Добавление пунктов в меню "По вертикали":
      vert.add(up);
      vert.add(down);
      vert.add(vnone);
      // Добавление меню "По вертикали" в панель меню:
      add(vert);
      // Положение и размер панели меню:
      setBounds(1,1,frame.getWidth()-1,frame.getHeight()/10);
      // Добавление панели меню в окно:
      frame.add(this);
   }}
// Специальная панель:
class MyPanel extends JPanel{
   // Ссылки на метки с текстом,
   // указывающим направление перемещения:
   public JLabel HLab; // по горизонтали
   public JLabel VLab; // по вертикали
   // Конструктор класса:
   MyPanel(){
      // Вызов конструктора суперкласса:
      super();
      // Создание текстовых меток:
      JLabel L1=new JLabel("По горизонтали: ");
      JLabel L2=new JLabel("По вертикали: ");
      HLab=new JLabel("на месте");
      VLab=new JLabel("на месте");
      // Подключается менеджер компоновки.
      // Панель разбивается на четыре ячейки (две строки и два столбца).
      // Элементы в панели размещаются последовательно в эти ячейки.
      // Зазор между ячейками составляет 3 пикселя.
      setLayout(new GridLayout(2,2,3,3));
      // Добавление меток в панель:
      add(L1);
      add(HLab);
      add(L2);
```

```
      add(VLab);}
   }
// Класс для реализации окна:
class MyFrame extends JFrame{
   // Ссылка на панель с текстовыми метками:
   public MyPanel move;
   // Числовое поле - массив, содержащий значения
   // для шага перемещения окна по горизонтали и вертикали:
   private int[] step;
   // Метод для изменения шага перемещения по горизонтали:
   public void setH(int i){
      step[0]=i;}
   // Метод для изменения шага перемещения по вертикали:
   public void setV(int i){
      step[1]=i;}
   // Конструктор класса.
   // Аргументы - название окна и его размеры:
   MyFrame(String text,int width,int height){
      // Вызов конструктора суперкласса:
      super();
      // Начальное значение для шага перемещений:
      step=new int[]{0,0};
      // Применение заголовка:
      setTitle(text);
      // Положение и размеры окна:
      setBounds(400,300,width,height);
      // Отключение менеджера компоновки:
      setLayout(null);
      // Реакция на закрытие окна через системную пиктограмму:
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      // Создание кнопок:
      new MyButton(this,true);
      new MyButton(this,false);
      // Создание панели меню:
      new MyMenuBar(this);
      // Создание метки с изображением:
      new MyIconLabel(this);
      // Создание метки с текстом:
      JLabel msg=new JLabel("Хотите переместить окно?");
      // Положение и размеры метки с текстом:
      msg.setBounds(getWidth()/4,15,3*getWidth()/4-10,getHeight()/4-5);
      // Создание шрифта:
      Font fnt=new Font("Arial",Font.BOLD,18);
      // Применение шрифта для метки с текстом:
      msg.setFont(fnt);
      // Добавление текстовой метки в окно:
      add(msg);
      // Создание панели с текстовыми метками:
      move=new MyPanel();
```

```
      // Положение и размеры панели:
      move.setBounds(msg.getX(),msg.getY()+msg.getHeight()+1,msg.
      getWidth(),msg.getHeight());
      // Добавление панели в окно:
      add(move);
      // Установка режима, не позволяющего изменять размеры окна:
      setResizable(false);
      // Отображение окна:
      setVisible(true);}
   // Метод вызывается для перемещения окна
   // при обработке щелчка на кнопке "Да" или
   // выборе пункта меню "Подтверждаю":
   void handler(){
      // Изменение позиции окна:
      setLocation(getX()+step[0],getY()+step[1]);}
   }
// Класс обработчика для щелчка на кнопке "Нет" или выбора
// пункта меню "Выход":
class HandlerNo implements ActionListener{
   public void actionPerformed(ActionEvent ae){
      // Завершение работы программы:
   System.exit(0);}
}
// Класс обработчика для щелчка на кнопке "Да" или выбора
// пункта меню "Подтверждаю":
class HandlerYes implements ActionListener{
   // Ссылка на объект окна, в котором расположена кнопка:
   private MyFrame frame;
   // Конструктор класса:
   HandlerYes(MyFrame frame){
      // Присваивание значения ссылке:
      this.frame=frame;}
   public void actionPerformed(ActionEvent ae){
      // Вызов метода handler() для обработки события:
      frame.handler();}
   }
// Класс обработчика для выбора пунктов меню
// "По горизонтали" и "По вертикали":
class MenuHandler implements ActionListener{
   // Ссылка на объект окна, в котором размещена панель меню:
   private MyFrame frame;
   // Логическое поле, определяющее, какое из двух меню используется:
   private boolean dir;
   // Конструктор класса.
   // Аргументы - ссылка на объект окна и логический
   // параметр, определяющий одно из двух меню
   // (true - меню "По горизонтали" и false - меню "По вертикали"):
   MenuHandler(MyFrame frame,boolean dir){
      // Присваивание значения ссылке:
```

```
      this.frame=frame;
      // Присвоаивание значения логическому параметру:
      this.dir=dir;}
   // Переопределение метода для обработки выбора пункта меню:
   public void actionPerformed(ActionEvent ae){
      // Считывание команды действия:
      String cmd=ae.getActionCommand();
      // Реакция на выбор меню:
      if(dir){
         frame.move.HLab.setText(cmd.toLowerCase());
         if(cmd.equalsIgnoreCase("Вправо")) frame.setH(50);
            else if(cmd.equalsIgnoreCase("Влево")) frame.setH(-50);
                    else frame.setH(0);}
         else{
            frame.move.VLab.setText(cmd.toLowerCase());
            if(cmd.equalsIgnoreCase("Вверх")) frame.setV(-50);
            else if(cmd.equalsIgnoreCase("Вниз")) frame.setV(50);
            else frame.setV(0);}
   }}
// Класс с главным методом программы:
public class MakeComponentsDemo{
   public static void main(String[] args){
      // Создание окна:
new MyFrame(" Служба перемещения окна \"Быстрый Заяц\"...",380,180);
}}
```

Первым описывается класс `MyButton`, который наследует библиотечный класс `JButton` и представляет собой описание кнопки пользовательского типа. У этого класса есть закрытое поле `frame` класса `MyFrame`. Класс `MyFrame` описывается далее. Это класс окна пользовательского типа. Здесь поле `frame` необходимо в качестве ссылки на то окно, в которое будет добавляться кнопка. Представленный класс кнопки описывает как кнопку, с помощью которой перемещается окно (кнопка **Да**), так и кнопку, предназначенную для завершения работы программы (кнопка **Нет**). Поэтому у конструктора класса два аргумента: ссылка на окно, в которое добавляется кнопка, и аргумент логического типа, определяющий тип кнопки (значение `true` соответствует кнопке **Да**, а значение `false` соответствует кнопке **Нет**). Кроме ряда банальных команд, в конструкторе для записи имени файла изображения, которое используется для отображения в кнопке, создается текстовая переменная `fileName`. Текст с именем кнопки записывается в переменную `bName`. Заполнение этих переменных зависит от того, какая кнопка создается. Поэтому вся процедура реализуется через условный оператор, в котором проверяется значение второго, логического аргумента конструктора. В этом же условном операторе в создаваемой кнопке регистрируется обработчик щелчка на кнопке. Для кнопки **Да** регистрация выполняется командой

addActionListener(new HandlerYes(frame)), а для кнопки **Нет** обработчик регистрируется командой addActionListener(new HandlerNo()). Здесь следует сказать, что обработчики для каждой из кнопок, а также пунктов меню реализуются в виде отдельных классов (реализующих интерфейс ActionListener). Для кнопки **Да** обработчик создается как объект класса HandlerYes. У конструктора класса есть аргумент - ссылка на объект окна. В данном случае при регистрации обработчика для кнопки **Да** аргументом метода addActionListener() указывается анонимный объект класса HandlerYes. При создании этого объекта конструктору класса передается поле frame. Обработчик для кнопки **Нет** создается на основе класса HandlerNo. Конструктору этого класса аргументы передавать не нужно.

Командой fileName="d:\\books\\files\\"+fileName выполняется формирование полного имени файла с изображением для кнопки. Для создания объекта изображения для кнопки используем команду ImageIcon icn=new ImageIcon(fileName). Затем с помощью команды setIcon(icn) это изображение применяется в качестве пиктограммы для кнопки. Название кнопки задается командой setText(bName). Отмена режима отображения рамки вокруг текста кнопки при передаче ей фокуса выполняется инструкцией setFocusPainted(false). Размеры кнопки определяем как int w=frame.getWidth()/4 и int h=frame.getHeight()/8. Затем применяем их командой setSize(w,h). Координаты кнопки определяются командами int x=frame.getWidth()/4, y=frame.getHeight()-h-50 и if(!state) x+=w+50.

На заметку:

Здесь и выше для определения соответственно ширины и высоты формы frame использованы методы getWidth() и getHeight(). Таким образом, размеры и положение кнопки определяются в отношении к размерам окна, в которое помещается кнопка. При определении позиции кнопки по горизонтали сначала задается положение как для кнопки **Да**. Затем с помощью условного оператора для кнопки **Нет** положение (по горизонтали) уточняется.

Положение кнопки в окне задается командой setLocation(x,y), а добавляется кнопка в окно командой frame.add(this). Таким образом, команда добавления кнопки в окно размещена не в конструкторе окна, а в конструкторе кнопки. Собственно, это одна из причин, почему в классе MyButton в качестве поля использовалась ссылка frame на объект окна, в которое добавляется кнопка.

На заметку:

С практической точки зрения из сказанного следует, что для добавления кнопки в окно достаточно в конструкторе класса окна добавить команду создания объекта кнопки с аргументом - ссылкой на объект окна (инструкция this).

Для метки с изображением, которая размещается слева в области окна, также создается специальный класс. Он называется `MyIconLabel` и наследует класс `JLabel`. В этом классе, в силу уже отмеченных выше причин, есть закрытое поле `frame`, являющееся ссылкой на объект класса `MyFrame`. Значение этой ссылки передается аргументом конструктору класса. В конструкторе после создания изображения командой `ImageIcon icn=new ImageIcon ("d:\\books\\files\\rabbit.gif")` это изображение добавляется в метку с помощью команды `setIcon(icn)`. Положение и размер метки задается командой `setBounds(5,15,frame.getWidth()/4-10,frame.getHeight()-30)`. Как и в случае с кнопками, размер и положение метки определяются относительно размеров окна, в которое добавляется метка. А добавляется метка в окно командой `frame.add(this)`.

Класс `MyMenuBar` для *панели меню* расширяет библиотечный класс `JMenuBar`. Традиционно у класса в качестве поля определяется ссылка `frame` на объект класса `MyFrame` (окно, в которое добавляется панель меню). В конструкторе класса, кроме присваивания значения ссылке `frame`, командой `JMenu wind=new JMenu("Окно")` создается объект `wind` класса `JMenu`. Это одно из трех *меню*, которые добавляются на панель меню. Аргументом конструктору класса `JMenu` передается имя меню (в данном случае это `"Окно"`). У этого меню два *пункта меню*, которые мы создаем командами `JMenuItem apply=new JMenuItem("Подтверждаю")` и `JMenuItem exit=new JMenuItem("Выход")`. Таким образом, пункт меню - это объект класса `JMenuItem`. Как и в предыдущем случае, название пункта меню указывается аргументом конструктора класса.

Регистрация обработчиков событий выполняется командами `apply.addActionListener(new HandlerYes(frame))` и `exit.addActionListener(new HandlerNo())`.

На заметку:

Обратите внимание, что для пунктов меню **Подтверждаю** (объект `apply`) и **Выход** (объект меню `exit`) регистрируются такие же обработчики, как и для кнопок **Да** и **Нет** соответственно.

Создать меню и пункты меню - мало. Их необходимо добавить: пункты меню добавляются в меню, а меню добавляется в панель меню. Для этого используем такие команды: `wind.add(apply)` и `wind.add(exit)` (добавление пунктов меню в меню), а также `add(wind)` (добавление меню в панель меню). Аналогично создается второе и третье меню со всеми их пунктами. При регистрации обработчиков для пунктов меню используем метод `addActionListener()` с аргументом - анонимным объектом класса `MenuHandler`. Первым аргументом конструктору класса передается

объект frame окна, в которое добавляется поле меню. Для пунктов меню **По горизонтали** вторым аргументом конструктору класса MenuHandler передается логическое значение true, а для пунктов меню **По вертикали** вторым аргументом конструктору класса MenuHandler передается логическое значение false.

На заметку:

То обстоятельство, что для пунктов меню обработчик регистрируется методом addActionListener(), то есть таким же, как и для кнопки, не случайно. Дело в том, что класс JMenuItem является подклассом класса AbstractButton. Другими словами, пункт меню - это нечто близкое к кнопке.

Положение и размер панели меню определяется командой setBounds(1,1,frame.getWidth()-1,frame.getHeight()/10), после чего командой frame.add(this) панель меню добавляется в окно.

Также в программе используется специальная *панель*. Она содержит четыре текстовых метки. Область панели разбивается на четыре одинаковые ячейки, формируя таблицу размерами 2×2. Каждая такая ячейка содержит текстовую метку. Ячейки в первом столбце содержат стационарный (неизменный текст): **По горизонтали** и **По вертикали**. Текст меток во втором столбце может меняться в зависимости от выбранного пункта меню, определяющего направление перемещения окна.

Упомянутая панель описывается в классе MyPanel, который наследует класс JPanel. Класс имеет открытые поля HLab и VLab - объектные переменные класса JLabel. Это те метки, текстовые значения которых будут меняться при обработке события выбора пункта меню. Вначале (по умолчанию) текстовым значениям этих меток присваивается значение "на месте". Доступ к полям будем получать извне класса, поэтому поля открытые.

Изюминкой конструктора класса MyPanel является команда setLayout (new GridLayout(2,2,3,3)). Здесь аргументом метода подключения компоновщика setLayout() указывается анонимный объект класса GridLayout(). Командой new GridLayout(2,2,3,3) создается объект менеджера компоновки, которым область панели разбивается на ячейки, размещенные в две строки (первый аргумент конструктора класса GridLayout) и два столбца (второй аргумент конструктора класса GridLayout). Третий и четвертый аргументы конструктора класса GridLayout определяют зазоры (в пикселях) между границами ячеек. При добавлении компонентов в панель они последовательно заносятся в ячейки слева направо, сверху вниз.

Класс MyFrame для реализации окна наследует класс JFrame. У класса открытое поле move — объектная переменная класса MyLabel. Это панель с текстовыми метками. Две из них будем интенсивно менять в процессе перебора

пунктов меню. Еще у класса есть числовое поле-массив `step`. Это массив из двух элементов, значения которых определяют направление и интервал шага (в пикселях) для перемещения окна (соответственно, по горизонтали и вертикали). Поскольку это поле закрытое, то для доступа к нему описываются также открытые методы `setH()` и `setV()` для изменения значения элементов поля-массива. Элементы имеют начальные нулевые значения.

У конструктора класса такие аргументы: текстовое значение для названия окна (отображается в строке заголовка окна), а также ширина и высота окна в пикселях.

Начальное значение для шага перемещений задается командой `step=new int[]{0,0}`. Для применения заголовка используем метод `setTitle()`. Положение и размеры окна определяем с помощью метода `setBounds()`. Также командой `setLayout(null)` отключаем менеджер компоновки для окна. Реакция на закрытие окна через системную пиктограмму задается уже знакомой нам командой `setDefaultCloseOperation(EXIT_ON_CLOSE)`. Создание (и автоматическое добавление) кнопок выполняется командами `new MyButton (this,true)` и `new MyButton(this,false)`. А панель меню создается командой `new MyMenuBar(this)`. Создание метки с изображением выполняется инструкцией `new MyIconLabel(this)`.

На заметку:

Еще раз обращаем внимание читателя, что классы `MyButton`, `MyMenuBar` и `MyIconLabel` определены так, что создание объекта класса автоматически означает добавление соответствующего компонента в окно, переданное первым аргументом конструктору класса. Здесь первым аргументом указывается ключевое слово `this`, что означает добавление компонента в окно, соответствующее объекту вызова.

Но на этом мучения окна не заканчиваются. Так, командой `JLabel msg=new JLabel("Хотите переместить окно?")` создается текстовая метка. Ее положение и размеры задаются командой `msg.setBounds (getWidth()/4,15,3*getWidth()/4-10,getHeight()/4-5)`. Для метки применяется специальный шрифт. Шрифт - это объект класса `Font`.

На заметку:

Для использования класса `Font` мы командой `import java.awt.*` подключили библиотеку AWT.

Объект шрифта создается командой `Font fnt=new Font("Arial", Font.BOLD,18)`. В данном случае создается объект для шрифта типа *Arial*, *жирного*, размера *18*. Применение шрифта для метки с текстом выполняет-

ся с помощью команды `msg.setFont(fnt)`. Здесь метод `setFont()` вызывается из объекта метки `msg`, а аргументом метода указывается объект `fnt` шрифта. Добавление текстовой метки в окно выполняем с помощью команды `add(msg)`.

Панель с текстовыми метками создается командой `move=new MyPanel()`. Это как раз та панель, текстовые значения меток которой меняются при работе с панелью меню. Положение и размеры панели определяются командой `move.setBounds(msg.getX(),msg.getY()+msg.getHeight()+1,msg.getWidth(),msg.getHeight())`. Добавление панели в окно выполняется с помощью команды `add(move)`.

На заметку:

Обратите внимание, что в окне менеджер компоновки отключен. В это окно добавляется панель, для которой задан менеджер компоновки. Противоречия здесь нет. Компоненты в панели упорядочиваются менеджером компоновки этой панели. Панель является компонентом окна, поэтому при размещении панели в окне ее позиция определяется менеджером компоновки окна.

Кроме этого, для окна устанавливается режим, не позволяющий изменять размеры окна, для чего используем инструкцию `setResizable(false)`. Отображается окно командой `setVisible(true)`. Все эти чудесные действия выполняются в конструкторе класса `MyFrame`.

Кроме конструктора, в классе описан метод `handler()`, который вызывается для перемещения окна при обработке щелчка на кнопке **Да** или при обработке пункта меню **Подтверждаю**. Метод не возвращает результат, и в теле метода выполняется всего одна команда `setLocation(getX()+step[0],getY()+step[1])`. Здесь для определения текущей позиции по горизонтали и вертикали используются методы `getX()` и `getY()`. Значение шага перемещения по горизонтали и вертикали считывается из поля-массива `step`.

Класс обработчика для щелчка на кнопке **Нет** или выбора пункта меню Выход называется `HandlerNo`, реализует интерфейс `ActionListener`. В классе переопределяется метод `actionPerformed()`, в котором командой `System.exit(0)` завершается работа программы. Класс обработчика для щелчка на кнопке **Да** или выбора пункта меню **Подтверждаю** называется `HandlerYes`. Он тоже, по определению, что называется, реализует интерфейс `ActionListener`. У класса есть закрытое поле `frame` (ссылка на объект окна класса `MyFrame`), а также конструктор с одним аргументом. В конструкторе закрытому полю присваивается значение. В переопределяемом методе `actionPerformed()` для обработки щелчка на кнопке из объекта окна `frame` вызывается метод `handler()` для обработки события щелчка на кнопке или выбора пункта меню.

Более содержательным является класс `MenuHandler` обработчика для выбора пунктов меню **По горизонтали** и **По вертикали**. Класс реализует интерфейс `ActionListener`. У класса есть закрытое поле `frame` класса `MyFrame`, а также закрытое логическое поле `dir`, с помощью которого различаем меню **По горизонтали** и **По вертикали** (значение `true` соответствует меню **По горизонтали**, а значение `false` соответствует меню **По вертикали**). Все интересное происходит в методе `actionPerformed()`. Этот метод вызывается при выборе одного из пунктов меню. Сначала командой `String cmd=ae.getActionCommand()` выполняется считывание команды действия, то есть названия пункта меню, которое выбрано. Дальше с помощью системы вложенных условных операторов определяем последовательность выполняемых действий.

Так, для меню **По горизонтали** командой `frame.move.HLab.setText (cmd.toLowerCase())` текстовое значение команды действия (переменная `cmd`) переводится в нижний регистр (метод `toLowerCase()`). Результат передается аргументом методу `setText()`. Метод вызывается из объекта метки `HLab` и задает текстовое значение этой метки. Сама метка `HLab` является полем объекта панели `move`, которая, в свою очередь, является полем окна `frame`. Поэтому полная ссылка на панель выглядит как `frame.move.HLab`.

Кроме изменения текстового значения метки, необходимо также установить/изменить значение соответствующего элемента поля `step`. Здесь уместно напомнить, что элемент `step[0]` отвечает за перемещение по горизонтали. Элемент может быть изменен методом `setH()`. Элемент `step[1]` отвечает за перемещение по вертикали. Его значение изменяется методом `setV()`. Какой метод и с каким аргументом вызывать, определяем во вложенных условных операторах, в которых имя выбранного пункта меню (переменная `cmd`) проверяется на предмет совпадения с возможными вариантами. При этом для сравнения текстовых значений используется метод `equalsIgnoreCase()`, то есть сравнение выполняется без учета состояния регистра.

Аналогичная процедура выполняется для случая, когда выбран пункт меню **По горизонтали**. В классе `MakeComponentsDemo` с главным методом программы инструкцией `new MyFrame(" Служба перемещения окна \"Быстрый Заяц\"...",380,180)` создается анонимный объект класса `MyFrame`. В результате на экране появляется окно, которое мы видели в самом начале раздела (см. рис. 14.5).

Резюме

1. Чтобы научиться программировать — нужно программировать!

Глава 15
Апплеты

Нет, это не Жазель! Жазель была брюнетка. А эта — вся белая!
(Из к/ф "Формула любви")

Как правило, программа после компиляции выполняется под управлением операционной системы. Принципиально *апплет* отличается от обычной программы тем, что выполняется под управлением браузера. Апплет - небольшая программа, которая в откомпилированном виде размещается на сервере, импортируется через Internet и выполняется как составная часть веб-документа. Апплеты имеют ограниченный доступ к системным ресурсам, поэтому представляют собой относительно безопасный инструмент программирования.

Для работы с апплетами предназначен пакет `java.applet`. В этом пакете есть класс `Applet`, который находится в вершине иерархии классов, используемых при работе с апплетами. Нас будет интересовать расширение этого пакета `JApplet` из библиотеки Swing. Именно этот класс мы будем использовать как основу для создания апплетов.

Основные свойства апплетов и простой пример

Апплеты, по сравнению с Java-приложениями, имеют ряд особенностей. В первую очередь следует учесть, что апплет, для использования в штатном режиме, должен быть каким-то образом каптирован в веб-документ. В html-документах подключение апплета выполняется с помощью тега `<applet>` (открывающий тег) и `</applet>` (закрывающий тег). В открывающем теге можно указывать разные параметры, но обязательно следует указать, какой апплет подключается. Делаем это с помощью параметра `code`. Значением параметра указывается имя апплета. Параметры `width` и `height` отвечают соответственно за ширину и высоту (в пикселях) панели апплета в окне браузера. Таким образом, полная инструкция подключения апплета в `html`-документе может выглядеть следующим образом:

```
<applet code="имя_апплета" width=размер_ширина height=размер_
высота>
</applet>
```

Могут также использоваться и другие параметры.

Для использования апплета необходимо откомпилировать программу-апплет, создать html-документ с инструкцией подключения этого апплета и открыть созданный html-документ с помощью Веб-браузера. При этом следует иметь в виду, что по сравнению с прочими Java-приложениями работа с апплетами имеет ряд принципиальных особенностей. Вот они.

1. У апплетов нет метода `main()`.
2. В апплетах обычно не используется конструктор, поскольку для апплетов функциональность конструктора ограничена.
3. Поскольку апплет выполняется под управлением браузера, используемый для просмотра соответствующего html-документа браузер должен поддерживать Java-технологии.
4. Для создания апплетов используются классы `Applet` (библиотека AWT) или его более мощный и функциональный подкласс `JApplet` (библиотека Swing).

Многие свойства апплетов блокируются по соображениям безопасности. Обычно такие ограничения накладываются на апплет браузером и могут разниться для браузеров разных типов. Наиболее общие ограничения такие:

1. Апплет не может обращаться к файловой системе клиентского компьютера (компьютера, на котором запущен браузер).
2. У апплетов ограничен доступ к системным свойствам клиентского компьютера.

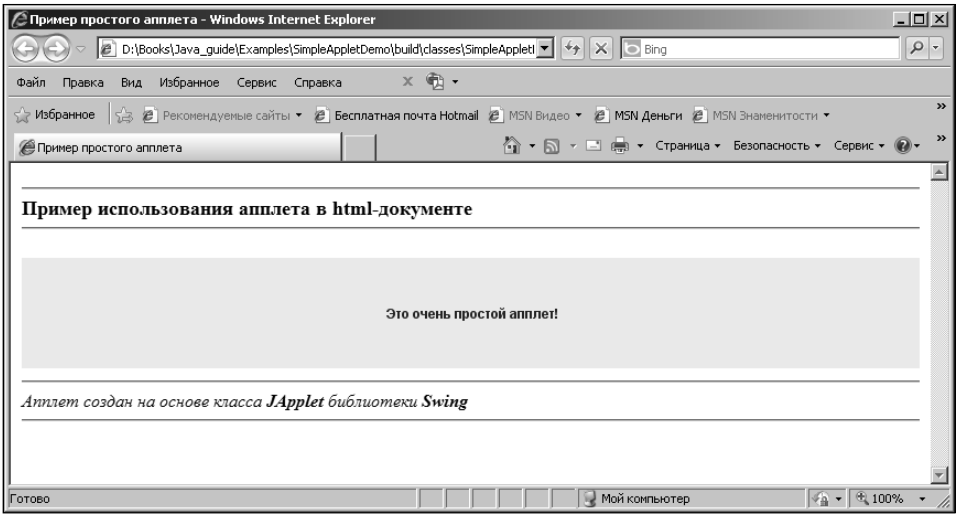


Рис. 15.1. Окно браузера Internet Explorer с html-документом, содержащим апплет

3. Апплеты не имеют доступа к буферу обмена.

4. Апплеты имеют весьма ограниченный доступ к пакетам и многое другое.

Как отмечалось выше, здесь мы будем использовать возможности библиотеки Swing и реализуем апплеты на основе класса `JApplet`. Начнем с небольшого примера. На рис. 15.1 показан html-документ (файл `SimpleAppletDemo.html`) в окне браузера.

В данном случае "работа" апплета состоит в отображении центрального серого поля с текстом **Это очень простой апплет!**. Все остальное появляется в окне браузера благодаря коду гипертекстовой разметки. Код файла `SimpleAppletDemo.html` такой (листинг 15.1).

Листинг 15.1. Код html-документа

```
<html>
<head><title>Пример простого апплета</title></head>
<body>
<h3><hr>Пример использования апплета в html-документе<hr></h3>
<applet code="SimpleAppletDemo.class" width="100%" height="100">
</applet>
<i><hr>Апплет создан на основе класса <b>JApplet</b> библиотеки
<b>Swing</b><hr></i>
</body>
</html>
```

На всякий случай, специально для читателей, не знакомых с кодами гипертекстовой разметки, прокомментируем представленный выше код.

Язык гипертекстовой разметки *HTML* (сокращение от *HyperText Markup Language*) - специальная система кодов (или инструкций), которые интерпретируются браузером при просмотре документа с гипертекстовой разметкой. Классические инструкции в html-документе называются *тегами* или *дескрипторами*. Здесь будем использовать термин *тег*. Обычно теги идут парами — есть *открывающий тег* и соответствующий ему *закрывающий тег*. Открывающий тег — это идентификатор с названием тега, заключенный в угловые скобки, то есть имею такую структуру: <*имя_тега*>. Закрывающий тег тоже заключается в угловые скобки, а его имя совпадает с именем открывающего тега, но начинается с обратной косой черты, то есть имеет структуру </*имя_тега*>. Пара тегов определяет структурные элементы html-документа. В шапке html-документа обычно отображается информация о версии HTML. Здесь ничего этого нет, поскольку нам ничего этого не нужно. Основная часть документа заключается между парой тегов `<html>` (открывающий тег) и `</html>` (закрывающий тег).

На заметку:

Имена дескрипторов могут писаться как большими, так и маленькими буквами. Здесь мы будем использовать теги, набранные в основном маленькими буквами. В открывающем теге также могут указываться значения *параметров*. Параметры указываются в открывающемся теге после имени тега внутри угловых скобок. Значения параметрам передаются так: имя параметра, знак равенства, значение параметра. Значение параметра обычно заключается в двойные кавычки. Пример передачи значений параметров в теге дает `<applet>`-блок, который обсуждается далее.

Пара тегов `<head>` и `</head>` определяет *заголовок документа*, который является контейнером для блоков, содержащих техническую (или "официальную") информацию о документе. В данном случае в заголовке документа указано только его имя, которое отображается в строке названия браузера. Имя документа задается в блоке, определенном тегами `<title>` и `</title>`.

Тело html-документа указывается между парами тегов `<body>` и `</body>`. Между этими тегами размещено основное содержимое документа. Это текст и теги. Пара тегов `<h3>` и `</h3>` создает заголовок третьего уровня. Тег `<hr>` не имеет закрывающей пары и представляет собой инструкцию отобразить горизонтальную линию.

Текст, указанный между тегами `<i>` и `</i>`, выделяется курсивом, а текст, размещенный между тегами `<b>` и `</b>`, выделяется жирным шрифтом. Но нас в первую очередь интересует та часть html-документа, которая выделена тегами `<applet>` и `</applet>`. Это то место в html-документе, куда вставляется *апплет*!

В открывающем теге `<applet>` указано три параметра. Главный из них — параметр `code`. Значением этого параметра указано имя откомпилированного файла. В данном случае исходный файл с кодом апплета называется `SimpleAppletDemo.java`. После компиляции файла получаем файл с байт-кодом `SimpleAppletDemo.class`. Именно этот файл указываем в качестве значения параметра `code` (инструкция `code="SimpleAppletDemo.class"`).

На заметку:

Файл html-документа `SimpleAppletDemo.html` и файл с байт-кодом апплета `SimpleAppletDemo.class` должны находиться в одной директории.

Кроме параметра `code`, в теге `<applet>` задаются значения параметров `width` (команда `width=100%`) и `height` (команда `height=100`). Параметром `width` задается ширина панели апплета. Значение `100%` означает, что панель апплета по ширине займет всю область окна браузера. Параметр `height` определяет высоту панели апплета. Значение `100` означает, что па-

нель апплета будет занимать по высоте 100 пикселей. С этим все просто. Осталось только определиться с тем, как выполняется апплет, а до этого - как апплет описывается и как компилируется. Начнем с программного кода апплета. Он представлен в листинге 15.2.

Листинг 15.2. Программный код простого апплета

```
import javax.swing.*;
public class SimpleAppletDemo extends JApplet{
// Метод запускается при загрузке апплета:
public void init(){
// Создание метки:
JLabel labl=new JLabel(" Это очень простой апплет!",JLabel.CENTER);
// Добавление метки в апплет:
add(labl);}
}
```

Код совсем небольшой, поэтому проанализируем его построчно. Первая команда `import javax.swing.*` нужна для того, чтобы создаваемый в программе класс апплета пользователя `SimpleAppletDemo` мог наследовать класс `JApplet` из библиотеки Swing. В классе `SimpleAppletDemo` описывается всего один метод `init()`. В этом методе командой `JLabel labl=new JLabel(" Это очень простой апплет!",JLabel.CENTER)` создается объект `labl` класса `JLabel`. Аргументом конструктору класса `JLabel` передается текстовое значение для метки и константа `JLabel.CENTER`, которая означает, что текст в метке по горизонтали выравнивается по центру метки. Добавление метки в апплет выполняется командой `add(labl)`. Во и все.

Название метода `init()` не случайно. Это один из четырех стандартных методов, которые определены в классе `JApplet` и определяют основные свойства и поведение апплета. Метод `init()` используется для инициализации апплета. Этот метод фактически играет роль конструктора. Метод автоматически вызывается при первом вызове апплета.

На заметку:

Три других замечательных представителя великолепной четверки — методы `start()`, `stop()` и `destroy()`. Метод `start()` вызывается сразу после метода `init()`, а также каждый раз при обращении к веб-странице с апплетом. Метод `stop()` вызывается автоматически при уходе со страницы с апплетом. Метод `destroy()` вызывается при завершении работы апплета.

Формально необходимости в переопределении метода `init()`, как и прочих из означенных выше методов, нет. Но если этого не сделать, то апплет никаких действий выполнять не будет и пользы от такого апплета никакой.

Таким образом, в рассмотренном выше программном коде мы определили метод `init()`, который автоматически вызывается, когда происходит обращение к апплету через браузер.

На заметку:

Обратите внимание, что в апплете нет метода `main()`. В этом методе не необходимости, поскольку исполняется апплет под управлением браузера.

Осталось теперь определиться со способом компилирования апплета и его предварительного просмотра. В принципе, если процедура компилирования выполняется через командную строку, то выглядит она достаточно стереотипно: вызывается компилятор `javac.exe`, после которого указывается имя компилируемого файла (в данном случае это файл `SimpleAppletDemo.java`). В результате создается файл `SimpleAppletDemo.class` с байт-кодом. Именно на этот файл делается ссылка в html-документе с апплетом. Как выглядит апплет, можно узнать, воспользовавшись утилитой предварительного просмотра, которая называется `appletviewer.exe` и входит в состав стандартного набора JDK. Для запуска апплета можно воспользоваться этой утилитой, указав в строке запуска после имени файла `appletviewer.exe` имя html-документа (для рассматриваемого примера это файл `SimpleAppletDemo.html`). Но это не самый лучший способ. Удобнее воспользоваться встроенными возможностями интегрированной среды разработки. В нашем случае это NetBeans.

В среде NetBeans для компиляции и предварительного просмотра апплета используем команду **Debug ▶ Debug File** или используем комбинацию клавиш <Ctrl>+<Shift>+<F5>, как показано на рис. 15.2.

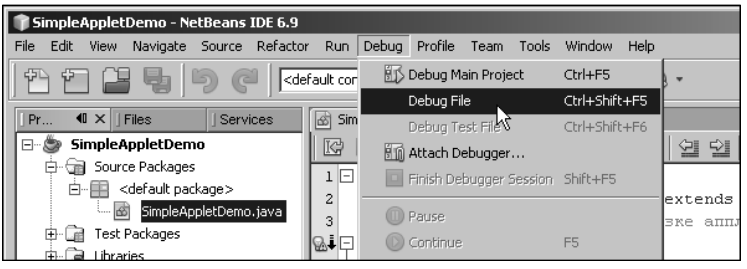


Рис. 15.2. Запуск утилиты отладки апплета в NetBeans

Если с кодом апплета все нормально, то появится окно просмотра апплетов **Applet Viewer**, содержащее апплет (рис. 15.3).

Более того, автоматически создается html-документ, в который включена инструкция вызова апплета. На рис. 15.4 показано окно браузера с html-документом, автоматически созданным средой NetBeans при отладке и просмотре апплета.

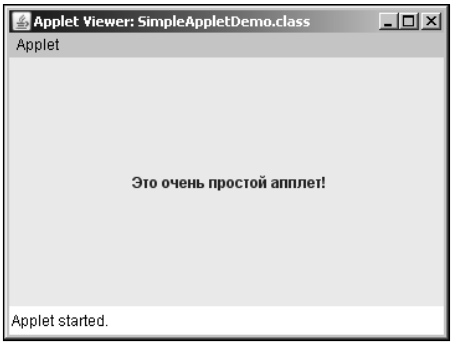


Рис. 15.3. Просмотр апплета в NetBeans

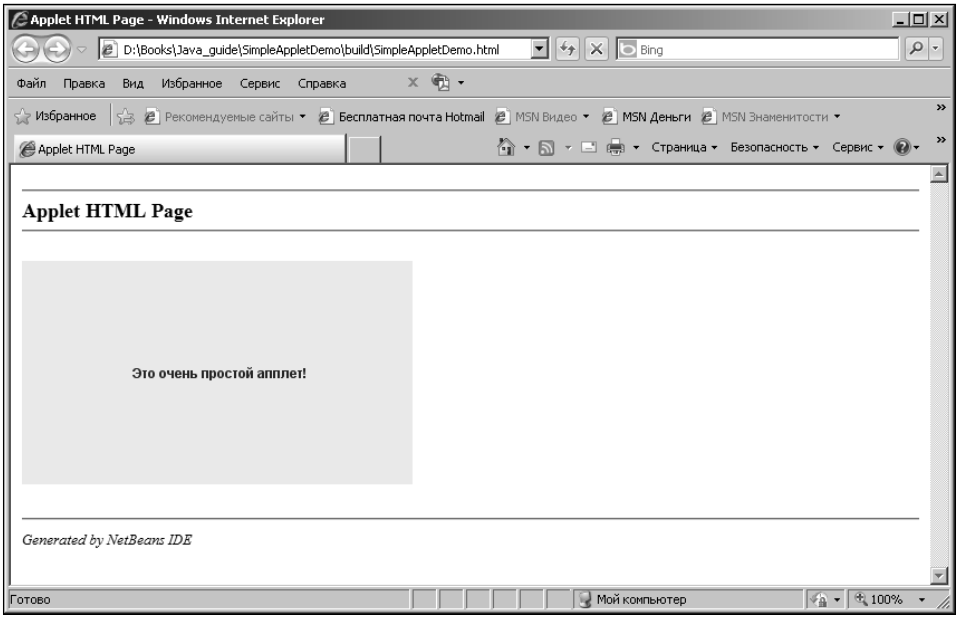


Рис. 15.4. Окно браузера с html-документом, автоматически созданным при просмотре апплета в NetBeans

В листинге 15.3 представлен код созданного средой NetBeans html-документа.

Листинг 15.3. Код автоматически созданного html-документа

```
<HTML>
<HEAD>
   <TITLE>Applet HTML Page</TITLE>
</HEAD>
<BODY>
```

```
<!--
*** GENERATED applet HTML launcher - DO NOT EDIT IN 'BUILD'
FOLDER ***
If you need to modify this HTML launcher file (e.g., to add
applet parameters), copy it to where your applet class is
found in the SRC folder. If you do this, the IDE will use it
when you run or debug the applet.
Tip: To exclude an HTML launcher from the JAR file, use
exclusion filters in the Packaging page in the Project
Properties dialog.
For more information see the online help.
-->
<H3><HR WIDTH="100%">Applet HTML Page<HR WIDTH="100%"></H3>
<P>
<APPLET codebase="classes" code="SimpleAppletDemo.class"
width=350 height=200></APPLET>
</P>
<HR WIDTH="100%"><FONT SIZE=-1><I>Generated by NetBeans IDE</
I></FONT>
</BODY>
</HTML>
```

На заметку:

По умолчанию созданный средой NetBeans html-документ находится в поддиректории `build` директории проекта. Название html-документа совпадает с именем класса апплета. Для рассматриваемого примера файл `SimpleAppletDemo.html`. Что касается представленного выше html-кода, то следует учесть, что все, находящееся между символами `<!--` и `-->`, считается комментарием и при отображении браузером игнорируется. Пара тегов `<P>` и `</P>` означает блок абзаца.

Передача апплету аргументов

Апплету через html-документ могут передаваться аргументы. Другими словами, мы в состоянии добиться ситуации, когда загружаемый апплет получает внешние входные аргументы прямо из html-документа, содержащего код загрузки апплета. Для передачи аргументов в апплете используется тег `<param>`. Этих тегов может быть несколько. Все эти теги размещаются внутри блока апплета, то есть между тегами `<applet>` и `</applet>`. Внутри этого тега указываются два параметра: `name` со значением-названием передаваемого апплету аргумента и второй параметр `value`, значением которого является значение передаваемого апплету аргумента. Вся конструкция выглядит так: `<param name="`*аргумент_апплета*`" value="`*значение_аргумента*`">`. В этом случае апплету передается аргумент с названием *аргумент_апплета*, а значение этого аргумента - *значение_аргумента*.

На заметку:

У тега <param> закрывающей пары нет.

При этом в коде апплета для считывания значения аргумента апплета используют статический метод getParameter(). Этот метод наследуется в классе JApplet из класса Applet. Аргументом методу передается текстовая строка с именем аргумента апплета, считываемого из html-документа. В качестве результата возвращается текстовая строка со значением (текстовым представлением значения) соответствующего аргумента апплета, считываемого из html-документа. Пример апплета, в котором считываются аргументы из html-документа, приведен в листинге 15.4.

Листинг 15.4. Считывание аргументов апплета из html-документа

```
import java.awt.*;
import javax.swing.*;
public class AppletWithParams extends JApplet{
public void init(){
// HTML-текст для метки (начальное значение):
String text="<html>Используется шрифт размера ";
// Размер шрифта - считывание параметра из html-документа:
int size=Integer.parseInt(getParameter("size"));
// Создание объекта шрифта:
Font fnt=new Font("Arial",Font.BOLD,size);
// Изменение текста метки:
text+=size+".<br>";
// Объектная переменная для цвета фона:
Color clr;
// Цвет фона - считывание параметра из html-документа:
String clrName=getParameter("color");
// Определение цвета:
if(clrName.equalsIgnoreCase("синий")) clr=Color.BLUE;
else if(clrName.equalsIgnoreCase("красный")) clr=Color.RED;
else if(clrName.equalsIgnoreCase("желтый")) clr=Color.YELLOW;
else if(clrName.equalsIgnoreCase("зеленый")) clr=Color.GREEN;
else clr=Color.GRAY;
// Применение цвета:
getContentPane().setBackground(clr);
// Изменение текста метки:
text+="Цвет фона: "+clrName.toLowerCase()+".</html>";
// Создание метки:
JLabel lb=new JLabel(text,JLabel.CENTER);
// Применение шрифта для метки:
lb.setFont(fnt);
// Добавление метки на панель апплета:
add(lb);}
}
```

Поскольку в данном примере предполагается работать с классами Font (для работы со шрифтом) и Color (для работы с цветом), командой import java.awt.* подключаем библиотеку AWT. Подключение командой import javax.swing.* библиотеки Swing представляется вполне оправданным, ведь класс апплета AppletWithParams мы будем создавать путем наследования класса JApplet из библиотеки Swing.

В классе всего один метод init(), который автоматически вызывается при загрузке апплета. Командой String text="<html>Используется шрифт размера " объявляется текстовая переменная text с начальным текстовым значением. Значение этой переменной, в конечном ее варианте, будет использовано в текстовой метке как содержимое. Особенность текстового значения, записанного на начальном этапе в переменную text, связана с тем, что соответствующий текст начинается с тега <html>. Наличие этой инструкции в тексте метки означает, что используется так называемый *html-текст*. В этом тексте можно использовать html-теги. Нас в первую очередь будет интересовать тег
, который означает переход к новой строке и с помощью которого текст метки можно отображать в несколько строк.

На заметку:

Инструкция <html>, как и прочие теги, в тексте метки не отображается.

В апплете считывается 2 параметра из html-аргумента: размер шрифта, который у текста метки, а также цвет фона для панели апплета. Размер шрифта записывается в целочисленную переменную size, которая объявляется и инициализируется командой int size=Integer.parseInt(getParameter("size")). В данном случае аргументом метода преобразования текста в число указана инструкция getParameter("size"). Этой инструкцией из html-документа считывается *значение* аргумента, который *называется* size.

На заметку:

Другими словами, при выполнении команды getParameter("size") в html-документе выполняется поиск среди тегов <param> такого, в котором есть инструкция name="size", а в качестве результата возвращается значение параметра value в том же теге. Это значение считывается как текст. Для преобразования текста в число используем статический метод parseInt() класса Integer.

После того, как размер шрифта считан, командой Font fnt=new Font("Arial",Font.BOLD,size) создается объект fnt класса Font. Конструктору класса при создании объекта шрифта в качестве аргументов передаются: текстовое значение семейства шрифта (в данном случае "Arial"), тип начертания (определяется статической константой Font.BOLD, что означает жирный шрифт) и размер шрифта (переменная size).

Размер шрифта отображаем в тексте метки. Поэтому командой `text+= size+"."<br>"` модифицируем текстовую переменную `text`. К текущему значению переменной добавляется (дописывается в конец) размер шрифта, точка, и добавляется html-инструкция (тег) `<br>` для выполнения перехода к новой строке.

На заметку:

Инструкция `\n` в данном случае не поможет!

Переменная `clr` для цвета фона объявляется командой `Color clr` как объектная переменная класса `Color`. В эту переменную нужно записать значение. Но для начала необходимо считать значение аргумента, который передается через html-документ в апплет и фактически, определяет цвет фона. Считывание этого аргумента (его значения) выполняется командой `String clrName=getParameter("color")`. Этой командой из html-документа считывается значение параметра, который называется `color`. Определение цвета на основе считанного значения параметра выполняется через систему вложенных условных операторов:

```
if(clrName.equalsIgnoreCase("синий")) clr=Color.BLUE;
else if(clrName.equalsIgnoreCase("красный")) clr=Color.RED;
else if(clrName.equalsIgnoreCase("желтый")) clr=Color.YELLOW;
else if(clrName.equalsIgnoreCase("зеленый")) clr=Color.GREEN;
else clr=Color.GRAY;
```

Здесь следует учесть, что значение параметра считывается как текст, поэтому обрабатывается тоже как текст. Предполагается, что фон панели апплета может быть *синим*, *красным*, *желтым*, *зеленым* и *серым*. Цвет фона в html-документе указывается кириллицей. При сравнении значений состояние регистра в расчет не принимается (при сравнении использована функция `equalsIgnoreCase()`). При наличии совпадения переменной `clr` присваивается значение - используется соответствующая одна из статических констант класса `Color`.

На заметку:

Синему цвету соответствует константа `Color.BLUE`, красному цвету соответствует константа `Color.RED`, желтому цвету соответствует константа `Color.YELLOW`, зеленому цвету соответствует константа `Color.GREEN`, серому цвету соответствует константа `Color.GRAY`.

Серый фон назначается по остаточному принципу, если нет совпадения для синего, красного, желтого и зеленого цвета.

Применяется цвет фона для панели апплета командой `getContentPane().setBackground(clr)`. Метод `setBackground()`, аргументом которому

передается объектная переменная clr, содержащая информацию о цвете, вызывается из *панели содержимого апплета*. Доступ к этой панели можно получить с помощью метода getContentPane(). В качестве результата метод возвращает ссылку на панель содержимого апплета.

Очередное изменение текста метки выполняем командой text+= "Цвет фона: "+clrName.toLowerCase()+".</html>". Это новая строка, она содержит название цвета фона апплета (текстовое значение clrName, считанное из html-документа и переведенное в нижний регистр с помощью метода toLowerCase()). Туда же добавляется финальная точка и инструкция окончания html-текста (тег </html>). Создается метка, ради которой выполнялись все эти манипуляции, с помощью команды JLabel lb=new JLabel(text,JLabel.CENTER). Благодаря второму аргументу JLabel.CENTER, текст в метке выравнивается по центру. Применение шрифта для метки осуществляется с помощью команды lb.setFont(fnt), а добавление метки в апплет - с помощью команды add(lb).

Чтобы представить, как будет выглядеть конечный результат, необходимо кратко проанализировать еще код html-документа, представленный в листинге 15.5.

Листинг 15.5. Код html-документа с передачей аргументов апплету

```
<html>
<head><title>Апплет с параметрами</title></head>
<body>
<h3><hr>Размер шрифта и цвет фона считываются апплетом из
html-документа<hr></h3>
<applet code="AppletWithParams.class" width="100%"
height="200">
<param name="size" value="25">
<param name="color" value="Желтый">
</applet>
<hr><i>Апплет создан на основе класса <b>JApplet</b>
библиотеки <b>Swing</b></i>
</body>
</html>
```

Обращаем внимание на наличие тегов <param> в блоке апплета. В соответствии с указанными там значениями в метке апплета применяется шрифт размера 25 и желтый фон. Как будет выглядеть соответствующий html-документ в окне браузера, показано на рис. 15.5.

Если теперь внести изменения в html-код (имеются в виду изменения в тегах <param> в части значений параметров name и color), при перезагрузке документа в окне браузера эти изменения автоматически вступят в силу.

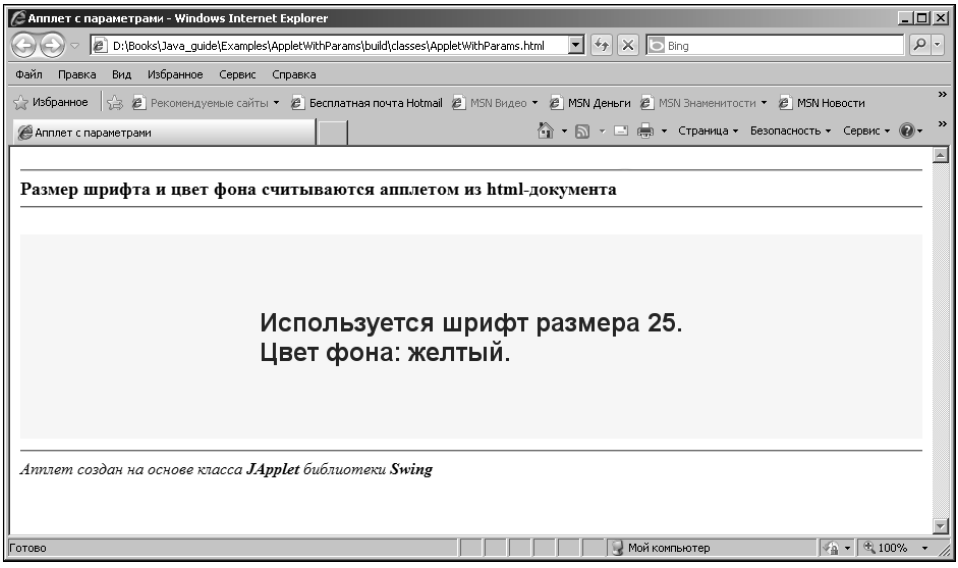


Рис. 15.5. Окно браузера с html-документом и апплетом, которому аргументы передаются через html-код

На заметку:

При перекомпиляции апплета перезагрузки html-документа в окне браузера окажется недостаточно. Придется заново открывать документ. Особенность рассмотренной выше ситуации как раз в том, что изменение значений аргументов, передаваемых апплету, не предполагает его перекомпиляции. Это удобно.

Апплет с элементами управления

Следующий пример посвящен созданию апплета с несколькими управляющими элементами. А именно, панель апплета состоит из нескольких блоков, или подпанелей. Документ с апплетом в окне браузера показан на рис. 15.6.

В верхней части панели апплета расположена панель с двумя вкладками. На корешке первой вкладки имеется надпись **Выбор шрифта**, а на корешке второй вкладки — надпись **Выбор текста**. По умолчанию открыта первая вкладка. Под панель с вкладками имеется область с текстом **Образец текста**, а под ней находится выделенная рамкой область с текстом **Здесь отображается текстовая строка**. Этот текст может меняться в процессе работы с апплетом.

Интерес в первую очередь вызывает верхняя панель с вкладками. На первой вкладке есть три выделенных рамками блока, содержащие утилиты для управления параметрами шрифта, с помощью которого отображается текст в нижней части панели апплета. В первом блоке размещен раскрывающийся спи-

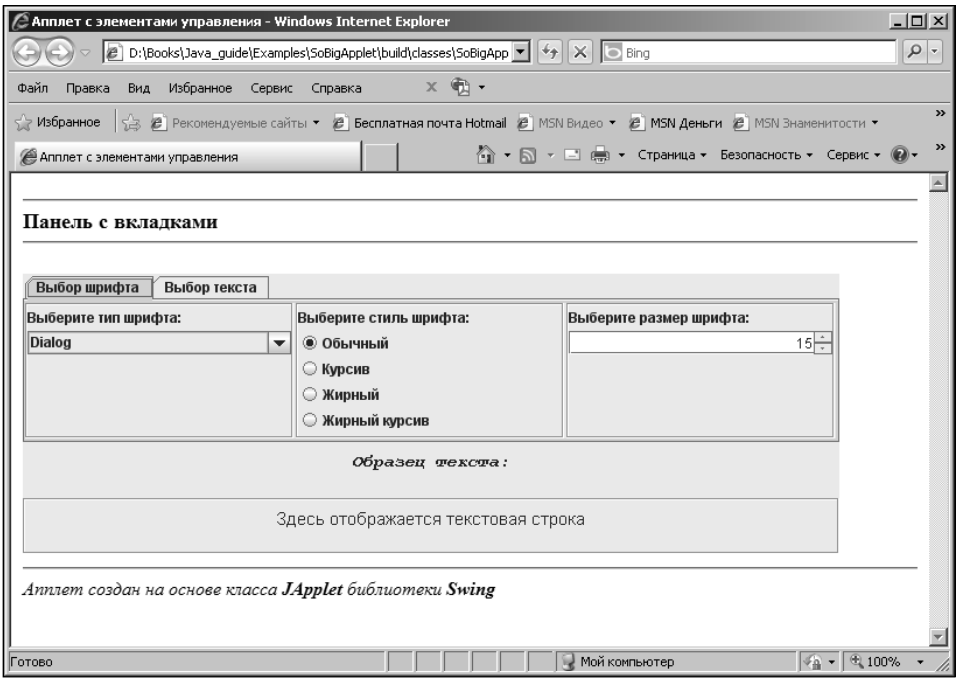


*Рис. 15.6. Окно браузера с апплетом и панелью, открытой на вкладке **Выбор шрифта***

сок для выбора типа шрифта. Над списком можно видеть текст **Выберите тип шрифта**. Во втором блоке под надписью **Выберите стиль шрифта** находится группа из четырех переключателей для выбора стиля шрифта (*обычный*, *курсив*, *жирный* и *жирный курсив*). В третьем блоке содержится *регулятор* (поле числового ввода с пиктограммами инкремента/декремента значения в поле ввода). Над полем есть надпись **Выберите размер шрифта**, а сам регулятор, как несложно догадаться, предназначен как раз для этой цели.

Содержимое вкладки **Выбор текста** можно видеть на рис. 15.7.

На вкладке имеется опция **Изменить значение текста** (по умолчанию *не выбрана*), недоступное по умолчанию поле для ввода текста (находится под надписью Поле для ввода текста), а также недоступная по умолчанию кнопка **Подтверждаю**.

Апплет работает так. Если верхняя панель открыта на первой вкладке, то любая манипуляция с элементом управления приводит к соответствующему изменению настроек шрифта, которым отображается надпись в нижней части панели апплета. К "любым манипуляциям" относится: выбор пункта с названием шрифта в раскрывающемся списке в первом блоке, выбор переключателя с типом шрифта во втором блоке или изменение значения

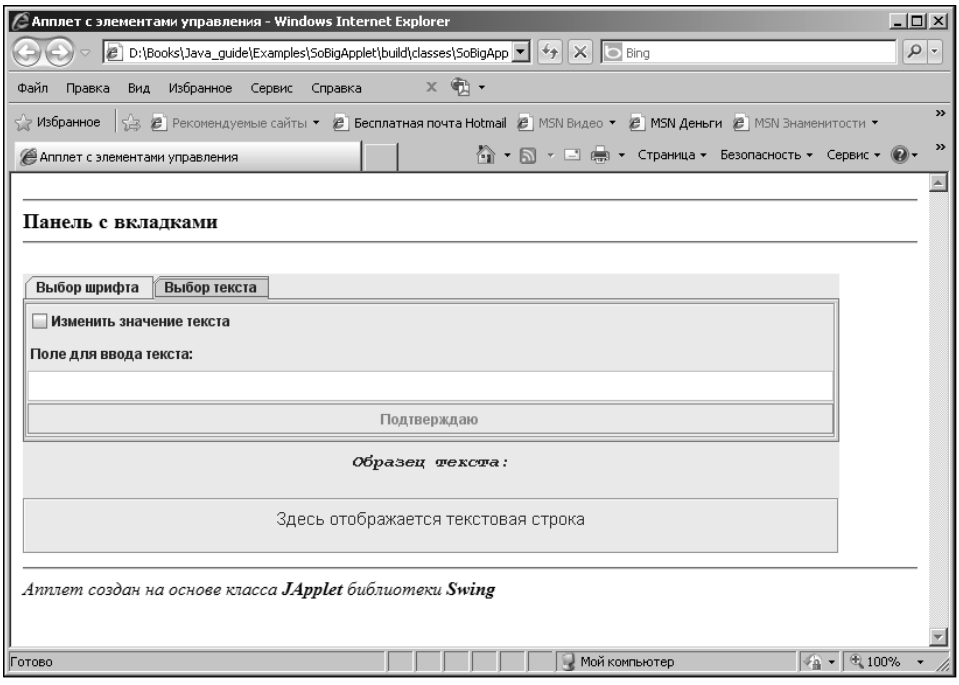


*Рис. 15.7. Окно браузера с апплетом и панелью, открытой на вкладке **Выбор текста***

регулятора в третьем блоке. На рис. 15.8 показано, как будет отображаться фраза **Здесь отображается текстовая строка** при изменении параметров шрифта по умолчанию (тип — **Dialog**, стиль — **Обычный**, размер — **15**) на новые значения (соответственно, тип — **Monospaced**, стиль — **Жирный курсив**, размер — **25**).

На заметку:

В Java определены пять логических шрифтов: **Dialog**, **DialogInput**, **Monospaced**, **SansSerif** и **Serif**. Именно они представлены в раскрывающемся списке для выбора типа шрифта. В зависимости от используемой операционной системы отображаться логические шрифты могут по-разному.

Вкладка **Выбор текста** предназначена для изменения текстового содержимого в нижней части панели апплета. Новый текст для отображения вводится в поле ввода, а с помощью кнопки **Подтверждаю** подтверждается сделанный ввод. Как уже отмечалось, по умолчанию поле ввода и кнопка недоступны. Чтобы изменить их отношение к своим функциональным обязанностям, необходимо установить флажок опции **Изменить значение текста**. На рис. 15.9 это героическое действие уже выполнено. В результате поле текстового ввода доступно (и туда вводится текст `Новый текст`), доступна и кнопка.

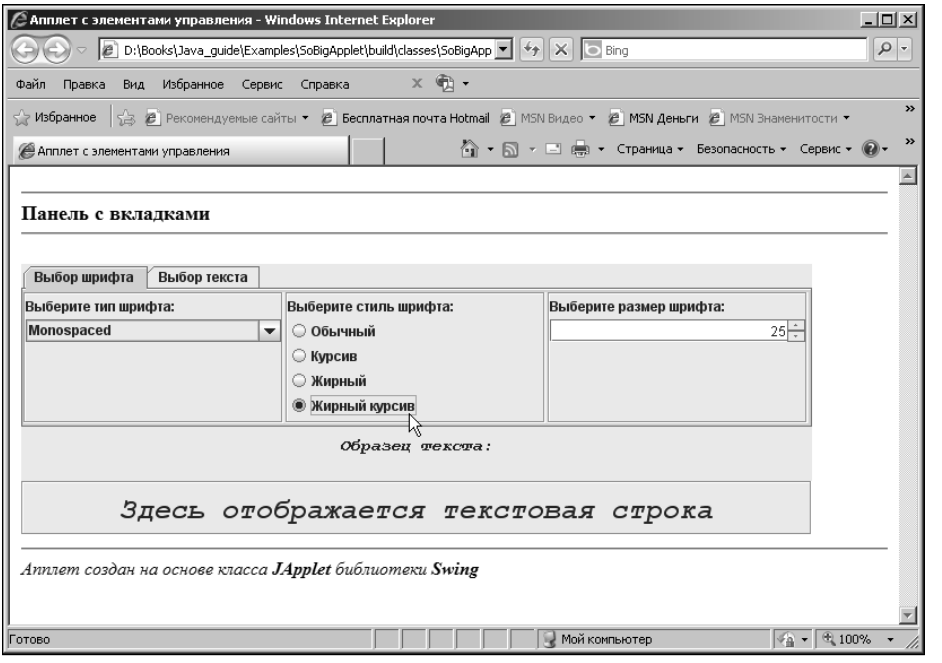


Рис. 15.8. Изменение шрифта в панели апплета

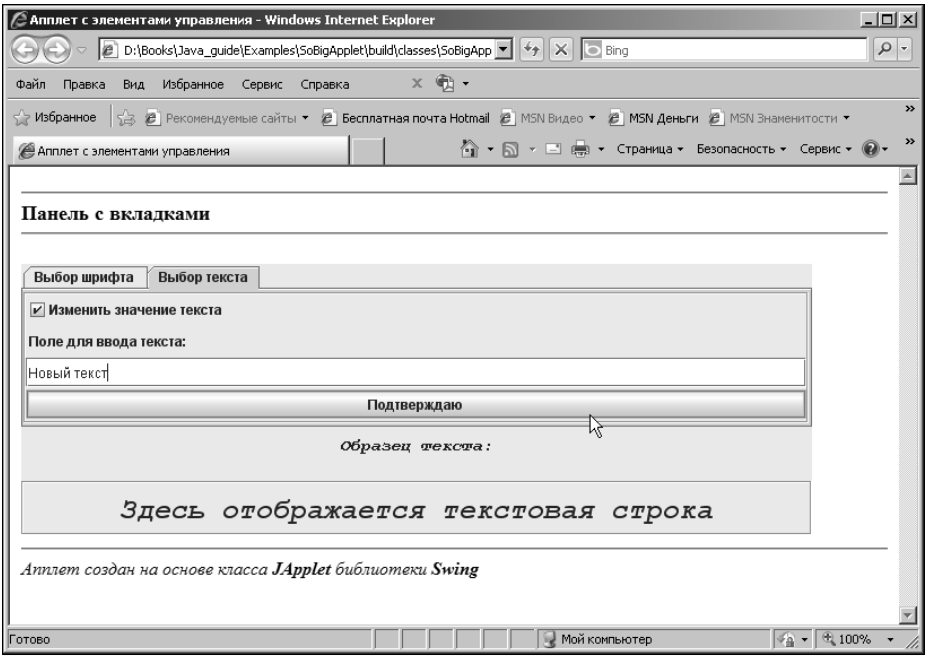


Рис. 15.9. Изменение текста в панели апплета

После щелчка на кнопке **Подтверждаю** новое текстовое значение отображается в нижней части окна панели. При этом выбор опции **Изменить значение текста** отменяется, а поле ввода и кнопка снова становятся недоступными. Результат показан на рис. 15.10.

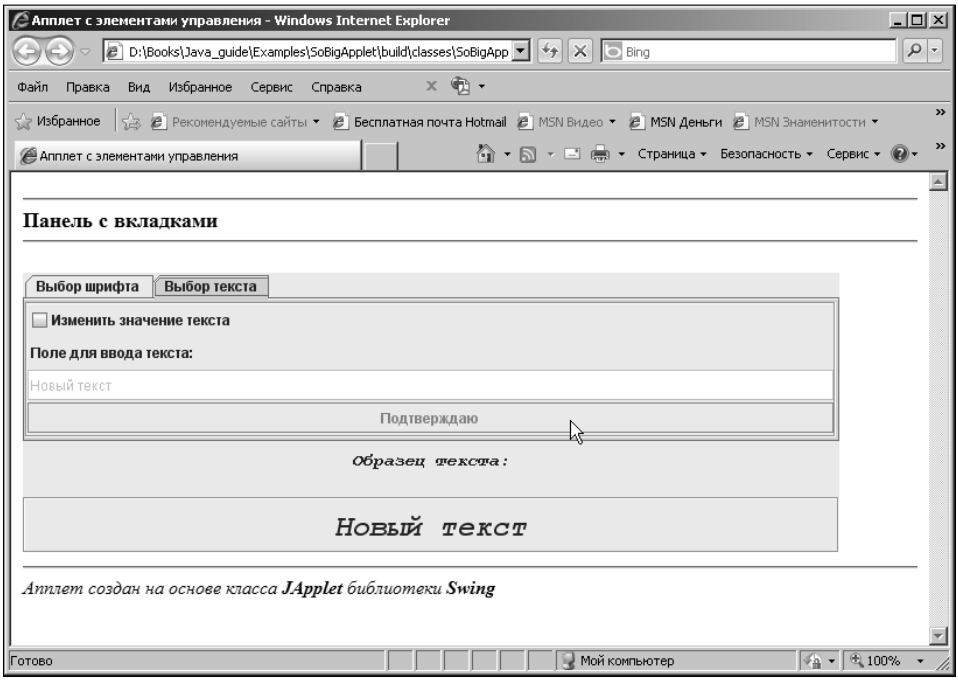


Рис. 15.10. Результат изменения текста в панели апплета

На заметку:

Если просто отменить выбор опции **Изменить значение текста**, то текстовое поле ввода и кнопка становятся недоступными.

Теперь рассмотрим программный код, который обеспечивает реализацию всего этого счастья. Обратимся к листингу 15.6.

Листинг 15.6. Код апплета с элементами управления

```
// Подключение пакетов:
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
// Класс для панели со статическим текстом:
class SimpleTextPanel extends JPanel{
   // Конструктор класса:
   SimpleTextPanel(String text){
```

```
      // Конструктор суперкласса:
      super();
      // Создание метки:
      JLabel lbl=new JLabel(text,JLabel.CENTER);
      // Применение шрифта для метки:
      lbl.setFont(new Font("Monospaced",Font.BOLD+Font.ITALIC,15));
      // Добавление метки на панель:
   add(lbl);}}
// Класс для панели с изменяемым текстом:
class MainTextPanel extends JPanel{
   // Метка с изменяемым текстом:
   private JLabel MainText;
   // Метод для изменения текста метки:
   public void setMainText(String text){
      MainText.setText(text);}
   // Метод для применения шрифта метки:
   public void setMainFont(Font fnt){
      MainText.setFont(fnt);}
   // Конструктор класса:
   MainTextPanel(String text,Font fnt){
      // Конструктор суперкласса:
      super();
      // Отображение рамки вокруг панели:
      setBorder(BorderFactory.createEtchedBorder());
      // Создание метки:
      MainText=new JLabel(text,JLabel.CENTER);
      // Применение для метки шрифта:
      MainText.setFont(fnt);
      // Добавление метки на панель:
      add(MainText);}}
// Классы подпанелей, используемые в панели с вкладками.
// Класс панели с раскрывающимся списком
// для выбора типа шрифта:
class FontTypePanel extends JPanel{
   // Поле - объект раскрывающегося списка:
   private JComboBox FontTypeList;
   // Метод для определения типа шрифта:
   public String getFontType(){
      return (String)FontTypeList.getSelectedItem();}
   // Конструктор класса:
   FontTypePanel(ActionListener hnd){
      // Вызов конструктора суперкласса:
      super();
      // Отображение рамки вокруг панели:
      setBorder(BorderFactory.createEtchedBorder());
      // Подключение менеджера компоновки:
      setLayout(new GridLayout(5,1,2,2));
      // Добавление текстовой метки:
      add(new JLabel("Выберите тип шрифта:"));
```

```
      // Список доступных шрифтов:
String[] fonts={"Dialog","DialogInput","Monospaced","Serif","SansSerif"};
      // Создание раскрывающегося списка:
      FontTypeList=new JComboBox(fonts);
      // Регистрация обработчика:
      FontTypeList.addActionListener(hnd);
      // Добавление раскрывающегося списка в панель:
      add(FontTypeList);}}
// Класс для панели с группой переключателей
// для выбора стиля шрифта:
class FontStylePanel extends JPanel{
   // Поля - объекты переключателей:
   private JRadioButton normal,italic,bold,it_bold;
   // Метод для определения стиля шрифта:
   public int getFontStyle(){
      if(normal.isSelected()) return Font.PLAIN;
      if(italic.isSelected()) return Font.ITALIC;
      if(bold.isSelected()) return Font.BOLD;
         else return Font.ITALIC+Font.BOLD;}
   // Конструктор класса:
   FontStylePanel(ActionListener hnd){
      // Вызов конструктора суперкласса:
      super();
      // Отображение рамки вокруг панели:
      setBorder(BorderFactory.createEtchedBorder());
      // Подключение менеджера компоновки:
      setLayout(new GridLayout(5,1,2,2));
      // Добавление в панель текстовой метки:
      add(new JLabel("Выберите стиль шрифта:"));
      // Объект для группы переключателей:
      ButtonGroup rbg=new ButtonGroup();
      // Первый переключатель:
      normal=new JRadioButton("Обычный",true);
      // Регистрация обработчика:
      normal.addActionListener(hnd);
      // Добавление первого переключателя в группу:
      rbg.add(normal);
      // Добавление первого переключателя на панель:
      add(normal);
      // Второй переключатель:
      italic=new JRadioButton("Курсив",false);
      // Регистрация обработчика:
      italic.addActionListener(hnd);
      // Добавление второго переключателя в группу:
      rbg.add(italic);
      // Добавление второго переключателя на панель:
      add(italic);
      // Третий переключатель:
      bold=new JRadioButton("Жирный",false);
```

```
      // Регистрация обработчика:
      bold.addActionListener(hnd);
      // Добавление третьего переключателя в группу:
      rbg.add(bold);
      // Добавление третьего переключателя на панель:
      add(bold);
      // Четвертый переключатель:
      it_bold=new JRadioButton("Жирный курсив",false);
      // Регистрация обработчика:
      it_bold.addActionListener(hnd);
      // Добавление четвертого переключателя в группу:
      rbg.add(it_bold);
      // Добавление четвертого переключателя на панель:
      add(it_bold);}}
// Класс панели с регулятором выбора размера шрифта:
class FontSizePanel extends JPanel{
   // Поле - объект регулятора:
   private JSpinner FontSize;
   // Метод для определения размера шрифта:
   public int getFontSize(){
      return (Integer)FontSize.getValue();}
   // Конструктор класса:
   FontSizePanel(ChangeListener hnd){
      // Вызов конструктора суперкласса:
      super();
      // Отображение рамки вокруг панели:
      setBorder(BorderFactory.createEtchedBorder());
      // Подключение менеджера компоновки:
      setLayout(new GridLayout(5,1,2,2));
      // Добавление текстовой метки:
      add(new JLabel("Выберите размер шрифта:"));
      // Объект для модели регулятора:
      SpinnerNumberModel snm=new SpinnerNumberModel(15,15,25,1);
      // Создание объекта регулятора:
      FontSize=new JSpinner(snm);
      // Регистрация обработчика:
      FontSize.addChangeListener(hnd);
      // Добавление регулятора на панель:
      add(FontSize);}}
// Класс для панели первой вкладки:
class FontPanel extends JPanel{
   // Поле - ссылка на панель выбора типа шрифта:
   private FontTypePanel FTP;
   // Поле - ссылка на панель выбора стиля шрифта:
   private FontStylePanel FSP;
   // Поле - ссылка на панель выбора размера шрифта:
   private FontSizePanel FZP;
   // Метод для определения шрифта:
   public Font newFont(){
```

```
   return new Font(FTP.getFontType(),FSP.getFontStyle(),FZP.getFontSize());}
   // Конструктор класса:
   FontPanel(ActionListener hnd1,ChangeListener hnd2){
      // Конструктор суперкласса:
      super();
      // Подключение менеджера компоновки:
      setLayout(new GridLayout(1,3,2,2));
      // Панель выбора типа шрифта:
      FTP=new FontTypePanel(hnd1);
      // Панель выбора стиля шрифта:
      FSP=new FontStylePanel(hnd1);
      // Панель выбора размера шрифта:
      FZP=new FontSizePanel(hnd2);
      // Добавление панели выбора типа шрифта в панель-контейнер:
      add(FTP);
      // Добавление панели выбора стиля шрифта в панель-контейнер:
      add(FSP);
      // Добавление панели выбора размера шрифта в панель-контейнер:
      add(FZP);}}
// Класс для панели второй вкладки:
class TextPanel extends JPanel{
   // Ссылка на текстовое поле:
   public JTextField TF;
   // Ссылка на опцию:
   public JCheckBox CB;
   // Ссылка на кнопку:
   public JButton BT;
   // Конструктор класса:
   TextPanel(ActionListener hnd){
      // Конструктор суперкласса:
      super();
      // Подключение менеджера компоновки:
      setLayout(new GridLayout(4,1,2,2));
      // Отображение рамки вокруг панели:
      setBorder(BorderFactory.createEtchedBorder());
      // Создание опции:
      CB=new JCheckBox("Изменить значение текста",false);
      // Регистрация обработчика:
      CB.addActionListener(hnd);
      // Создание текстового поля:
      TF=new JTextField();
      // Текстовое поле недоступно:
      TF.setEnabled(false);
      // Создание кнопки:
      BT=new JButton("Подтверждаю");
      // Регистрация обработчика:
      BT.addActionListener(hnd);
      // Кнопка недоступна:
      BT.setEnabled(false);
```

```
      // Добавление опции на панель:
      add(CB);
      // Добавление метки на панель:
      add(new JLabel(" Поле для ввода текста:"));
      // Добавление текстового поля на панель:
      add(TF);
      // Добавление кнопки на панель:
      add(BT);}}
// Класс для панели с вкладками:
class TPanel extends JTabbedPane{
   // Ссылка на панель выбора шрифта:
   public FontPanel FP;
   // Ссылка на панель выбора текста:
   public TextPanel TP;
   // Конструктор класса:
   TPanel(ActionListener hnd1,ChangeListener hnd2,ActionListener hnd3){
      // Конструктор суперкласса:
      super();
      // Создание панели выбора шрифта:
      FP=new FontPanel(hnd1,hnd2);
      // Добавление вкладки с панелью выбора шрифта:
      addTab("Выбор шрифта",FP);
      // Создание панели выбора текста:
      TP=new TextPanel(hnd3);
      // Добавление вкладки с панелью выбора текста:
      addTab("Выбор текста",TP);
   }}
// Класс создания апплета:
public class SoBigApplet extends JApplet{
   // Ссылка на панель с вкладками:
   private TPanel tpnl;
   // Ссылка на панель со статическим текстом:
   private SimpleTextPanel stp;
   // Ссылка на панель с изменяемым текстом:
   private MainTextPanel mtp;
   // Метод инициализации апплета:
   public void init(){
      // Обработчик для утилит выбора шрифта:
      FontHandler fhnd=new FontHandler();
      // Обработчик для утилит выбора текста:
      TextHandler thnd=new TextHandler();
      // Отключение менеджера компоновки:
      setLayout(null);
      // Переменные для записи размеров апплета:
      int w,h;
      // Считывание ширины апплета:
      w=getWidth();
      // Считывание высоты апплета:
      h=getHeight();
```

```
    // Создание панели с вкладками:
    tpnl=new TPanel(fhnd,fhnd,thnd);
    // Положение и размеры панели с вкладками:
    tpnl.setBounds(0,0,w,3*h/5);
    // Создание панели со статическим текстом:
    stp=new SimpleTextPanel("Образец текста:");
    // Положение и размеры панели со статическим текстом:
    stp.setBounds(0,3*h/5,w,h/5);
    // Панель с изменяемым текстом:
    mtp=new MainTextPanel("Здесь отображается текстовая строка",tpnl.
    FP.newFont());
    // Положение и размеры панели с изменяемым текстом:
    mtp.setBounds(0,4*h/5,w,h/5);
    // Добавление в апплет панели с вкладками:
    add(tpnl);
    // Добавление в апплет панели со статическим текстом:
    add(stp);
    // Добавление в апплет панели с изменяемым текстом:
    add(mtp);}
//Внутренний класс для обработки изменений типа, стиля и размера шрифта:
class FontHandler implements ActionListener,ChangeListener{
    // Внутренний метод для "считывания" и применения шрифта:
    private void applyFont(){
       mtp.setMainFont(tpnl.FP.newFont());}
    // Переопределение метода для реакции на изменение типа и стиля шрифта:
    public void actionPerformed(ActionEvent ae){
       applyFont();}
    // Переопределение метода для реакции на изменение размера шрифта:
    public void stateChanged(ChangeEvent ce){
       applyFont();}}
// Внутренний класс для обработки изменения отображаемого текста:
class TextHandler implements ActionListener{
// Внутренний метод для изменения состояния элементов управления панели:
private void OffOn(boolean state){
    tpnl.TP.CB.setSelected(state);
    tpnl.TP.TF.setEnabled(state);
    tpnl.TP.BT.setEnabled(state);}
// Переопределение метода для реакции на изменение
// состояния управляющих элементов панели:
public void actionPerformed(ActionEvent ae){
    String ac=ae.getActionCommand();
    if(ac.equalsIgnoreCase(tpnl.TP.BT.getText())){
       mtp.setMainText(tpnl.TP.TF.getText());
       OffOn(false);}
    else OffOn(tpnl.TP.CB.isSelected());}
}}
```

Код достаточно большой, но вместе с тем достаточно простой. Состоит он из описания нескольких классов, и в основном это классы панелей. Перед опи-

санием каждого из этих классов дадим краткое описание общей структуры программы. Общая идея такова: мы создаем панели, которые объединяем в новые панели, и так далее. А именно, панель апплета состоит из трех панелей: панели с вкладками (верхняя панель), панели со статическим текстом (панель по центру) и панели с изменяемым текстом (нижняя панель). При этом верхняя панель (с вкладками) на самом деле состоит из двух вкладок - по каждой панели на вкладку. Панель вкладки для выбора шрифта, в свою очередь, состоит из трех панелей: панели выбора типа шрифта, панели выбора стиля шрифта и панели выбора размера шрифта. Классы панелей описываются от более простых к более сложным.

Один из самых простых — класс для панели со статическим текстом `SimpleTextPanel`, который наследует класс `JPanel`. У конструктора класса есть текстовый аргумент `text`, который передается метке `lbl` при ее создании командой `JLabel lbl=new JLabel(text,JLabel.CENTER)`. Текст размещается по центру метки (второй аргумент `JLabel.CENTER`). Командой `lbl.setFont(new Font("Monospaced",Font.BOLD+Font.ITALIC,15))` для созданной метки применяется шрифт. Аргументом методу `setFont()` передается анонимный объект класса `Font`, который создается командой `new Font("Monospaced",Font.BOLD+Font.ITALIC,15)`. Объект определяет шрифт типа `Monospaced`, размера 15, стиль - жирный курсив.

На заметку:

Обратите внимание, что стиль *жирный курсив* (то есть *и жирный*, *и курсив*) определяется с помощью инструкции `Font.BOLD+Font.ITALIC`.

На панель метка добавляется командой `add(lbl)`.

Класс для панели с изменяемым текстом называется `MainTextPanel` и наследует класс `JPanel`. В этом классе есть закрытое поле `MainText` - ссылка на объект класса `JLabel`. Поскольку поле закрытое, для изменения текста метки предусмотрен специальный метод `setMainText()`, аргументом которому передается новое текстовое значение для метки. Для присваивания текстового значения метке в теле метода из объекта `MainText` вызывается метод `setText()`. Также в классе есть метод `setMainFont()` для применения шрифта метки. Аргументом методу передается объект класса `Font`. Для применения соответствующего шрифта из объекта метки в теле метода вызывается метод `setFont()`.

У конструктора класса два аргумента — текстовое значение для метки и шрифт (объект класса `Font`). В теле конструктора этот текст и шрифт используются при создании текстовой метки. Стоит также обратить внимание на команду `setBorder(BorderFactory.createEtchedBorder())`.

С ее помощью вокруг панели отображается рамка. Для применения рамки используется метод `setBorder()`. Аргументом методу передается объект класса `Border`, определяющий параметры рамки. В данном случае с помощью статического метода `createEtchedBorder()` класса `BorderFactory` создается объект, соответствующий рельефной рамке.

Описанные выше два класса описывают среднюю и нижнюю панели в области апплета. Далее рассмотрим классы, которые соответствуют внутренним панелям (подпанелям) для панели с вкладками.

Класс `FontTypePanel`, расширяющий класс `JPanel`, описывает панель-блок с раскрывающимся списком для выбора типа шрифта. У класса есть закрытое поле `FontTypeList` — ссылка на раскрывающийся список (объект класса `JComboBox`). Также в классе описан открытый метод `getFontType()`, который в качестве результата возвращает текстовую строку с именем выбранного в раскрывающемся списке шрифта. Чтобы узнать, какой пункт меню выбран, используем метод `getSelectedItem()` - вызываем его из объекта `FontTypeList`. Методом возвращается объект выбранного пункта меню. Чтобы преобразовать этот объект к текстовому формату, перед инструкцией `FontTypeList.getSelectedItem()` указывается инструкция преобразования к текстовому типу `(String)`.

У конструктора класса есть аргумент. Это обработчик для раскрывающегося списка. Обработчик должен быть объектом класса, реализующего интерфейс `ActionListener`. Здесь при передаче аргумента конструктору выполнена интерфейсная ссылка — в качестве типа аргумента указано имя интерфейса `ActionListener`.

📖 На заметку:

Напомним, что если класс реализует интерфейс, то для ссылки на объект этого класса можно использовать интерфейсную переменную. В качестве типа интерфейсной переменной указывается имя интерфейса. Через такую переменную можно получить доступ только к тем методам, которые объявлены в интерфейсе.

В конструкторе командой `add(new JLabel("Выберите тип шрифта:"))` на панель добавляется текстовая метка. Она реализована через анонимный объект класса `JLabel`, который передан аргументом методу `add()`. Список названий доступных шрифтов реализуется в виде текстового массива. Для этого используем команду `String[] fonts={"Dialog", "DialogInput","Monospaced","Serif","SansSerif"}`. Объект раскрывающегося списка создается командой `FontTypeList=new JComboBox(fonts)`. Регистрация обработчика для раскрывающегося списка выполняется командой `FontTypeList.addActionListener(hnd)`. Здесь

аргументом методу addActionListener() передается интерфейсная ссылка-аргумент конструктора класса. Созданный раскрывающийся список добавляется на панель (команда add(FontTypeList)).

Класс для панели с группой переключателей называется FontStylePanel. Он создается на основе класса JPanel и имеет закрытые поля normal, italic, bold и it_bold. Это ссылки на объекты класса JRadioButton, то есть переключатели. В классе определен открытый метод getFontStyle() для определения стиля шрифта. Основу метода составляет группа условных операторов, в которых, в зависимости от того, какой переключатель выбран (установлен), возвращается одна из трех целочисленных констант Font.PLAIN (обычный шрифт), Font.ITALIC (курсив), Font.BOLD (жирный шрифт) или сумма Font.ITALIC+Font.BOLD (жирный курсив). Для определения того, выбран переключатель или нет, использовался метод isSelected(), возвращающий значение true для выбранного переключателя, и false для невыбранного.

Конструктору класса в качестве аргумента передается обработчик класса, реализующего интерфейс ActionListener, для регистрации этого обработчика в переключателях.

Создание *группы* переключателей требует некоторых пояснений. Дело в том, что мало создать и добавить отдельные переключатели (объекты класса JRadioButton) на панель. Необходимо еще и объединить их в группу, чтобы в этой группе мог быть выделен один и только один переключатель. Группа переключателей - это объект класса ButtonGroup. Объект для группы переключателей создается командой ButtonGroup rbg=new ButtonGroup(). Затем группой команд для каждого из четырех переключателей группы создается объект переключателя, в переключателе регистрируется обработчик и переключатель добавляется в группу переключателей, а затем переключатель добавляется на панель:

```
normal=new JRadioButton("Обычный",true);
// Регистрация обработчика:
normal.addActionListener(hnd);
// Добавление первого переключателя в группу:
rbg.add(normal);
// Добавление первого переключателя на панель:
add(normal);
// Второй переключатель:
italic=new JRadioButton("Курсив",false);
// Регистрация обработчика:
italic.addActionListener(hnd);
// Добавление второго переключателя в группу:
rbg.add(italic);
```

```
// Добавление второго переключателя на панель:
add(italic);
// Третий переключатель:
bold=new JRadioButton("Жирный",false);
// Регистрация обработчика:
bold.addActionListener(hnd);
// Добавление третьего переключателя в группу:
rbg.add(bold);
// Добавление третьего переключателя на панель:
add(bold);
// Четвертый переключатель:
it_bold=new JRadioButton("Жирный курсив",false);
// Регистрация обработчика:
it_bold.addActionListener(hnd);
// Добавление четвертого переключателя в группу:
rbg.add(it_bold);
// Добавление четвертого переключателя на панель:
add(it_bold);
```

При создании объекта переключателя конструктору класса `RadioButton` передаются два аргумента: текстовая надпись для переключателя и логическое значение, определяющее, выделен указатель или нет (выделенным может быть только один переключатель в группе).

Класс `FontSizePanel` описывает панель с регулятором выбора размера шрифта. У класса объявлено закрытое поле `FontSize` - ссылка на объект класса `JSpinner`. Это объект *регулятора* - поле ввода (обычно числовое) с пиктограммами увеличения/уменьшения значения в поле. Для считывания размера шрифта на основе показателя регулятора предназначен открытый метод `getFontSize()`. Методом в качестве значения возвращается результат преобразования в целое число считанного из регулятора значения (инструкция `(Integer)FontSize.getValue()`). Значение считывается с помощью метода `getValue()`, а для приведения к целочисленному формату использована инструкция `(Integer)`.

Конструктору класса аргументом передается интерфейсная ссылка на объект класса, реализующего интерфейс `ChangeListener`. В этом интерфейсе объявлены методы, которые необходимо описать в классе обработчика события, связанного с изменением состояния регулятора. Переданный конструктору аргумент будет зарегистрирован в объекте регулятора.

Перед созданием непосредственно регулятора предварительно командой `SpinnerNumberModel snm=new SpinnerNumberModel(15,15,25,1)` создается *модель регулятора*. Это объект класса `SpinnerNumberModel`. Аргументы конструктора класса `SpinnerNumberModel` означают, что

начальное значение в поле регулятора равно 15, минимально возможное значение в поле регулятора равно 15, максимально возможное значение в поле регулятора равно 25, а шаг дискретности изменения значения в поле регулятора равен 1. На основе такой модели командой `FontSize=new JSpinner(snm)` создается объект регулятора. Командой `FontSize.addChangeListener(hnd)` для этого регулятора регистрируется обработчик-аргумент конструктора класса. После этого регулятор добавляется на панель (команда `add(FontSize)`).

Класс `FontPanel` для панели первой вкладки определяет панель, содержимое которой состоит из трех панелей: панели выбора типа шрифта (закрытое поле `FTP` класса `FontTypePanel`), панели выбора стиля шрифта (закрытое поле `FSP` класса `FontStylePanel`) и панели выбора размера шрифта (закрытое поле `FZP` класса `FontSizePanel`). У класса есть метод `newFont()` для определения шрифта на основе настроек элементов управления, размещенных на внутренних панелях `FTP`, `FSP` и `FZP`. Метод возвращает в качестве значения объект класса `Font`. Возвращаемый в качестве результата объект создается командой `new Font(FTP.getFontType(),FSP.getFontStyle(),FZP.getFontSize())`. В этой команде использованы методы, определяющие тип, стиль и размер шрифта и определенные в классах внутренних панелей.

У конструктора класса `FontPanel` два аргумента. Первый является обработчиком для раскрывающегося списка выбора типа шрифта и переключателей, задающих стиль шрифта. Второй аргумент является ссылкой на объект обработчика для регулятора, определяющего размер шрифта. Эти аргументы передаются, в свою очередь, аргументами конструкторам классов внутренних панелей.

На заметку:

Обратите внимание, что изменения состояния раскрывающегося списка и состояния группы переключателей обрабатываются одним и тем же обработчиком.

В конструкторе класса `FontPanel` командой `setLayout(new GridLayout(1,3,2,2))` подключается менеджер компоновки, который размещает компоненты на панели в один ряд последовательно в три ячейки (по одной ячейке на панель). Следующими командами создаются три внутренние панели:

```
// Панель выбора типа шрифта:
FTP=new FontTypePanel(hnd1);
// Панель выбора стиля шрифта:
FSP=new FontStylePanel(hnd1);
// Панель выбора размера шрифта:
FZP=new FontSizePanel(hnd2);
```

Добавляются внутренние панели в панель-контейнер командами `add(FTP)`, `add(FSP)` и `add(FZP)`.

Класс `TextPanel` для панели второй вкладки имеет такие открытые поля: ссылка на текстовое поле `JTextField TF` (поле, в которое вводится новый текст), ссылка на опцию `JCheckBox CB` (опция позволяет контролировать режим доступа к полю и кнопке), ссылка на кнопку `JButton BT` (кнопка позволяет применить новое текстовое значение для отображения на панели апплета).

У конструктора класса один аргумент - обработчик для кнопки и опции (один на двух). Создание опции реализуется через команду `CB=new JCheckBox("Изменить значение текста",false)`. Первый текстовый аргумент конструктора класса `JCheckBox` задает надпись для опции. Второй аргумент `false` означает, что вначале флажок опции не установлен. Обработчик для опции регистрируется командой `CB.addActionListener(hnd)` (здесь `hnd` - конструктор класса `TextPanel`). Пустое текстовое поле создаем командой `TF=new JTextField()`. Вначале текстовое поле недоступно (команда `TF.setEnabled(false)`). Командами `BT=new JButton("Подтверждаю")`, `BT.addActionListener(hnd)` и `BT.setEnabled(false)` создается кнопка, в ней регистрируется обработчик и кнопка делается недоступной соответственно. Командами `add(CB)`, `add(new JLabel(" Поле для ввода текста:"))` (здесь анонимный объект метки передан аргументом методу `add()` для добавления на панель), `add(TF)` и `add(BT)` на панель добавляются последовательно опция, текстовая метка, текстовое поле и кнопка.

Класс для панели с вкладками объявляется с именем `TPanel` на основе встроенного класса `JTabbedPane` библиотеки Swing.

На заметку:

Стандартная панель с вкладками - это объект класса `JTabbedPane`.

У класса `TPanel` есть два открытых поля `FontPanel FP` и `TextPanel TP`. Это ссылки на объекты первой и второй панелей кладок. У конструктора класса три аргумента, и все они обработчики. Первый аргумент - интерфейсная ссылка на объект класса, реализующего интерфейс `ActionListener`. Первым аргументом (ссылка `ActionListener hnd1`) передается обработчик для раскрывающегося списка выбора типа шрифта и переключателей выбора стиля шрифта. Второй аргумент - интерфейсная ссылка на объект класса, реализующего интерфейс `ChangeListener`. Вторым аргументом (ссылка `ChangeListener hnd2`) передается обработчик для регулятора размера шрифта. Третий аргумент - интерфейсная ссылка на объект класса, реализующего интерфейс `ActionListener`. Третьим аргу-

ментом (ссылка ActionListener hnd3) передается обработчик для опции и кнопки на вкладке изменения текста. В конструкторе класса TPanel командой FP=new FontPanel(hnd1,hnd2) создается панель с утилитами для настройки параметров шрифта. Добавление новой (пока что первой) вкладки с этой панелью в панель с вкладками выполняется командой addTab("Выбор шрифта",FP). Первым аргументом методу добавления вкладки addTab() указывается текст корешка вкладки, а второй аргумент - панель, которая собственно и формирует вкладку.

Панель выбора текста создается командой TP=new TextPanel(hnd3). Добавление вкладки с этой панелью выполняем командой addTab("Выбор текста",TP). Назначение аргументов такое же, как в предыдущем случае.

На заметку:

Такой объект, как панель с вкладками, - в данном случае объект класса TPanel. Это возможно, поскольку класс TPanel наследует класс JTabbedPane. Объект состоит из нескольких панелей, каждая из которых формирует вкладку.

Класс для создания апплета называется SoBigApplet. Класс стандартно создается на основе класса JApplet. В этом классе мы объявляем закрытое поле TPanel tpnl (ссылка на панель с вкладками), закрытое поле SimpleTextPanel stp (ссылка на панель со статическим текстом) и закрытое поле MainTextPanel mtp (ссылка на панель с изменяемым текстом). Класс SoBigApplet интересен также тем, что в нем описаны два внутренних класса. Первый — внутренний класс FontHandler для обработки изменений типа, стиля и размера шрифта. Класс реализует сразу два интерфейса: ActionListener и ChangeListener. Поэтому объект класса может использоваться как для обработки событий, связанных с раскрывающимся списком и переключателями, так и регулятором размера шрифта. Задачу по обработке изменения состояния управляющих элементов решаем в комплексе - при любом действии с хотя бы одним из упомянутых выше элементов определяем параметры шрифта и применяем их. Другими словами, каждый раз в таких случаях последовательность действий одна и та же. Реализуем ее в виде закрытого метода applyFont() внутреннего класса FontHandler. В методе выполняется всего одна команда mtp.setMainFont(tpnl.FP.newFont()). Для применения настроек шрифта из объекта панели mtp с изменяемым текстом (самая нижняя панель в апплете) вызывается метод setMainFont(). Аргументом этому методу передается объект класса Font, который, в свою очередь, вычисляется инструкцией tpnl.FP.newFont(). Здесь мы учли, что у панели с вкладками tpnl есть поле FP - объект панели настройки шрифта, у которого, в свою очередь, есть метод newFont(), возвращающий на основе настроек управляющих элементов соответствующий объект класса Font.

В рамках реализации интерфейса ActionListener переопределяется метод actionPerformed(). В методе вызывается другой метод - applyFont(). Это же метод вызывается в теле метода stateChanged(). Метод stateChanged() должен быть переопределен, поскольку внутренний класс FontHandler кроме интерфейса ActionListener реализует еще и интерфейс ChangeListener. Аргументом методу stateChanged() передается событие - объект класса ChangeEvent (в данном случае явно не используется).

На заметку:

Для того чтобы обрабатывать события класса ChangeEvent, связанные с регулятором, в заголовке программы использовали команду импорта пакета import javax.swing.event.*.

Внутренний класс TextHandler реализует интерфейс ActionListener и предназначен для обработки событий, связанных с управляющими элементами на второй вкладке, предназначенной для изменения отображаемого текста. В классе описан закрытый метод OffOn(), который в зависимости от значения своего логического аргумента делает доступными или недоступными текстовое поле и кнопку и соответственно устанавливает или отменяет установку флажка опции. Также в классе переопределяется метод actionPerformed(). Этот метод вызывается и при операциях с опцией, и при щелчке на кнопке. Поэтому в теле метода командой String ac=ae.getActionCommand() на основе аргумента ae метода определяется команда действия и результат записывается в переменную ac.

На заметку:

Напомним, что объект класса ActionEvent, который передается методу actionPerformed(), содержит информацию о компоненте, который вызвал событие. Метод getActionCommand(), вызванный из этого объекта, возвращает команду действия. Для кнопки по умолчанию это ее текст. Текст кнопки можно определить с помощью метода getText().

В условном операторе проверяем, какой компонент вызвал событие. Для этого используем условие ac.equalsIgnoreCase(tpnl.TP.BT.getText()), в котором текст команды действия сравнивается с текстом кнопки. Если условие истинно, значит, был выполнен щелчок на кнопке. Иначе методом исключения остается установка/отмена флажка опции.

На заметку:

Кнопка (объект BT) является полем панели второй вкладки (объект TP), которая является полем панели с вкладками (объект tpnl). Поэтому полная ссылка на кнопку имеет вид tpnl.TP.BT.

Если событие вызвала кнопка, то командой `mtp.setMainText(tpnl.TP.TF.getText())` применяется новое значение текста. В этой команде инструкция `tpnl.TP.TF.getText()` представляет собой текстовое значение в текстовом поле `TF` (полная ссылка на поле имеет вид `tpnl.TP.TF`) и этот текст передается аргументом методу `setMainText()`. Затем командой `OffOn(false)` отменяется флажок опции, а также делаются недоступными текстовое поле и кнопка.

Если событие произошло не с кнопкой, то командой `OffOn(tpnl.TP.CB.isSelected())` все три элемента (опция, поле и кнопка) переводятся в то состояние, в котором находится опция.

В методе `init()` прописаны те действия, которые выполняются при инициализации (создании) апплета. Проанализируем код этого метода. В нем командами `FontHandler fhnd=new FontHandler()` и `TextHandler thnd=new TextHandler()` создаются обработчики для утилит выбора шрифта и текста соответственно. Отключение менеджера компоновки для панели апплета выполняется командой `setLayout(null)`. Целочисленные переменные `w` и `h` предназначены для записи размеров апплета, которые считываются методами `getWidth()` (ширина панели апплета) и `getHeight()` (высота панели апплета).

На заметку:

В нашем примере высота и ширина панели апплета задаются в коде html-документа, в котором отображается апплет. Код html-документа приведен несколько позже.

Создание панели с вкладками выполняется с помощью команды `tpnl=new TPanel(fhnd,fhnd,thnd)`. Здесь первым и вторым аргументом конструктору класса `TPanel()` передается один и тот же обработчик `fhnd`. Это возможно, поскольку класс обработчика `FontHandler` реализует сразу два интерфейса: `ActionListener` и `ChangeListener`. Поэтому интерфейсная переменная каждого из этих интерфейсов может быть ссылкой на объект класса.

На заметку:

Напомним, что первым аргументом конструктору класса должна передаться переменная интерфейса `ActionListener`, а вторым - переменная интерфейса `ChangeListener`.

Положение и размеры панели с вкладками задаются командой `tpnl.setBounds(0,0,w,3*h/5)`. Создание панели со статическим текстом выполняется командой `stp=new SimpleTextPanel("Образец текста:")`. Положение и размеры панели со статическим текстом задаем командой `stp.setBounds(0,3*h/5,w,h/5)`.

Панель с изменяемым текстом создается так:

```
mtp=new MainTextPanel("Здесь отображается текстовая
строка",tpnl.FP.newFont());
```

Здесь применяемый для отображения текста шрифт считывается инструкцией `tpnl.FP.newFont()`. Положение и размеры панели с изменяемым текстом задаем командой `mtp.setBounds(0,4*h/5,w,h/5)`. Наконец, выполняется добавление в апплет созданных и настроенных панелей:

```
// Добавление в апплет панели с вкладками:
add(tpnl);
// Добавление в апплет панели со статическим текстом:
add(stp);
// Добавление в апплет панели с изменяемым текстом:
add(mtp);
```

На заметку:

В рассматриваемом примере линейные размеры внутренних панелей апплета определяются в отношении к размерам апплета.

Что касается кода соответствующего html-документа, то этот код достаточно прост. Убедиться в этом можно, взглянув на листинг 15.7.

Листинг 15.7. Код html-документа для апплета с элементами управления

```
<html>
<head><title>Апплет с элементами управления</title></head>
<body>
<h3><hr>Панель с вкладками<hr></h3>
<applet code="SoBigApplet.class" width=700 height="240">
</applet>
<hr><i>Апплет создан на основе класса <b>JApplet</b>
библиотеки <b>Swing</b></i>
</body>
</html>
```

Интерес здесь могут представлять разве что ширина и высота апплета. При изменении этих параметров в коде html-документа (и сохранении изменений) перезагрузка документа приведет к автоматическому перемасштабированию размеров внутренних панелей и элементов управления апплета.

Резюме

1. Апплет представляет собой программу, которая выполняется под управлением браузера.
2. Создаются апплеты на основе класса `Applet` библиотеки AWT или класса `JApplet` библиотеки Swing.
3. У апплетов нет метода `main()`, роль конструктора играет метод `init()`. Каждый раз при обращении к странице с апплетом вызывается метод `start()`. Метод `stop()` вызывается при уходе со страницы, а метод `destroy()` — при завершении работы апплета.
4. В html-документ апплет добавляется с помощью пары тегов `<applet>` и `</applet>`.
5. Для передачи апплету параметров через html-документ используют тег `<param>`.
6. В апплете могут использоваться практически те же компоненты графического интерфейса, что и в обычном приложении.

Заключение

О том, что было, и чего не было
Вот так всегда, на самом интересном месте!
(Из м/ф "Возвращение блудного попугая")

Любую книгу, которую начинаешь писать, нужно рано или поздно заканчивать. Откровенно говоря, заканчивать книгу всегда приятно. Но вместе с тем, каждый раз остается ощущение чего-то недосказанного. В случае с книгами по языкам программирования это "чего-то" имеет четко определенное содержание. Причина простая и очевидная невозможно написать книгу обо всем даже в пределах вполне конкретного языка программирования. А если бы и удалось, то такая книга, в силу своего объема, имела бы достаточно ограниченную область применимости. Поэтому даже откровенные справочники в несколько томов содержат все в разумных пределах.

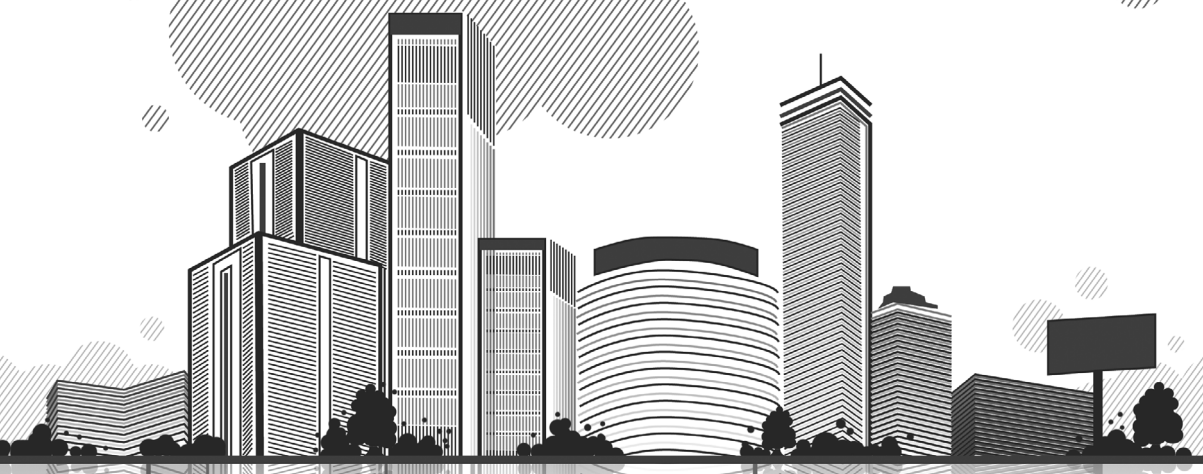
Язык программирования — это в первую очередь синтаксис, плюс стандартные классы/библиотеки. Если язык программирования достаточно популярный и "профессиональный", то, как правило, стандартных библиотек впечатляюще много, так что каждой из них можно посвятить отдельную книгу. В принципе нечто похожее имеет место и в отношении Java. Поэтому при подборе материала для книги важно было выделить ключевые моменты в идейном подходе, реализованном в Java. Понятно, что в основе лежит ООП, но в таком деле даже второстепенные на первый взгляд детали не всегда на самом деле являются второстепенными.

В книге представлены все основные синтаксические конструкции, которые понадобятся или могут понадобиться читателю при составлении программных кодов – во всяком случае, на начальном этапе. Отмечены нашим вниманием были и некоторые стандартные классы. Особого отношения здесь требуют классы библиотеки Swing. С одной стороны, базовые классы этой библиотеки в книге описаны и показано, как их использовать на практике. С другой стороны, на эту тему можно писать и писать. И единственный способ освоить тему создания приложений с графическим интерфейсом – это создавать такие приложения. Пробовать, экспериментировать, а если не получается снова пробовать. Сделать первые шаги в этом направлении поможет книга. Все остальное зависит от читателя. Так что в добрый путь!

softline®

Services Software Cloud

ИТ-архитектура вашего бизнеса



CLOUD

SaaS

PROJECTS

SOLUTIONS

remote workplace

SAM

юридическая поддержка

VIRTUALIZATION

LINUX

программное обеспечение

hardware

web-разработка

legal support

education

Mirapolis

ERP&CRM

защита персональных данных

SERVICES

SECURITY

учебный центр

CAD
CAM
CAE

SOFTWARE

САПР и ГИС

информационная безопасность

решения Microsoft

техническая поддержка

search

flexible

CONSULTING

data leak prevention

корпоративный портал DeskWork

open source

PLATFORM

protection

hosting

technical support

16+

+7(812) 777-4446 www.softline.ru info.spb@softline.ru

Уважаемые господа!
Книги издательства «Наука и Техника»

Вы можете заказать наложенным платежом
в нашем интернет-магазине

www.nit.com.ru,

а также приобрести

➢ **в крупнейших магазинах г. Москвы:**

Т Д «БИБЛИО-ГЛОБУС»	ул. Мясницкая, д. 6/3, стр. 1, ст. М «Лубянка» тел. (495) 781-19-00, 624-46-80
Московский Дом Книги, «ДК на Новом Арбате»	ул.Новый Арбат, 8, ст. М «Арбатская», тел. (495) 789-35-91
Московский Дом Книги, «Дом технической книги»	Ленинский пр., д.40, ст. М «Ленинский пр.», тел. (499) 137-60-19
Московский Дом Книги, «Дом медицинской книги»	Комсомольский пр., д. 25, ст. М «Фрунзенская», тел. (499) 245-39-27
Дом книги «Молодая гвардия»	ул. Б. Полянка, д. 28, стр. 1, ст. М «Полянка» тел. (499) 238-50-01
Сеть магазинов «Новый книжный»	тел. (495) 937-85-81, (499) 177-22-11

➢ **в крупнейших магазинах г. Санкт-Петербурга:**

Санкт-Петербургский Дом Книги	Невский пр. 28 тел. (812) 448-23-57
«Энергия»	Московский пр. 57 тел. (812) 373-01-47
«Аристотель»	ул. А. Дундича 36, корп. 1 тел. (812) 778-00-95
Сеть магазинов «Книжный Дом»	тел. (812) 559-98-28

➢ **в регионах России:**

г. Воронеж, пл. Ленина д. 4	«Амиталь»	(4732) 24-24-90
г. Екатеринбург, ул. Антона Валека д. 12	«Дом книги»	(343) 253-50-10
г. Екатеринбург	Сеть магазинов «100 000 книг на Декабристов»	(343) 353-09-40
г. Нижний Новгород, ул. Советская д. 14	«Дом книги»	(831) 277-52-07
г. Смоленск, ул. Октябрьской революции д. 13	«Кругозор»	(4812) 65-86-65
г. Челябинск, ул. Монакова, д. 31	«Техническая книга»	(904) 972 50 04
г. Хабаровск	Сеть книжно-канцелярских магазинов фирмы «Мирс»	(4212) 26-87-30

➢ **и на Украине (оптом и в розницу) через представительство издательства**

г. Киев, ул. Курчатова 9/21, «Наука и Техника», ст. М «Лесная»
(044) 516-38-66
e-mail: nits@voliacable.com

Мы рады сотрудничеству с Вами!

Группа подготовки издания:

Зав. редакцией компьютерной литературы: *М. В. Финков*

Редактор: *М. А. Финкова*

Корректор: *А. В. Громова*

ООО «Наука и Техника»

Лицензия №000350 от 23 декабря 1999 года.

198097, г. Санкт-Петербург, ул. Маршала Говорова, д. 29.

Подписано в печать 19.04.2016. Формат 70x100 1/16.

Бумага газетная. Печать офсетная. Объем 23 п. л.

Тираж . Заказ