Поволжский государственный университет
телекоммуникаций и информатики

Акчурин Э.А.

Программирование на языке Java

Учебное пособие для студентов направления
«Информатика и вычислительная техника»

Самара 2011

Факультет информационных систем и технологий

Кафедра «Информатика и вычислительная техника»

Автор - д.т.н., профессор Акчурин Э.А.

Другие материалы по дисциплине Вы найдете на сайте

www.ivt.psuti.ru

0. Введение

0.1. Список литературы

1. Эккель Б.. Философия Java. Библиотека программиста. 4-е изд. – СПб.: Питер, 2009. – 640 с.
2. Хорстманн К. Java 2. Библиотека профессионала, том 1. Основы. 7-е изд.: Пер. с англ. - М.: "Вильямс", 2006. - 896 с.
3. Хорстманн К. Java 2. Библиотека профессионала, том 2. Тонкости программирования. Пер. с англ. - М.: "Вильямс", 2002. - 1120 с.
4. Щилдт Г. Холмс Д. Искусство программирования на Java. Пер. с англ. - М.: "Вильямс", 2005. - 335 с.
5. Арнольд К., Гослинг Д.. Язык программирования Java. Пер. с англ. - СПб.: Питер, 1997. - 304 с.
6. Монахов В. Язык программирования Java и среда NetBeans, 3-е издание. БХВ=Петербург, 2011, 704 с.
7. Казарин С., Клишин А. Среда разработки Java- приложений Eclipse - М., 2008. 77 с.
8. Хемраджани А. Гибкая разработка приложений на Java с помощью Spring, Hubernate и Eclipse.:Пер. с англ. - М.: "Вильямс", 2008. - 352 с.

0.2. Что такое Java

Это остров Ява в Малайском архипелаге, территория Индонезии. Это и сорт кофе, который любят пить создатели Java (произносится "Джава", с ударением на первом слоге)

Сначала Java (официальный день рождения технологии Java - 23 мая 1995 г.) предназначалась для программирования бытовых электронных устройств, таких как телефоны.

Потом Java стала применяться для программирования браузеров - появились апплеты.

Затем оказалось, что на Java можно создавать полноценные приложения. Их графические элементы стали оформлять в виде компонентов - появились JavaBeans, с которыми Java вошла в мир распределенных систем и промежуточного программного обеспечения.

Остался один шаг до программирования серверов - этот шаг был сделан, появились сервлеты и EJB (Enterprise JavaBeans).

Серверы должны взаимодействовать с базами данных - появились драйверы JDBC (Java Data Base Connection). Взаимодействие оказалось удачным, и мно-

гие системы управления базами данных и даже операционные системы включили, Java в свое ядро, например Oracle, Linux, MacOS X, AIX.

0.3. Почему Java

Современные средства создания Java-приложений разработаны для различных платформ: Linux, Solaris, Windows и MacOS. Важнейшее преимущество языка Java заключается в том, что приложение, написанное на основе данного языка, является независимым от платформы и архитектуры процессора, который будет выполнять алгоритм данного приложения.

Главным звеном в данном процессе является виртуальная машина Java — это специальная программа, которая имеющийся откомпилированный независимый байт-код преобразует в набор инструкций для конкретного процессора. Программа должна быть предварительно установлена на компьютер, где планируется запуск приложения. Байт-код формируется из кода Java встроенным компилятором

Язык Java является объектно-ориентированным и поставляется с достаточно объемной библиотекой классов. Библиотеки классов Java значительно упрощают разработку приложений, предоставляя в распоряжение программиста мощные средства решения стандартных задач.

Почти сразу же после появления Java было создано большое количество интегрированных сред разработки (ИСР) программ для этого языка: Eclipse (Eclipse Foundation), NetBeans (Sun), JBuilder (Inprise), Visual Age (IBM), VisualCafe (Symantec) и др. Большинство из существующих ИСР приложений сами написаны полностью на Java и имеют развитые средства визуального программирования.

Язык Java на сегодняшний день вхоит в тройку наиболее востребованных языков прогиммирования. В 2010 году ор был вторым а сейчас стал первым. Ниже общемировые данные по востребованности языков программирования за апрель 2011 года по данным всемирноизветной компании TIOBE Software..

Position Apr 2011	Position Apr 2010	Delta in Position	Programming Language	Ratings Apr 2011	Delta Apr 2010	Status
1	2	↑	Java	19.043%	+0.99%	A
2	1	↓	C	16.162%	-1.90%	A
3	3	=	C++	9.225%	-0.48%	A
4	6	↑↑	C#	7.185%	+2.75%	A
5	4	↓	PHP	6.584%	-3.08%	A
6	7	↑	Python	4.931%	+0.73%	A
7	5	↓↓	(Visual) Basic	4.682%	-1.71%	A
8	11	↑↑↑	Objective-C	4.386%	+2.10%	A
9	8	↓	Perl	1.991%	-1.56%	A
10	10	=	JavaScript	1.513%	-0.96%	A
11	12	↑	Ruby	1.482%	-0.74%	A
12	20	↑↑↑↑↑↑↑↑	Lua	1.035%	+0.51%	A
13	9	↓↓↓↓	Delphi	1.034%	-1.68%	A
14	-	=	Assembly	0.967%	0.00%	A
15	23	↑↑↑↑↑↑↑↑	Lisp	0.934%	+0.45%	A
16	32	↑↑↑↑↑↑↑↑↑↑	Ada	0.768%	+0.41%	A
17	16	↓	Pascal	0.713%	+0.06%	A
18	21	↑↑↑	Transact-SQL	0.583%	+0.08%	B
19	-	=	Scheme	0.581%	0.00%	B
20	15	↓↓↓↓↓	Go	0.557%	-0.15%	A--

0.4. Сборка мусора

Одни из самых неприятных ошибок, которые портят жизнь программисту, это ошибки, связанные с управлением памятью. В таких языках, как C и C++, в которых управление памятью целиком возложено на программиста, львиная

доля времени, затрачиваемого на отладку программы, приходится на борьбу с подобными ошибками.

Давайте перечислим типичные ошибки при управлении памятью (некоторые из них особенно усугубляются в том случае, если в программе существуют несколько указателей на один и тот же блок памяти):

- Преждевременное освобождение памяти (premature free). Эта беда случается, если мы пытаемся использовать объект, память для которого была уже освобождена. Указатели на такие объекты называются висящими (dangling pointers), а обращение по этим указателям дает непредсказуемый результат.
- Двойное освобождение (double free). Иногда бывает важно не перестараться и не освободить ненужный объект дважды.
- 3Утечки памяти (memory leaks). Когда мы постоянно выделяем новые блоки памяти, но забываем освобождать блоки, ставшие ненужными, память в конце концов заканчивается.
- Фрагментация адресного пространства (external fragmentation). При интенсивном выделении и освобождении памяти может возникнуть ситуация, когда непрерывный блок памяти определенного размера не может быть выделен, хотя суммарный объем свободной памяти вполне достаточен. Это происходит, если используемые блоки памяти чередуются со свободными блоками и размер любого из свободных блоков меньше, чем нам нужно.

Проблема особенно критична в серверных приложениях, работающих в течение длительного времени.

В программах, работающих в Java, все вышеперечисленные ошибки никогда не возникают, потому что эти программы используют реализованное в виртуальной машине автоматическое управление памятью, а именно – сборщик мусора.

Работа сборщика мусора заключается в освобождении памяти, занятой ненужными объектами. При этом сборщик мусора также умеет «двигать» объекты в памяти, тем самым устраняя фрагментацию адресного пространства.

1. Библиотеки классов Java

1.1. Библиотеки классов JDK

JDK (Java Development Kit) - бесплатно распространяемый Oracle Corporation (ранее Sun Microsystems) комплект разработчика приложений на языке Java, включающий в себя:

- Компилятор Java (javac).
- Стандартные библиотеки классов Java.
- Примеры, документацию.
- Различные утилиты.
- JRE (Исполнительную систему Java.

В состав JDK не входит ИСР на Java, поэтому разработчик, использующий только JDK, вынужден использовать внешний текстовый редактор и компилировать свои программы, используя утилиты командной строки.

Все современные интегрированные среды разработки на Java, такие, как NetBeans, IntelliJ IDEA, Borland JBuilder, Eclipse, опираются на сервисы, предоставляемые JDK. Большинство из них для компиляции Java-программ используют компилятор из комплекта JDK. Поэтому эти среды разработки либо включают в комплект поставки одну из версий JDK, либо требуют для своей работы предварительной инсталляции JDK на машине разработчика.

С некоторых пор фирма Sun предоставляет полные исходные тексты JDK, включая исходные тексты самого Java-компилятора.

JDK 1.0 (23 января 1996).

Кодовое имя Oak (дуб).

Первый выпуск. Первой стабильной версией стала JDK 1.0.2.

Графический интерфейс пользователя (ГИП - GUI) в библитеке компонент AWT (Abstract Window Toolkit – инструментарий абстрактного окна).

Компоненты AWT повторяют интерфейс исполняемой платформы без изменений.

JDK 1.1 (19 февраля 1997).

Наиболее значимые дополнения:

- Обширное изменение событий библиотеки AWT.
- В язык добавлены внутренние классы.

- JavaBeans (классы в языке Java, написанные по определённым правилам. Они используются для объединения нескольких объектов в один для удобной передачи данных).
- JDBC (Java Data Base Connection - соединение с базами данных. Платформенно-независимый промышленный стандарт взаимодействия Java-приложений с различными СУБД.
- RMI (Remote Method Invocation - программный интерфейс вызова удаленных методов).
- Ограниченная рефлексия (модификация во время выполнения невозможна, есть только наблюдение собственной структуры)

В настоящее время версия JDK 1.0.2 уже не используется.

J2SE 1.2 (8 декабря 1998).

Следующей базовой версией Java стала версия Java 2, символизировавшая собой второе поколение. Первой версии Java 2 был присвоен номер 1.2. С появлением версии 2, SUN Microsystems стала выпускать Java в виде пакета J2SE (Java 2 Platform Standard Edition - Стандартная версия платформы Java 2) и теперь номера версий указываются применительно к этому продукту.

Java 2, или Java 2.0 — дальнейшее развитие и усовершенствование спецификации исходного стандарта языка и платформы Java, на который теперь принято ссылаться как на Java 1.0. В настоящее время спецификация платформы Java 2 продолжает интенсивно развиваться и обогащаться, пополняясь новыми возможностями, особенно из за конкуренции с платформой.Net, перенявшей у Java ряд ключевых особенностей.

Основными усовершенствованиями Java 2 по сравнению с Java 1.0 являются:

- Включена библиотека компонент Swing.
- Коллекции.
- Policy файлы.
- Цифровые сертификаты пользователя.
- Библиотека Accessibility.
- Java 2D.
- Поддержка технологии drag-and-drop.
- Полная поддержка Unicode, включая поддержку ввода на японском, китайском и корейском языках.
- Поддержка воспроизведения аудиофайлов нескольких популярных форматов.
- Полная поддержка технологии CORBA.

- Включение в JDK для Java 2 JIT-компилятора, улучшенная производительность.
- Усовершенствования инструментальных средств JDK, включая поддержку профилирования Java-программ.

J2SE 5.0 (30 сентября 2004).

В данной версии разработчики внесли в язык целый ряд принципиальных дополнений:

- Перечислимые типы (enum). Ранее отсутствовавшие в Java типы оформлены по аналогии с C++, но при этом имеют ряд дополнительных возможностей.
  - Перечислимый тип является полноценным классом Java, то есть может иметь конструктор, поля, методы, в том числе скрытые и абстрактные.
  - Перечисление может реализовывать интерфейсы.
  - Для перечислений имеются встроенные методы, дающие возможность получения значений типа по имени, символьных значений, соответствующих именам, преобразования между номером и значением, проверки типа на то, что он является перечислимым.
- Аннотации — возможность добавления в текст программы метаданных, не влияющих на выполнение кода, но допускающих использование для получения различных сведений о коде и его исполнении. Одновременно выпущен инструментарий для использования аннотированного кода. Одно из применений аннотаций — упрощение создания тестовых модулей для Java-кода.
- Методы с неопределённым числом параметров.
- Autoboxing/Unboxing — автоматическое преобразование между скалярными типами Java и соответствующими типами-врапперами (например, между int — Integer). Наличие такой возможности упрощает код, поскольку исключает необходимость в выполнении явных преобразований типов в очевидных случаях.
- Разрешён импорт статических переменных.
- В язык введён цикл по коллекции объектов (итератор, foreach).

Java SE 6.0 (11 августа 2009).

Номера 2 в версии теперь нет. Изменения:

- В имеющихся корневых сертификатах (добавлен один новый корневой сертификат и удалено три корневых сертификата от Entrust, добавлено по три новых сертификата от Keynectis и Quovadis)
- Добавлена новая запись в Blacklist.

- Обнаружена (но не исправлена) ошибка, связанная с отладкой, которая при определённых условиях приводит к игнорированию точек остановки
- Исправлены обнаруженные уязвимости безопасности и другие ошибки, полный список которых можно посмотреть на официальном сайте (на английском).

Java SE 7.0.

Java 7 (кодовое имя Dolphin) это предстоящее крупное обновление Java. Проект Dolphin начался в августе 2006 и в предварительном порядке планируется к выпуску в конце 2010 года. Срок разработки состоит из 10 этапов.

1.2. Библиотеки классов SDK

SDK (Software Development Kit, или «devkit») - комплект средств разработки, который позволяет специалистам по программному обеспечению создавать приложения для определённого пакета программ, программного обеспечения базовых средств разработки, аппаратной платформы, компьютерной системы, видеоигровых консолей, операционных систем и прочих платформ.

Программист, как правило, получает SDK непосредственно от разработчика целевой технологии или системы. Часто SDK распространяется через Интернет. Многие SDK распространяются бесплатно для того, чтобы поощрить разработчиков использовать данную технологию или платформу.

1.3. Графичекий инструментарий

1.3.1. AWT

AWT (Abstract Window Toolkit – инструментарий абстрактного окна) - это исходная платформо-независимая оконная библиотека графического интерфейса (Widget toolkit) языка Java. Сейчас AWT является частью Java Foundation Classes (JFC) - стандартного API для реализации графического интерфейса в Java-программе.

Когда Sun Microsystems впервые выпустила Java в 1995 году, виджеты AWT предоставляли тонкий уровень абстракции над основным родным пользовательским интерфейсом операционной системы. Например, создание флажка AWT заставляет AWT напрямую вызвать более низкоуровневую нативную подпрограмму, которая и создает флажок. Однако, флажок (check box) на Microsoft Windows это не совсем то же, как флажок на Mac OS или на различных видах Unix. Некоторые разработчики предпочитают эту модель, поскольку она обеспечивает высокую степень соответствия основному оконному инструментарию и беспрепятственную интеграцию с родной приложений. Другими словами, GUI программа, написанная с использованием AWT, выглядит как родное приложе-

ние Microsoft Windows, будучи запущенной на Windows, и в то же время как родное приложение Apple Macintosh, будучи запущенным на Mac, и т.д.. Однако, некоторым разработчикам не нравится эта модель, потому что они предпочитают, чтобы их приложения выглядели одинаково на всех платформах.

AWT использует только стандартные элементы ОС для отображения, т.е. для каждого элемента создается отдельный объект ОС (окно), в связи с чем, AWT не позволяет создавать элементы произвольной формы (возможно использовать только прямоугольные компоненты.

В J2SE 1.2 виджеты AWT были в значительной степени заменены аналогичными из Swing. В дополнение к предоставлению более богатого набора элементов интерфейса пользователя, Swing рисует свои собственные виджеты (с помощью Java 2D для вызова низкоуровневых подпрограмм местной графической подсистемы), вместо того чтобы полагаться на высоком уровне модуля пользовательского интерфейса операционной системы. Swing обеспечивает возможность использования либо системного «look and feel», который использует родной «look and feel» платформы, либо кросс-платформенный внешний вид («Java Look and Feel»), который выглядят одинаково на всех платформах. Тем не менее, Swing использует AWT для взаимодействия с родной оконной системой.

AWT предоставляет два уровня API:

- Общий интерфейс между Java и родной системой, используемый для управления окнами, события, менеджеры компоновки. Этот API является основой программирования Java GUI и используется также Swing. Он содержит:
  - Интерфейс между родной оконной системой и Java приложением;
  - The core of the GUI event subsystem;
  - Некоторые менеджеры компоновки;
  - Интерфейс к устройствам ввода, таким как мышь и клавиатура; и
  - Пакет java.awt.datatransfer package для использования с буфером обмена и Drag and Drop.
- Базовый набор виджетов графического интерфейса, таких как кнопки, текстовые поля (text box) и меню. Она также предоставляет AWT Native Interface, который позволяет библиотекам в нативном коде рисовать непосредственно на Canvas.

1.3.2. Swing

Начиная с версии Java 1.2 Swing включён в JRE (Java Runtime Environment - среду выполнения Java).

Swing — библиотека для создания графического интерфейса на языке Java. Swing был разработан компанией Sun Microsystems. Он содержит ряд графических компонентов (Swing widgets), таких как кнопки, поля ввода, таблицы и др.

Swing относится к библиотеке классов JFC (Java Foundation Classes), которая представляет собой набор библиотек для разработки графических оболочек. К этим библиотекам относится и AWT.

Swing предоставляет более гибкие интерфейсные компоненты, чем более ранняя библиотека AWT. В отличие от AWT компоненты Swing разработаны для одинаковой **кросс-платформенной** работы, в то время как компоненты AWT повторяют интерфейс исполняемой **платформы** без изменений. Основным минусом таких «легковесных» (Lightweight) компонентов является относительно медленная работа. Положительная сторона - универсальность интерфейса созданных приложений на всех платформах.

«Lightweight» означает, что компоненты Swing отрисовываются самими компонентами на поверхности родительского окна, без использования компонентов операционной системы. В отличие от «Тяжелых» компонентов AWT, в приложении Swing может иметься только одно окно, и все прочие компоненты отрисовываются на ближайшем родителе, имеющем собственное окно (например, на JFrame).

В приложении могут сочетаться Swing и AWT элементы, хотя это может порождать некоторые проблемы, в частности, компоненты AWT всегда перекрывают Swing элементы, а также закрывают собой всплывающие меню JPopupMenu и JComboBox.

Начиная с Java 6 Update 12, стало возможно смешивать виджеты Swing and AWT без проблем с порядком наложения.

1.3.3. SWT

SWT (Standard Widget Toolkit, произносится «свит») - библиотека с открытым исходным кодом для разработки ГИП на языке Java.

Разработана фондом Eclipse, лицензируется под Eclipse Public License, одной из лицензий открытого ПО.

SWT не является самостоятельной графической библиотекой, а представляет собой кросс-платформенную оболочку для графических библиотек конкретных платформ. В ней используются методы ГИП операционной системы.

SWT - альтернатива AWT и Swing (Sun Microsystems) для разработчиков, желающих получить привычный внешний вид программы в данной операционной

системе и избежать части проблем, связанных с переучиванием пользователей. Использование SWT делает Java-приложение более эффективным, но снижает независимость от операционной системы и оборудования, требует ручного освобождения ресурсов и в некоторой степени нарушает Sun-концепцию платформы Java.Виртуальная Java-машина

Основной задачей программиста является написание исходного текста программы на одном из языков программирования. Хороший программист, разрабатывая новую программу, не пишет весь код заново. Он старается использовать уже готовые программные коды (библиотеки), написанные как им самим, так и другими разработчиками. Если рассматривать эти библиотеки, как строительные блоки, то программист из них, как из кирпичей, строит здание – новую программу.

Набор программ и пакетов классов JRE (Java Runtime Environment) содержит все необходимое для выполнения байт-кодов, в том числе интерпретатор java (в прежних версиях облегченнный интерпретатор jre) и библиотеку классов. Это часть JDK, не содержащая компиляторы, отладчики и другие средства разработки.

Именно JRE или его аналог других фирм содержится в браузерах, умеющих выполнять программы на Java, операционных системах и системах управления базами данных.

Хотя JRE входит в состав JDK, фирма SUN распространяет этот набор и отдельным файлом

Программы на Java исполняются в два этапа:

- На первом Java-компилятор преобразует исходный код в промежуточный байт-код (сборка).
- На втором Java-интерпретатор исполняет байт-код на виртуальной Java-машине с использованием ее библиотек.

Сборка – это единица повторного использования кода, в которой поддерживается система управления версиями и заложена система управления безопасности программного обеспечения. Сборка подключается только на время исполнения кода. Файл сборки называется управляемым.

Сборка наряду с программным байт-кодом содержит метаданные и данные (ресурсы), необходимые при исполнении сборки.

2. Ассемблер Java

Основным форматом исполняемых файлов в архитектуре Java является формат файла класса, описанный в The Java™ Virtual Machine Specification, издан-

ной компанией Sun. Файл данного формата имеет имя, совпадающее с идентификатором класса (за исключением вложенных классов), и расширение.class.

Этот файл на этапе исполнения интерпретируется JVM - виртуальной машиной Java.

2.1. Система команд JVM

В JVM используются байтовые команды. Первый байт каждой команды JVM содержит код операции. Существует несколько типичных форматов, которые имеют большинство команд:

- только код операции,
- код операции и 1-байтный индекс,
- код операции и 2-байтный индекс,
- код операции и 2-байтное смещение перехода,
- код операции и 4-байтное смещение перехода.

Несколько команд используют другие форматы:

- 2 команды переменного размера - tableswitch и lookupswitch.
- Специальный префикс wide, который изменяет размер некоторых команд, заменяя 1-байтный индекс локальной переменной 2-байтным.

В “The Java Virtual Machine Specification” для каждой команды установлено свое мнемоническое обозначение.

Существует много групп аналогичных по выполняемому действию команд, работающих с различными типами данных. Например, команды iload, lload, aload, fload, dload выполняют функцию загрузки значений соответствующих типов из локальных переменных на вершину стека. Реализация таких команд может быть идентичной, но они различаются при проверке корректности байт-кода. Приняты следующие обозначения для типов данных, с которыми работают команды:

- i - int (также byte, short, char и boolean),
- l - long,
- f - float,
- d - double,
- a - ссылка на объект или массив.

Есть несколько команд, работающих только с типами char, byte и short.

Можно выделить несколько групп команд по назначению:

Команды загрузки и сохранения:

- Загрузка локальной переменной на стек: iload, iload_<n>, lload, lload_<n>, fload, fload_<n>, dload, dload_<n>, aload, aload_<n>;
- Сохранение значения с вершины стека в локальной переменной: istore, istore_<n>, lstore, lstore_<n>, fstore, fstore_<n>, dstore, dstore_<n>, astore, astore_<n>;
- Загрузка констант на стек: istore, istore_<n>, lstore, lstore_<n>, fstore, fstore_<n>, dstore, dstore_<n>, astore, astore_<n>;

Арифметические и логические команды:

- сложение: iadd, ladd, fadd, dadd;
- вычитание: isub, lsub, fsub, dsub;
- умножение: imul, lmul, fmul, dmul;
- деление: idiv, ldiv, fdiv, ddiv;
- остаток: irem, lrem, frem, drem;
- изменение знака: ineg, lneg, fneg, dneg;
- сдвиги и побитовые операции: ior, lor, iand, land, ixor, lxor, ishl, ishr, iushr, lshl, lshr, lush;
- сравнение: dcmpg, dcmpl, fcmpg, fcmpl, lcmg;
- инкремент локальной переменной: iinc.

Все эти команды, за исключением iinc, не имеют параметров. Они извлекают операнды с вершины стека и записывают результат на вершину стека. Команда *iinc* имеет два операнда - индекс локальной переменной и величину, на которую значение данной переменной должно быть изменено;

Команды преобразования типов:

- расширяющее: i2l, i2f, i2d, l2f, l2d, f2d;
- сужающее: i2b, i2c, i2s, l2i, f2i, f2l, d2i, d2l, d2f.

Команды работы с объектами и массивами:

- создание объекта: new;
- создание массива: newarray (примитивного типа), anewarray (ссылочного типа), multianewarray(многомерного);
- доступ к полям: getfield, putfield (для полей экземпляра), getstatic, putstatic (для статических полей);
- загрузка элемента массива на стек: baload (тип byte), caload (тип char), saload (тип short), iaload, laload, faload, daload, aaload;
- сохранение значения с вершины стека в элемент массива: bastore, castore, sastore, iastore, lastore, fastore, dastore, aastore;

- получение размера массива: arraylength;
- проверка типов: instanceof (возвращает на вершине стека логическое значение) и checkcast (генерирует исключение в случае несоответствия типа ссылки на вершине стека требуемому типу).

Команды манипуляций со стеком операндов:

- pop - удаление верхнего элемент стека;
- pop2 - удаление двух верхних элемента стека;
- dup, dup2, dup_x1, dup2_x1, dup_x2, dup2_x2 - дублирование элементов на вершине стека;
- swap - перемена местами двух верхних элементов стека.

Команды безусловной передачи управления:

- jsr, jsr_w, ret - вызов подпрограмм и возврат из них. Используются при компиляции блока finally;
- goto, goto_w - безусловный переход.

Команды условного перехода:

- ifeq, iflt, ifle, ifne, ifgt, ifge, ifnull, ifnonnull, if_icmpeq, if_icmpne, if_icmplt, if_icmpgt, if_icmple, if_icmpge, if_acmpeq, if_acmpne.

Команды вызова методов:

- invokevirtual - обычный вызов метода экземпляра с использованием механизма позднего связывания;
- invokestatic - вызов статического метода;
- invokeinterface - вызов метода интерфейса у объекта, реализующего данный интерфейс;
- invokespecial - вызов метода экземпляра без использования механизма позднего связывания. Используется для вызова конструкторов, методов суперкласса и private-методов.

Команды возврата из метода:

- return - возврат из метода, возвращающего void;
- ireturn, lreturn, freturn, dreturn, areturn - возврат значения соответствующего типа.

Команда генерации исключений: athrow.

Команды синхронизации (механизм блокировок):

- monitorenter - установить блокировку (войти в критическую секцию);
- monitorexit - освободить блокировку (выйти из критической секции).

2.2. Синтаксис языка JASM

Файл исходного текста на языке ассемблера для платформы Java (языке JASM) представляет собой текстовый файл, строки которого разделены последовательностью символов с кодами 10 (переход нп новую строку) и 13 (ввод).

Имя файла исходного текста и его расширение могут быть любыми, однако рекомендуется, чтобы имя совпадало с именем описанного в файле класса, а расширением было .jasm либо .jsm. Файл исходного текста состоит из предложений, разделенных точкой с запятой. Последнее предложение может не иметь в конце точки с запятой.

Комментарии отделяются знаком процента и распространяются до конца строки. Точки с запятой и знаки процента внутри строковых констант, ограниченных двойными кавычками, не имеют своего специального значения. Две идущие подряд двойные кавычки внутри строковой константы интерпретируются как одна двойная кавычка в строке. Любые последовательности пробельных символов (пробелов, табуляций, переводов строки и т.д.) интерпретируются как один пробел, если с обеих сторон от них находятся символы следующих видов: буквы латинского алфавита, цифры, знак подчеркивания, либо, в противном случае, игнорируются.

Исключение составляют пробельные символы в строковых константах и комментариях. Верхний и нижний регистр букв в идентификаторах, именах команд и других лексемах различается.

Каждый файл исходного текста компилируется в один файл класса.

2.2.1. Структура файла

В описании приняты соглашения:

- В квадратные скобки заключены необязательные элементы.
- В фигурные скобки - альтернативные варианты (разделены вертикальной чертой).
- В угловые суобки (< >)- нетерминальные символы.

Файл исходного текста должен иметь структуру:

```
[модификаторы_доступа] {class|interface} <имя_класса>;
[extends <базовый класс>;]
[implements <интерфейс_1>, <интерфейс_2>,..., <интерфейс_n>;]
[fields;
    <описания_полей>
]
```

```
[methods;
    <описания_методов>
]
```

Модификаторы_доступа - это слова public, final, abstract, super, соответствующие флагам прав доступа ACC_PUBLIC, ACC_FINAL, ACC_ABSTRACT, ACC_STATIC. Эти флаги устанавливаются в единицу тогда и только тогда, когда в объявлении класса присутствует соответствующее ключевое слово.

Класс может иметь несколько различных модификаторов доступа, разделенных пробелом (или любой другой последовательностью пробельных символов). Повторение одинаковых модификаторов в заголовке одного класса не допускается.

Когда класс не имеет флага ACC_INTERFACE, в его объявлении используется слово class, иначе используется ключевое слово interface. Все имена классов и интерфейсов записываются с указанием полного пути (пакетов, в которых эти классы содержатся). Имена пакетов и класса отделяются точкой, например, java.lang.String. В аргументах команд, там, где это необходимо, вместо полного имени текущего класса можно использовать символ «@».

Если базовый класс не указан (предложение extends отсутствует), то по умолчанию используется java.lang.Object. Интерфейсы - предки описываемого интерфейса записываются в секции implements.

Для идентификаторов - имен пакетов, классов, полей и методов, а также меток, используются следующие правила: они должны состоять из букв латинского алфавита любого регистра (регистр имеет значение), знаков подчеркивания и цифр, причем не должны начинаться с цифры. Настоятельно не рекомендуется использование идентификаторов, совпадающих с ключевыми словами языка Java, что может привести к некорректной компиляции, либо интерпретации файлов классов JVM. Два специальных имени <init> и <clinit> также рассматриваются как идентификаторы.

2.2.2. Структура поля

```
[модификаторы_доступа] <имя_поля>:<тип_поля> [=<начальное значение>];
```

Здесь модификаторы_доступа - следующие слова: public, protected, private, final, static, transient, volatile, соответствующие флагам доступа поля ACC_PUBLIC, ACC_PROTECTED, ACC_PRIVATE, ACC_FINAL, ACC_STATIC, ACC_TRANSIENT, ACC_VOLATILE.

Повторение одинаковых модификаторов доступа в объявлении одного поля и сочетания модификаторов, соответствующие запрещенным сочетаниям фла-

гов доступа (см. The Java Virtual Machine Specification), вызывают ошибку времени компиляции.

Поля интерфейса обязательно должны быть объявлены с модификаторами public, static и final. Имя_поля - корректный идентификатор. Тип_поля - имя класса либо имя примитивного типа (имена примитивных типов совпадают с соответствующими ключевыми словами языка Java - byte, short, int, long, char, float, double, boolean).

Начальное значение может быть задано только для статического поля, если оно указано, то у поля создается атрибут ConstantValue. Начальное значение может быть целочисленной, вещественной, логической либо символьной константой для полей соответствующих типов. Вещественная константа может быть записана в десятичной либо экспоненциальной форме, в формате вещественных чисел, принятом в Java. Символьные константы заключаются в апострофы. Кроме того, может быть указан код символа как обычное целое число. Логические константы записываются в виде слов true и false.

2.2.3. Описание метода

В общем случае имеет вид:

```
[<модификаторы_доступа>]
<имя_метода>(<тип_параметра_1>,<тип_параметра_2>,...,<тип_параметра_n>)
:<тип_возвращаемого_значения> [throws <класс_исключения_1>,...,
<класс_исключения_n>];
% для методов с модификатором abstract нижележащая часть отсутствует
    maxstack <число>;
    maxlocals <число>;
    [<метка_1>:]
        <команда_1>;
        ...
    [<метка_n>:]
        <команда_n>;
    [
    protected_blocks;
        {<класс_исключения>|finally} <метка>: <метка> > <метка>;
        ...
        {<класс_исключения>|finally} <метка>: <метка> > <метка>;
    ]
    end;
```

Здесь модификаторы доступа - ключевые слова: public, protected, private, static, final, abstract, соответствующие следующим флагам доступа метода:

ACC_PUBLIC, ACC_PRIVATE, ACC_PROTECTED, ACC_STATIC, ACC_FINAL, ACC_ABSTRACT. Повторение одинаковых модификаторов доступа в заголовке одного метода и сочетания модификаторов, соответствующие запрещенным сочетаниям флагов доступа (см. The Java Virtual Machine Specification), вызывают ошибку времени компиляции.

Методы интерфейса обязательно должны быть объявлены с модификаторами public и abstract. Имя_метода - корректный идентификатор, либо <init> или <clinit> для конструкторов и статических инициализаторов. Типы параметров и тип возвращаемого значения должны быть именами классов, либо именами примитивных типов, принятыми в языке Java (byte, short, int, long, char, float, double, boolean). Кроме того, тип возвращаемого значения может быть указан как void. После ключевого слова throws в заголовке метода могут быть перечислены через запятую имена классов исключений, генерируемых методом.

Для методов, не являющихся абстрактными, после заголовка обязательно записываются предложения maxstack и maxlocals, в которых указывается размер стека операндов и области локальных переменных метода (в четырехбайтных ячейках). Затем следует код метода в виде последовательности команд, разделенных точками с запятыми.

Каждой команде может предшествовать метка, отделяемая от нее двоеточием. Метка должна быть корректным. Каждая команда может иметь не более одной метки, и каждая метка должна предшествовать той или иной команде. Однако, имеется специальная псевдокоманда none, для которой не генерируется какой-либо код (пустая команда). Ее можно использовать, если необходимо расположить более одной метки у одной команды или поместить метку в конец метода.

После ключевого слова protected_blocks могут быть перечислены защищенные блоки (обработчики исключений) метода. Описание каждого защищенного блока состоит из имени класса исключения или ключевого слова finally и трех меток, разделенных символами ‘:’ и ‘>’. Первая из них указывает на начало защищенного блока, вторая на его конец, третья - на место в коде метода, куда переходит управление при возникновении исключения или при выходе из защищенного блока в случае finally.

Используемые в коде мнемонические имена команд совпадают с принятыми в The Java Virtual Machine Specification. Однако, как исключение, префикс wide не рассматривается как отдельная команда, вместо этого команды, его имеющие, записываются как wide_<имя_команды>. Форматы записи основных команд:

- <мнемоническое_имя>;
- <мнемоническое_имя> <метка>; Такую форму имеют команды перехода.

- <мнемоническое_имя> <целое число>; Число должно удовлетворять ограничениям конкретной команды:
- <мнемон._имя> {<полное_имя_класса>|@}::<имя_поля>:<тип_поля>;
- <мнемоническое_имя> <полное_имя_класса>;
- <мнемон._имя> <целое_число_индекс_переменной> <целое_число>;
- <мнемоническое_имя> <тип> <константа>;

2.3. Этапы работы компилятора

При каждом запуске компилятора обрабатывается один файл исходного текста на языке ассемблера для платформы Java. Компилятор принимает два аргумента командной строки: имя файла исходного текста и имя создаваемого файла класса (явное указание расширения.class обязательно). В случае, если выходной файл уже существует, он перезаписывается без предупреждения. В случае синтаксической или иной ошибки на консоль выводится соответствующее сообщение.

Можно выделить несколько основных этапов компиляции (проходов):

- Чтение исходного файла. При этом он разбивается на предложения, разделенные точками с запятыми, также выбрасываются комментарии;
- Разбор исходного текста. При последовательном переборе списка предложений выделяются синтаксические конструкции. При разборе используется лексический анализатор, разделяющий предложения на лексемы. На основании выделенных синтаксических конструкций генерируется внутреннее представление программы, имеющее вид древовидной структуры данных, корнем которой является представление класса в целом, узлами первого уровня - объекты, соответствующие методам, полям, элементам Constant Pool и т.д.;
- Замена номеров меток соответствующими смещениями в коде методов;
- Генерация байт-кода методов как массивов байт;
- Генерация файла класса на основании внутреннего представления программы.

3. ИСР для Java

3.1. Основы ИСР для Java

В настоящее время для создания программного обеспечения (ПО) используются ИСР - интегрированные среды разработки (IDE – Integrated Development Environment), в которых поддерживается технология быстрой разработки RAD (Rapid Application Development). Для Java доступны две свободные ИСР:

- Eclipse.
- NetBeans.

Обе ИСР предназначены для разработки консольных приложений, приложений для ОС с графическим интерфейсом, DLL и др.

3.1.1. ИСР Eclipse

Первоначально Eclipse разрабатывалась фирмой IBM как преемник ИСР от IBM - VisualAge, в качестве корпоративного стандарта ИСР для разработки на разных языках под платформы IBM. По сведениям IBM проектирование и разработка стоили 40 миллионов долларов. Авторы проекта:

Дэн Ингаилс, Джон О'Коффи - Eclipse

Исходный код был полностью открыт и сделан доступным после того, как Eclipse был передан для дальнейшего развития независимому от IBM сообществу. Развивается и поддерживается консорциумом Eclipse Foundation, в который входят компании Borland, IBM, Merant,QNX Software Systems, Rational Software, RedHat, SuSE, и TogetherSoft.

Проект Eclipse Platform. получил распространение в виде открытого исходного кода согласно Универсальной публичной лицензии (Common Public License).

Основой Eclipse является платформа расширенного клиента (RCP - Rich Client Platform). В её состав входят:

- Ядро платформы (загрузка Eclipse, запуск модулей).
- Графический интерфейс пользователя (ГИП - GUI) в вариантах AWT, Swing и SWT. AWT – устаревший набор компонент.
- Swing эмулирует графические компоненты. Он создает очень длинные коды и работает медленнее SWT, пока поддерживается, но в новых проектах не употребляется.
- SWT использует для графических компонентов функции операционной системы, поэтому работает эффективнее, чем Swing.
- JFace (файловые буферы, работа с текстом, текстовые редакторы).
- Рабочая среда Eclipse (панели, редакторы, проекции, мастеры).

Гибкость Eclipse обеспечивается за счёт подключаемых модулей, благодаря чему в ИСР возможна разработка приложений не только на Java, но и на других языках, таких как C/C++, Perl, Groovy, Ruby, Python, PHP, Erlang, компонентный Паскаль, Zonnon и прочих.

В большинстве реализаций Eclipse интерфейс и справка на английском языке.

3.1.2. ИСР NetBeans

Разработка ИСР NetBeans началась в 1996 г. под названием Xelfi (игра букв на основе Delphi) в качестве проекта студентов под руководством Романа Станека по на Факультете Математики и Физики Карлова Университета в Праге.

В 1997г. начат выпуск коммерческих версий ИСР NetBeans до передачи всех прав на нее корпорации Sun Microsystems в 1999г. В 2000г. Sun открыла исходные коды ИСР NetBeans. Сейчас спонсор этой ИСР – Oracle.

В большинстве реализаций NetBeans интерфейс на русском, но справка на английском языке.

3.2. ИСР для Java подробнее

3.2.1. ИСР Eclipse

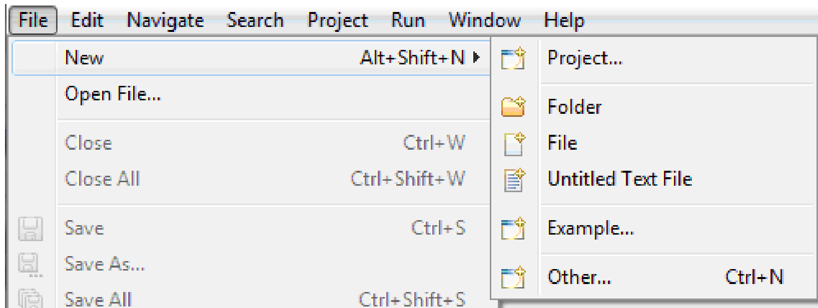
После первого запуска ИСР появляется окно ИСР. Вид окна зависит от типа проекта. Для каждого проекта пользователь может согласиться с шаблоном по умолчанию, но может выбрать свое представление, выбрав его из списка Resource perspective в правой части панели инструментов. По умолчанию для пустого проекта выбран шаблон, в котором отображаются:

- Строка заголовка с именем ИСР (Resource – Eclipce).
- Строка главного меню ИСР.
- Панели инструментов для быстрого выполнения часто используемых команд.
- Project Explorer (Обозреватель проекта). Окно слева вверху. В нем отображается иерархия компонент проекта. При первом запуске окно пустое.
- Outline – окно внешних ссылок. Окно слева внизу.
- Рабочее поле проекта. Окно вверху справа.
- Окно информационных сообщений с закладками по типам информации. Расположено справа внизу. В примере закладки следующие: Tasks (задачи), Properties (свойства), JavaBeans (используется с визуальным редактором Visual Editor для Java).

Пункты главного меню:

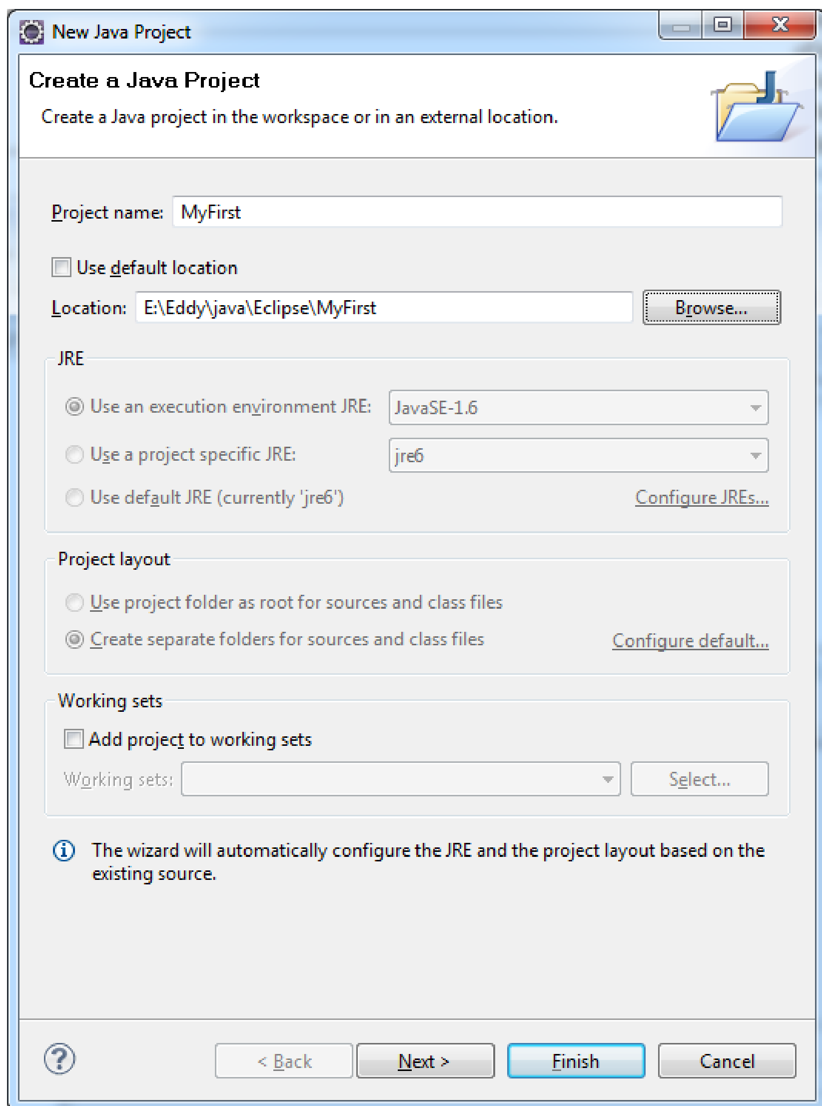
- Файл (File). - стандартные операции работы с файлами.
- Правка (Edit). - стандартные операции редактирования.
- Просмотр (Navigate) - команды выбора компонент проекта для просмотра.
- Поиск (Search). - стандартные операции поиска.
- Проект (Project).
- Выполнить (Run). Запуск.
- Окна (Windows). Выбор окон для отображения.
- Справка (Help). На английском языке.

Для начала работы с проектом нужно его создать. Для этого исполняется команда File=>New. Отображается выпадающее меню для выбора типа проекта.



Выбираем Project. Отображается окно выбора типа проекта. В нем выбираем Java Project. Это консольное приложение. Отображается окно со свойствами проекта. В нем должны быть заданы:

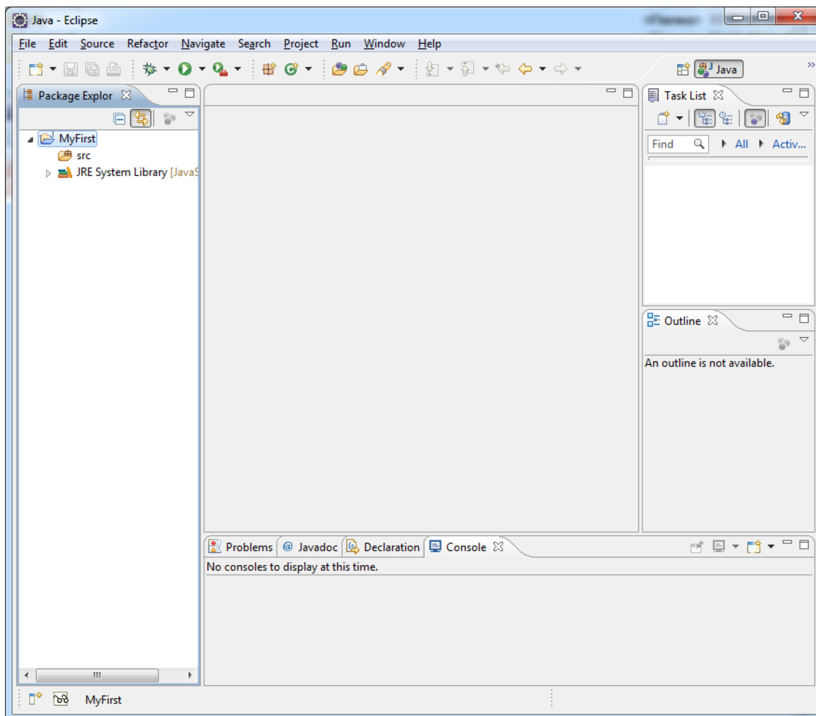
- Имя проекта (например, MyFirst).
- Путь к папке для файлов проекта (например, E:\Eddy\java\Eclipse). Чтобы выбрать место размещения с помощью браузера нужно снять флаг использования места по умолчанию (Use default location).



В итоге генерируется окно проекта, в котором в окне браузера проекта отображаются его компоненты

- Исходник (Src). Это файлы проекта..

- JRE System Library. Библиотека классов для выбранного проекта. Выбирается автоматически по типу проекта.



Появились новые пункты главного меню:

- Исходный код (Source) – операции для создания кода проекта. Они позволяют вносить в код проекта фрагмент, определяемый командой в выпадающем меню
- Реорганизатор кода (Refactor) - операции по изменению оформления имеющегося кода программы без изменения ее поведения.

Ниже просмотр команд пункта Source.

Операция	Описание
Toggle Comment	Переключатель. Отображает или скрывает комментарий. Текст комментария не трогается.
Add Block Comment	Добавляет блок комментария для выделенного фрагмента кода.
Remove Block Com-	Удаляет блок комментария для выделенного фраг-

ment	мента кода.
Generate Element Comment	Добавляет комментарий к выделенному элементу кода.
Shift Right	Сдвиг вправо. Добавляет уровень иерархии для выделенного фрагмента кода.
Shift Left	Сдвиг влево. Уменьшает уровень иерархии для выделенного фрагмента кода.
Correct Indentation	Корректирует иерархию для выделенного фрагмента кода.
Format	Форматизация выделенного фрагмента кода.
Format Element	Форматизация выделенного элемента кода.
Add Import	Добавить объявления импорта для выделенного фрагмента кода.
Organize Imports	Коррекция объявлений импорта для выделенного фрагмента кода. Ненужные удаляются.
Sort Members	Сортировка членов типа.
Clean Up	Очистка кода. Удаление лишнего.
Override/Implement Methods	Открывает диалог “Override Method dialog”, который позволяет перезагрузить или реализовать метод текущего типа.
Generate Getter and Setter	Открывает диалог “Generate Getters and Setters dialog”, который позволяет перезагрузить или реализовать метод текущего типа.
Generate Delegate Methods	Открывает диалог “ Generate Delegate Methods”, который позволяет создать метод делегата для полей текущего типа.
Generate hashCode() and equals()	Открывает методы Generate hashCode() и equals(), что позволяет сгенерировать хэш-код и методы выравнивания хэш-таблиц для текущего типа.
Generate toString()	Открывает диалог “Generate toString() dialog”, который запускает и управляет процессом преобразования текущего типа в строку.
Generate Constructor using Fields	Добавляет конструкторы, которые инициализируют поля текущего выделенного типа.
Add Constructor from SuperClass	Добавляет конструкторы, которые определены в суперклассе.
Surround With	Обрамляет выделенные инструкции шаблоном кода (пунктирный прямоугольник).
Externalize Strings	Открывает мастера экстернализации строк. Он позволяет заменить все строки в коде, используя инструкции доступа к файлу свойств.

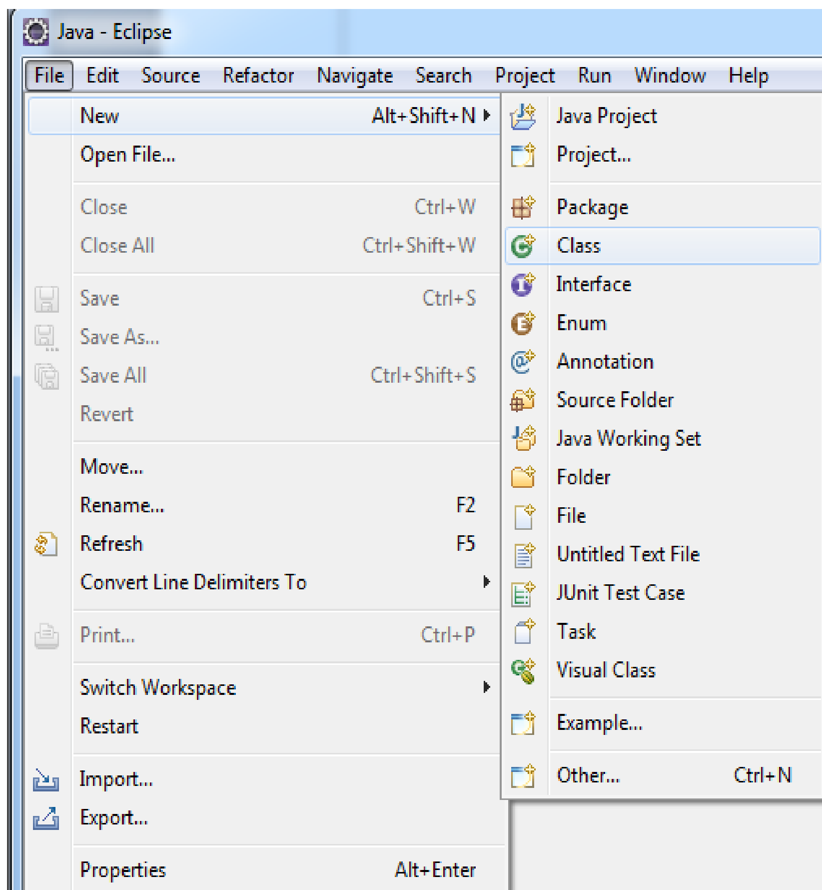
Find Broken Externalized Strings	Поиск разбитых строк экстернализации в выбранном файле свойств.

Ниже просмотр команд пункта Refactor.

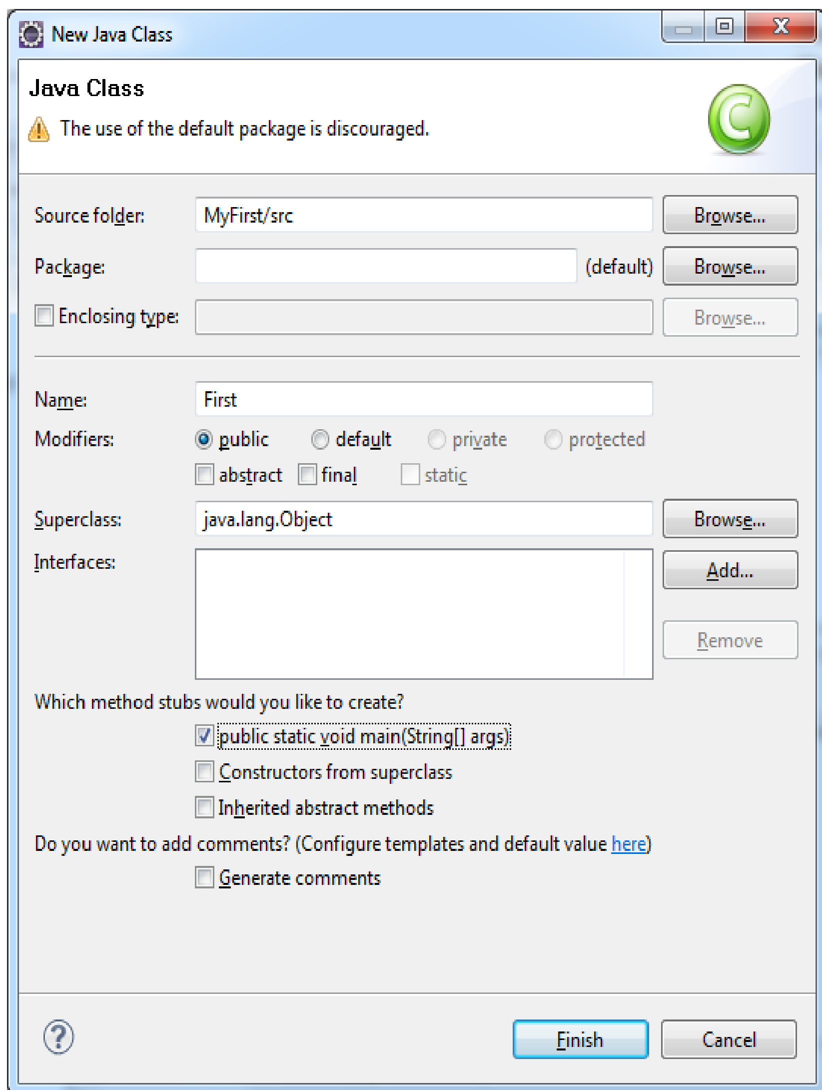
Операция	Описание
Rename	Переименование выбранного элемента и всех ссылок на него.
Move	Удаляет выбранные элементы и все ссылки на них.
Change Method Signature	Изменяет сигнатуру выделенного метода: имена, типы, порядок следования параметров, а также обновляет ссылки на них.
Extract Method	Создает новый метод, включающий выделенные инструкции или выражения.
Extract Local Variable	Создает новую переменную, связанную с выделенным выражением.
Extract Constant	Создает статическое поле, связанное с выделенным выражением, и подставляет ссылку не него.
Inline	Локальные переменные, методы и константы в режиме Inline (на языке ассемблера).
Convert Anonymous Class to Nested	Превращает анонимные классы во вложенные.
Move Type to New File	Создает новый скомпилированный Java модуль для выбранного типа и обновляет все ссылки на него.
Convert Local Variable to Field	Преобразует локальные переменные в поля.
Extract SuperClass	Создает суперкласс из выделенных типов.
Extract Interface	Создает новый интерфейс с множеством методов.
Use Supertype Where Possible	Заменяет появления типа на один из его супертипов там, где это возможно.
Push Down	Перемещает множество методов и полей из класса в подкласс.
Pull Up	Перемещает метод или поле из класса в суперкласс.
Extract Class	Заменяет множество полей объектом контейнер.
Introduce Parameter Object	Перемещает множество выделенных параметров в новый класс и обновляет все вызовы.
Introduce Indirection	Создает статический непрямой метод делегирования к выделенному методу.
Introduce Factory	Создает новый метод Factory, который вызывается выделенным конструктором и возвращает созданный объект.

Introduce Parameter	Заменяет выражение со ссылкой на новый параметр метода и обновляет все вызова метода.
Encapsulate Field	Заменяет ссылки на поля, используя методы getter и setter.
Generalize Declared Type	Разрешает использовать выбор супертипа для ссылки на текущий тип.
Infer Generic Type Arguments	Заменяет появления генерических типов параметрическими.
Migrate JAR File	Миграция JAR файла проекта в новую версию.
Create Script	Создает сценарий реорганизаций, которые были сделаны в проекте
Apply Script	Применяет сценарий реорганизаций.
History	Браузер рабочего пространства истории реорганизаций.

Создадим в проекте класс командой File=>New=>Class. Это будет наша программа.



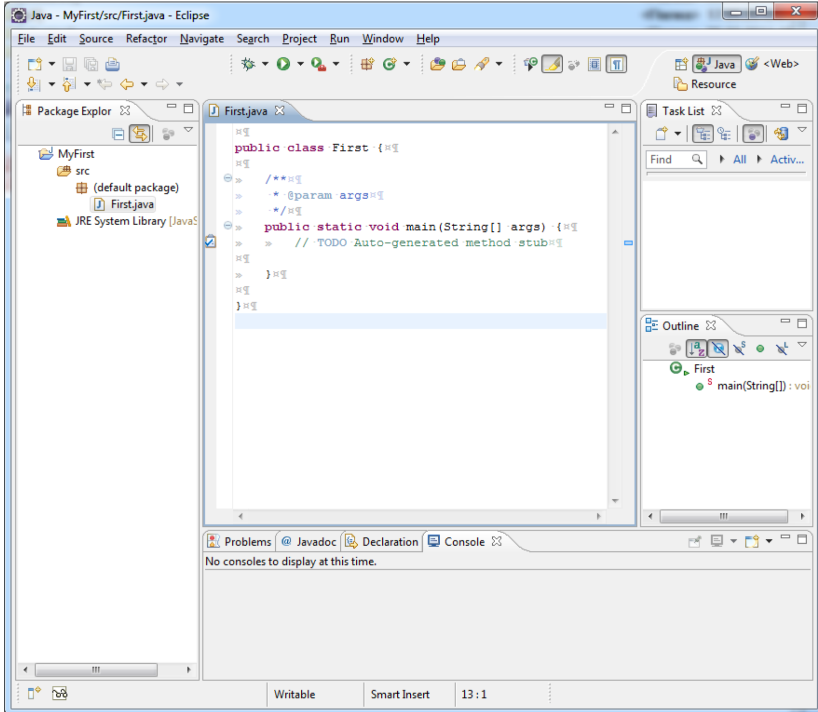
Теперь отображается окно со свойствами класса. В нем задаем имя класса First и флаг задания метода main().



Теперь в редакторе кода появился шаблон кода. Окно приобрело имя First.java. Программа – это класс с именем First. Внутри программы определен главный метод с именем main(). Исполнение программы начинается с его запуска.

В коде программы по умолчанию вставлены строки:

- Java комментария (в парных скобках /** и */). По нему можно автоматически построить справку (Help) к проекту.
- Текстового комментария в конце строк (с символами //).
- Пустые строки (отображены символами ¤q).



В метод main нужно добавлять свой функционал. Добавим команду вывода в консоль фразы Proba командой

System.out.println ("Proba");

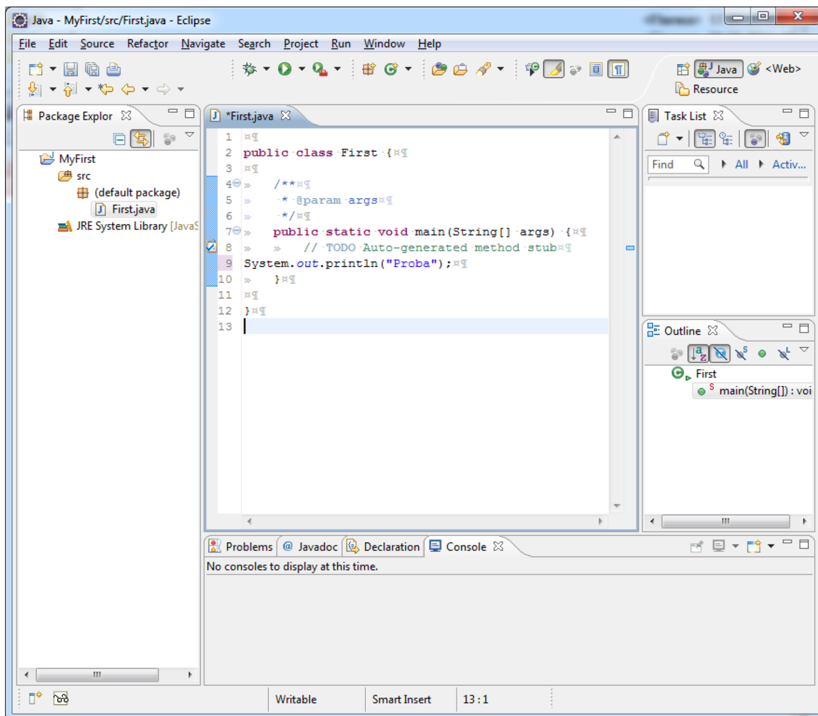
При наборе команды используем интерактивный подсказчик. После набора слова System и точки появится выпадающее меню.

.

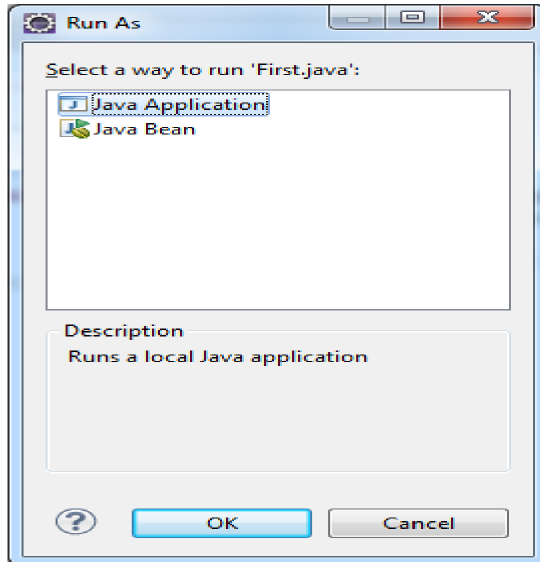
В нем ищем нужное действие. Если его не видно, то набираем следующую букву и ищем снова. Найденный метод выбираем. В качестве аргумента выбранного метода вводим строку "Proba". Если при наборе кода допускается ошибка, то ошибочный фрагмент кода подчеркивается волнистой линией. При

подведении курсора к месту ошибки во всплывающем окне ошибка описывается.

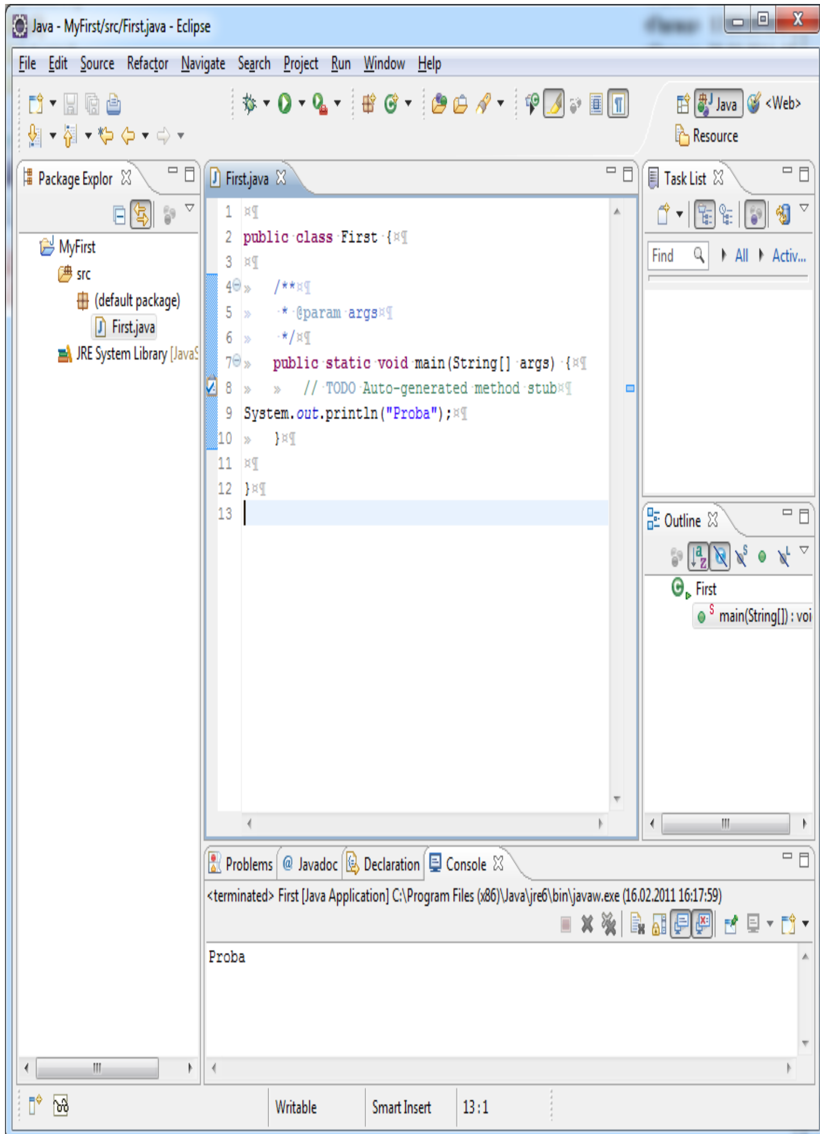
При запуске готовой программы файлы проекта должны сохраняться. При первом запуске в окне запуска вы получите возможность установить это по умолчанию. Теперь при любом новом прогоне обновленные файлы будут переписываться в месте размещения.



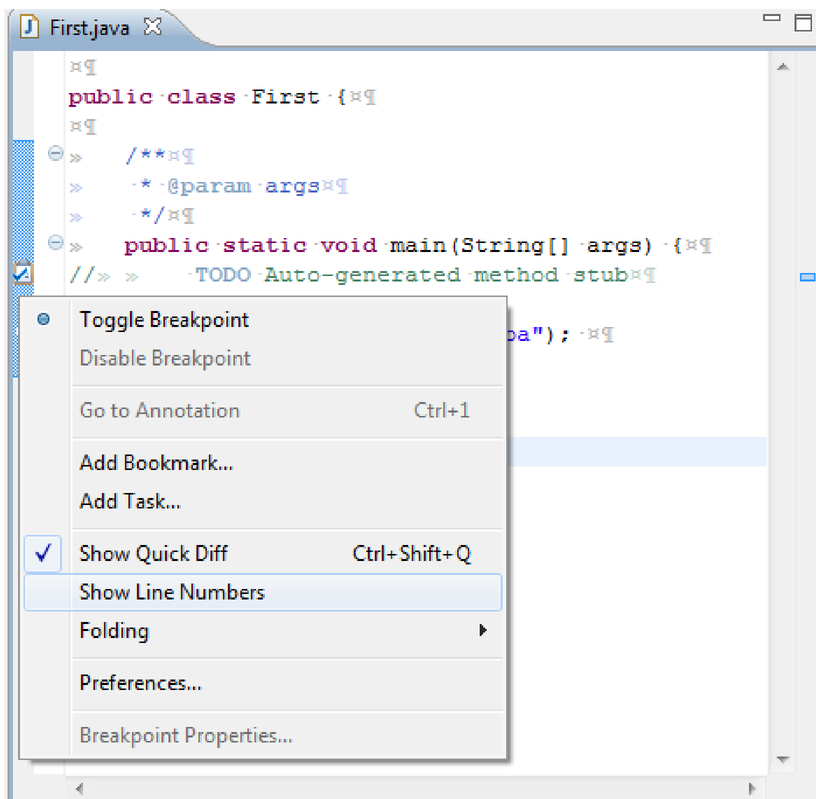
При прогоне возможен запрос способа прогона. Нужно выбрать Java Application.



При запуске в окне сообщений автоматически отображается закладка консоли, где и отображается результат Proba.



При больших программах желательно видеть номера строк кода. Если номера не отображаются, то для активизации нумерации строк нужно кликнуть правой кнопкой мыши по линейке слева от кода. В выпадающем меню выбрать команду Show Line Number.

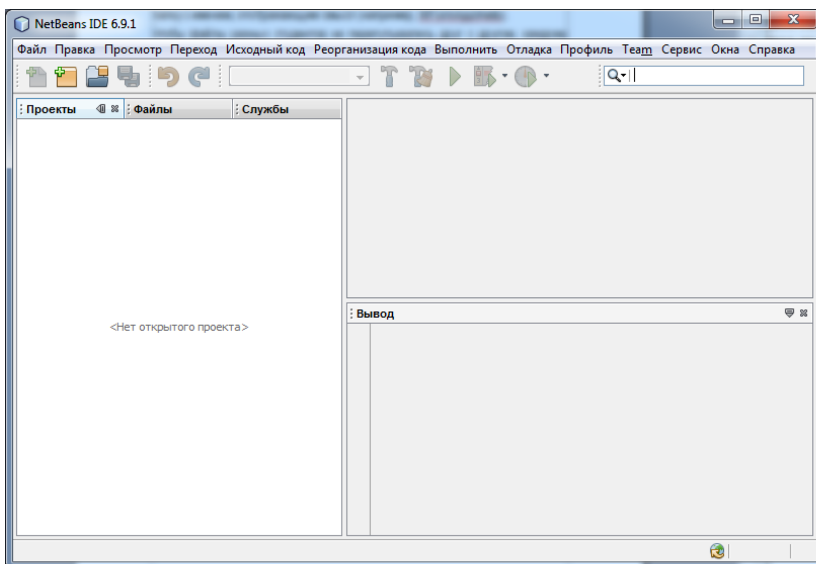


Это риведет к отображению в редакторе номеров строк.

3.2.2. Основы ИСР NetBeans

Окно ИСР пустого проекта содержит:

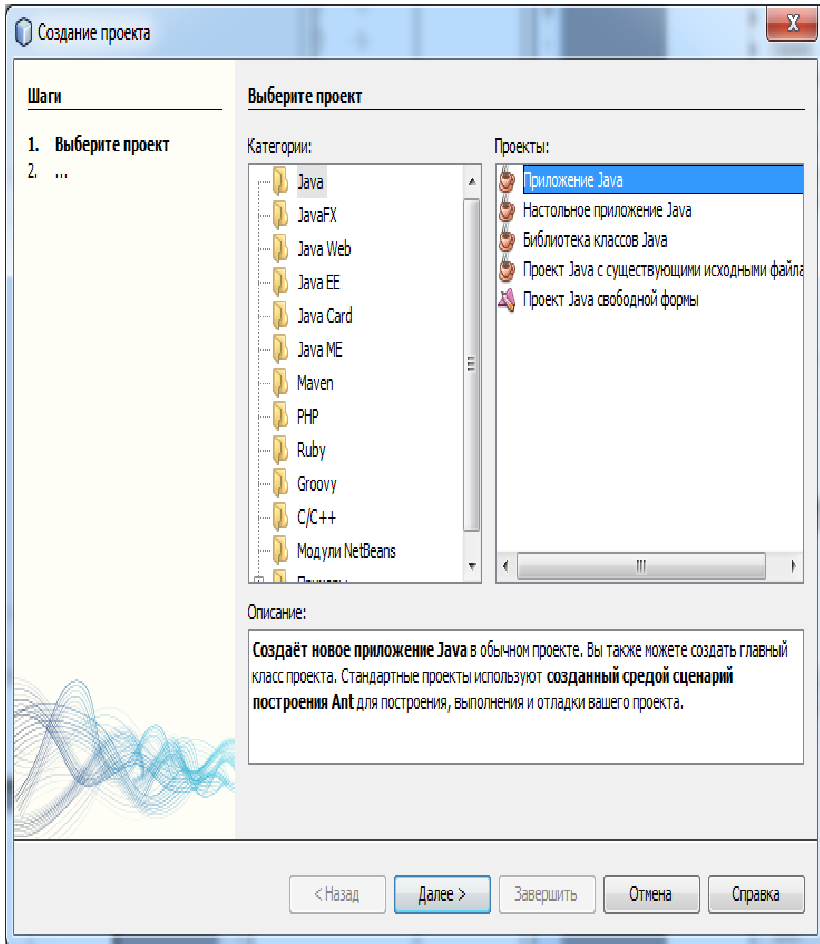
- Строку заголовка с именем ИСР.
- Главное меню, может быть на русском языке.
- Панель инструментов.
- Слева окно иерархии решения с вкладками – Проекты, Файлы, Службы.
- Справа вверху рабочее поле проекта.
- Справа внизу окно для вывода результатов.



Пункты главного меню:

- Файл (File) - стандартные операции работы с файлами.
- Правка (Edit) - стандартные операции редактирования.
- Просмотр (Navigate) - команды выбора компонент проекта для просмотра.
- Переход.
- Исходный код (Source) - команды выбора фрагментов кода..
- Реорганизация кода (Refactor) – операции по изменению оформления имеющегося кода программы без изменения ее поведения.
- Выполнить (Run). Запуск.
- Отладка.
- Профиль.
- Team. Связь с сервером команды разработчиков.
- Сервис.
- Окна (Windows). Выбор окон для отображения.
- Справка (Help). На английском языке.

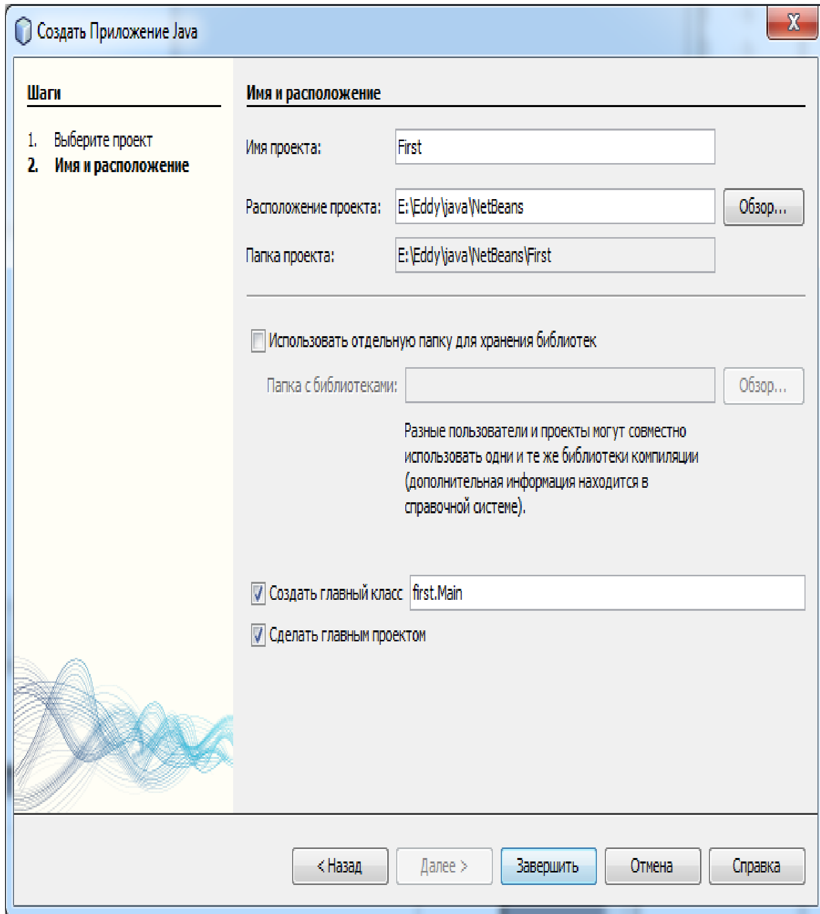
Для начала работы с проектом нужно его создать. Для этого исполняется команда Файл=>Создать проект. Отображается выпадающее меню для выбора типа проекта.



В нем нужно выбрать категорию проекта и тип проекта. Выбираем категорию Java и тип - Приложение Java. Это консольное приложение.

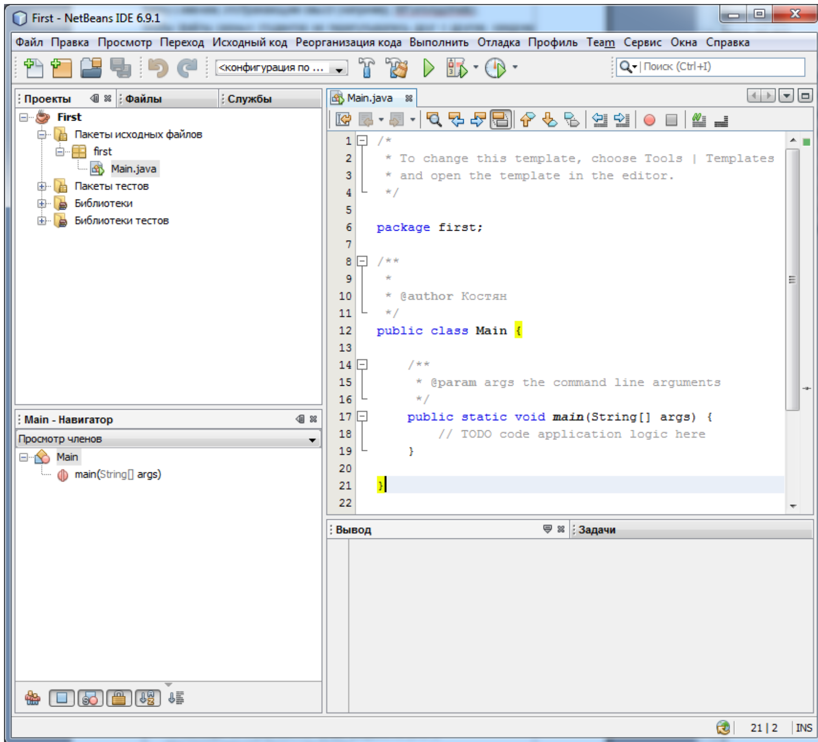
Отображается окно со свойствами проекта. В нем нужно задать:

- Имя проекта (например, First).
- Расположение проекта (например, E:\Eddy\java\NetBeans).
- Установить флаг - Создать главный класс.
- Установить флаг - Сделать главным проектом.



В итоге генерируется окно проекта, в котором отображаются его компоненты:

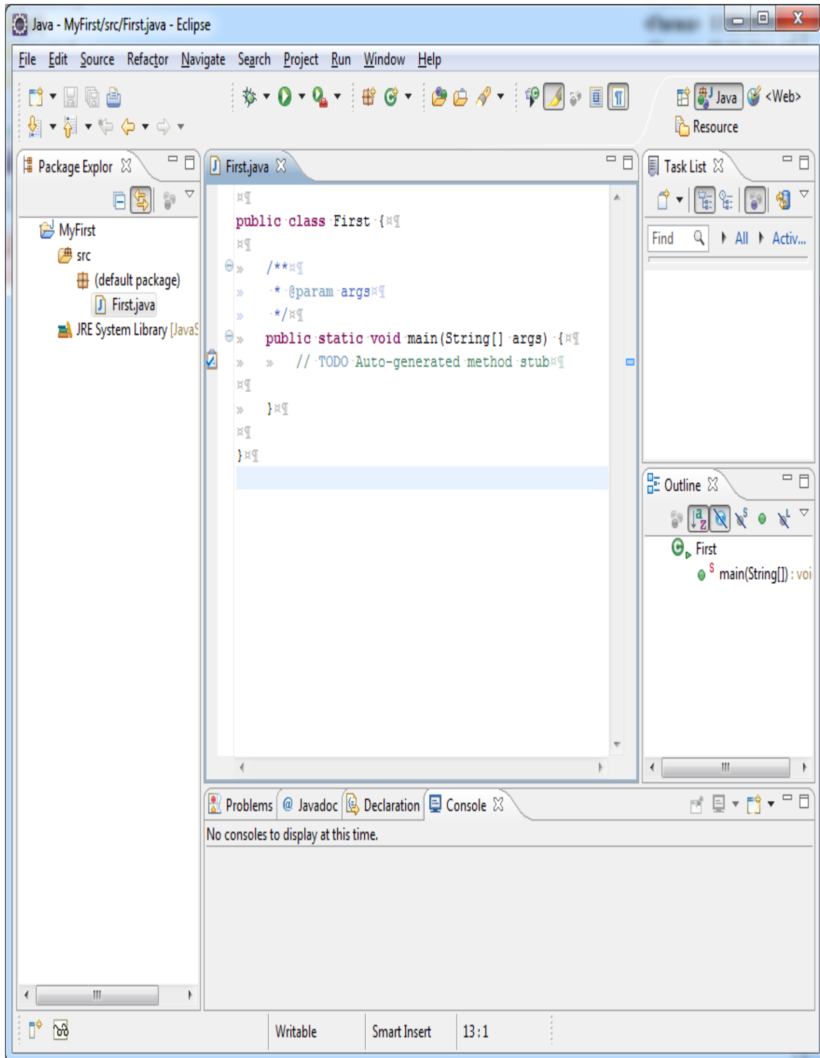
- В окне проекта иерархия First. Содержит автоматически подключенные библиотеки.
- В рабочем поле проекта вкладка Main.java с шаблоном кода проекта.
- В подключенном окне навигации Main средства просмотра членов класса Main.



Программа – это класс с именем Main. Внутри него определен главный метод с именем main(). Исполнение программы начинается с его запуска.

В коде программы по умолчанию вставлены строки:

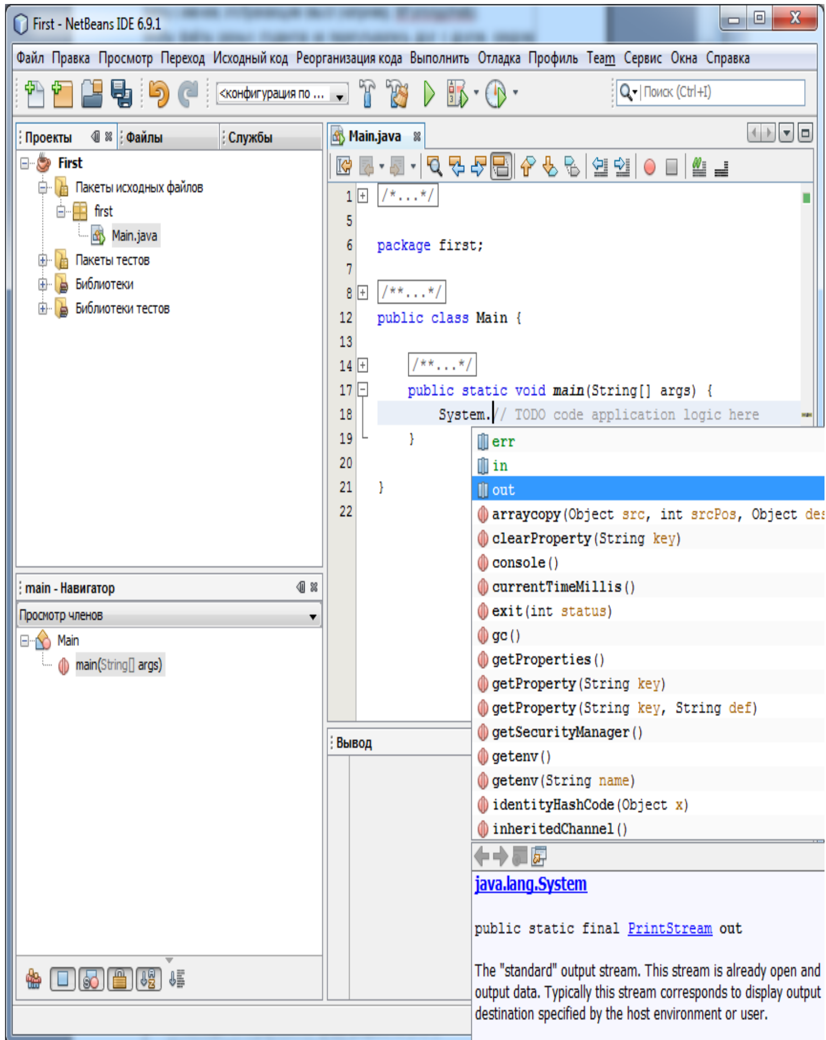
- Java комментария (в парных скобках /** и */). По нему можно автоматически построить справку (Help) к проекту.
- Текстового комментария в конце строк (с символами //).
- Пустые строки для визуального разделения фрагментов кода.



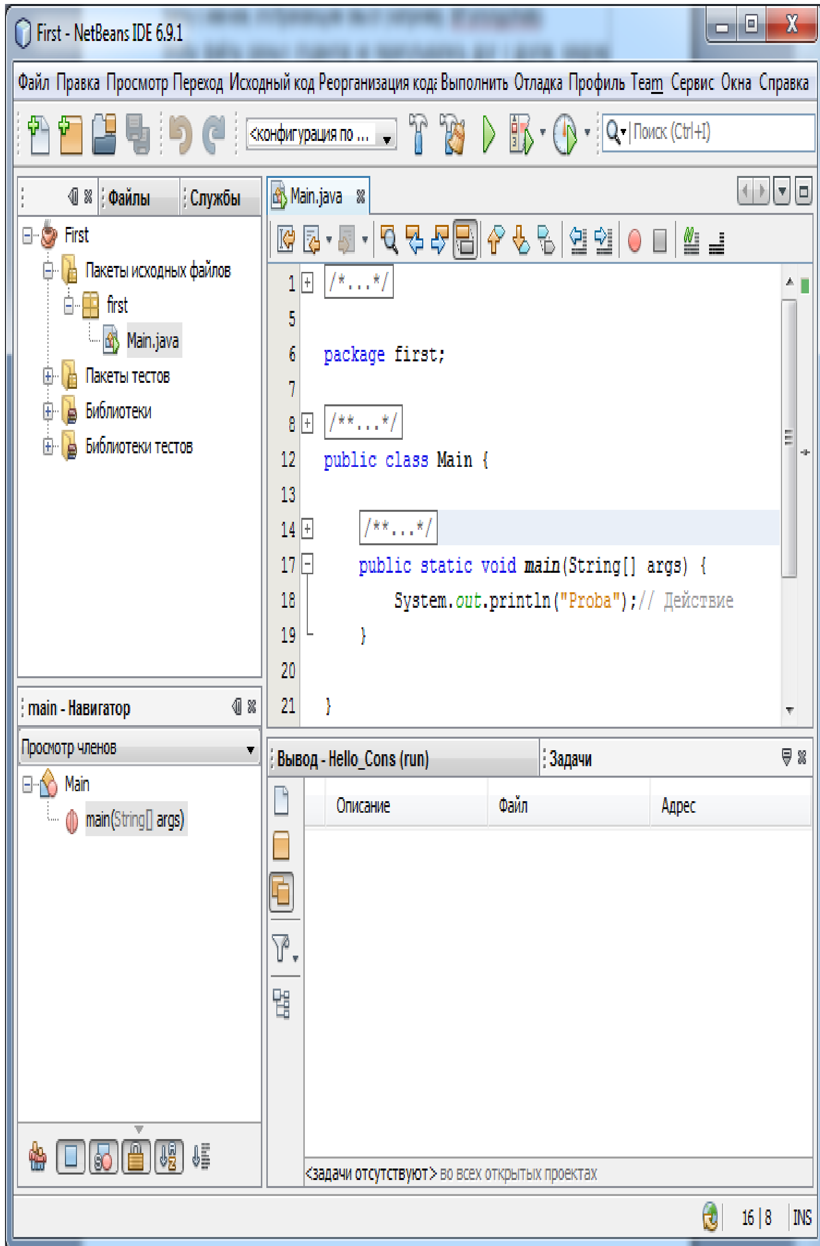
В метод main нужно добавлять свой функционал. Добавим команду вывода в консоль фразы Proba командой

System.out.println (“Proba”);

При наборе команды используем интерактивный подсказчик. После набора слова System и точки появится выпадающее меню.



В нем ищем нужное действие. Если его не видно, то набираем следующую букву и ищем снова. Найденный метод выбираем. В качестве аргумента выбранного метода вводим строку “Proba”. Если при наборе кода допускается ошибка, то ошибочный фрагмент кода подчеркивается волнистой линией. При подведении курсора к месту ошибки во всплывающем окне ошибка описывается.


First - NetBeans IDE 6.9.1
Файл Правка Просмотр Переход Исходный код Реорганизация кода Выполнить Отладка Профиль Team Сервис Окна Справка
<конфигурация по ...
Поиск (Ctrl+I)
Файлы
Службы
First
Пакеты исходных файлов
first
Main.java
Пакеты тестов
Библиотеки
Библиотеки тестов
Main.java
/*...*/
package first;
/**...*/
public class Main {
/**...*/
public static void main(String[] args) {
System.out.println("Proba");// Действие
}
}
main - Навигатор
Просмотр членов
Main
main(String[] args)
Вывод - Hello_Cons (run)
Задачи
Описание
Файл
Адрес
<задачи отсутствуют> во всех открытых проектах
16 | 8
INS

Листинг программы

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package first;
/**
 *
 * @author Костян
 */
public Class Main {
    /**
    * @param args the command line arguments
    */
    public static void main(String[] args) {
        System.out.println("Proba");// Действие
    }
}
```

Внимание. По умолчанию открывающая фигурная скобка тела класса или метода размещается в строке заголовка после сигнатуры. Это ухудшает визуальное восприятие кода, так как скобки пары не размещаются в одном и том же столбце.

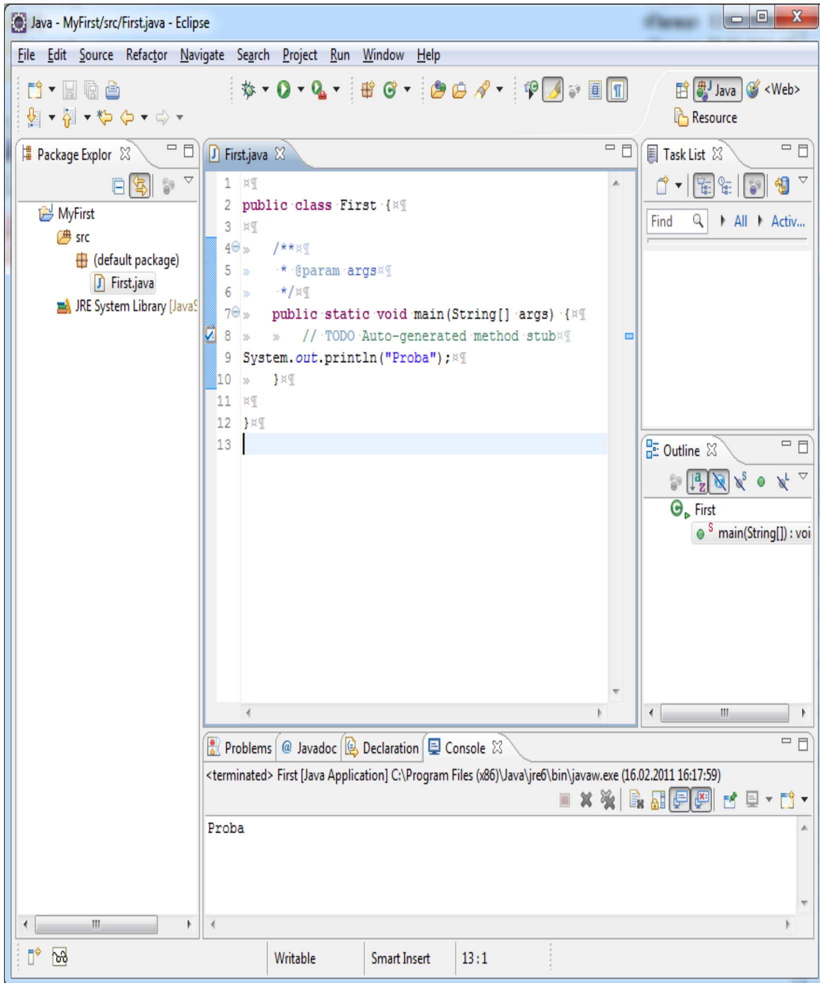
Листинг программы (другая иерархия)

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package first;
/**
 *
 * @author
 */
public Class Main
{
    /**
    * @param args the command line arguments
    */
    public static void main(String[] args)
    {
```

```
        System.out.println("Proba");// Действие
    }
}
```

Для прогона программы используем команду Выполнить=>Запустить главный проект. В закладке Вывод отображается результат консольного приложения:

- Фраза Proba.
- Сообщение о завершении и время исполнения.



Внимание!

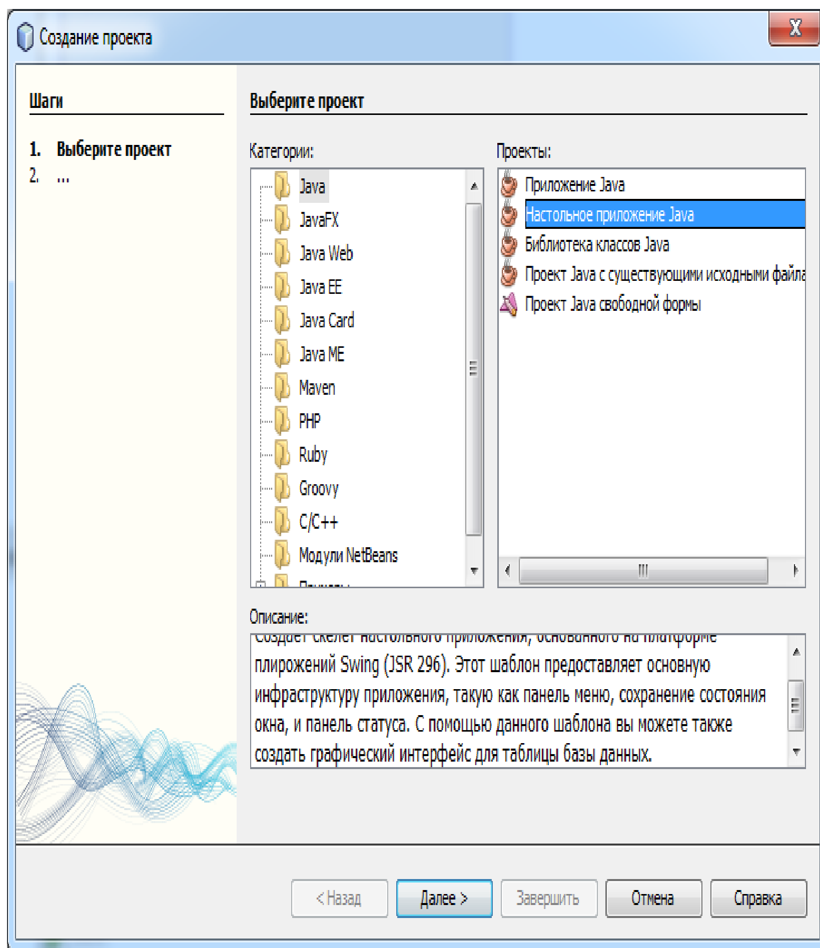
- При запуске готовой программы файлы проекта автоматически сохраняются. При любом новом прогоне обновленные файлы будут переписываться в месте размещения.
- Если Вы переходите к новому проекту, то **закройте** старый, но **не удаляйте** его. При удалении весь проект стирается из места расположения.

При больших программах желательно видеть номера строк кода. Если номера не отображаются, то для активизации нумерации строк нужно кликнуть правой кнопкой мыши по линейке слева от кода. В выпадающем меню выбрать команду - Показать номера строк. Это приведет к отображению в редакторе номеров строк.

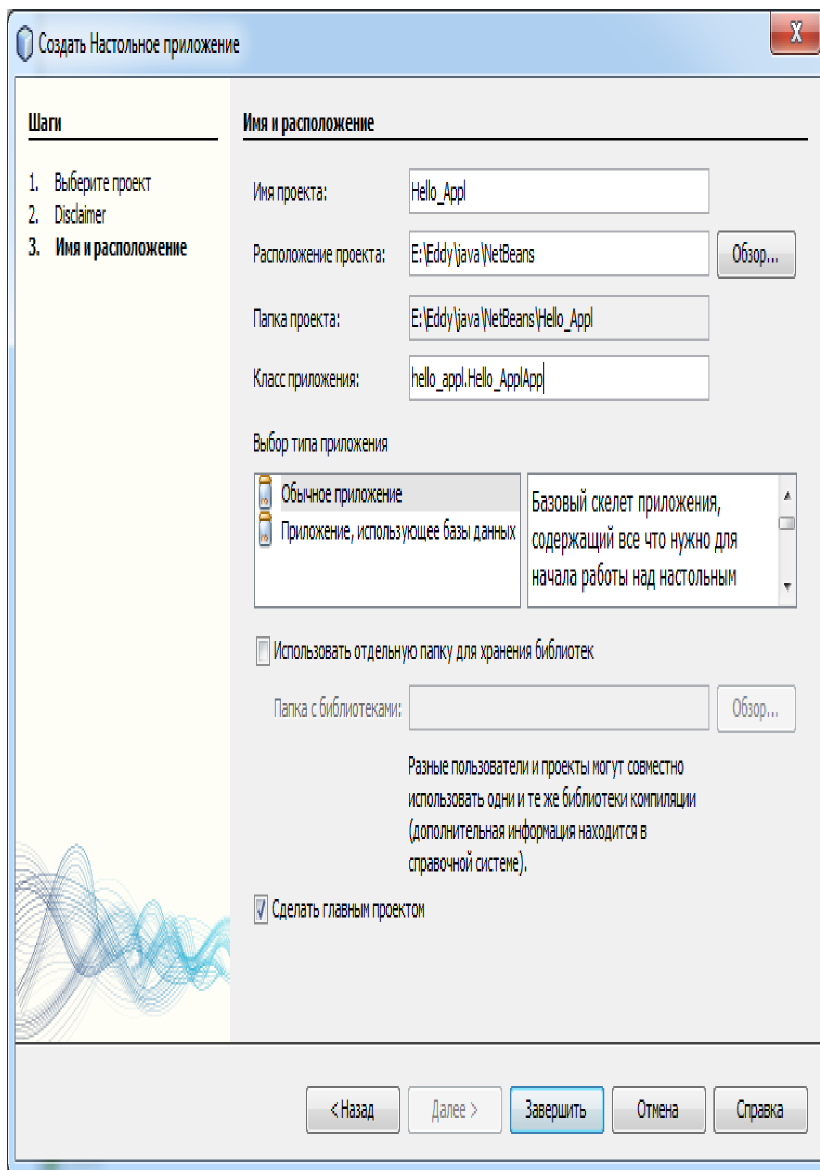
3.3. Использование графического режима

ГИП используется для создания Windows приложений. Рассмотрим пример приложения, которое должно в графическом режиме выводить в окно формы фразу «Hello, World и Россия от Акчурина». Используем ИСР NetBeans.

При создании нового проекта из меню командой Файл=>Создать_проект вызывается окно выбора типа проекта. В нем выбираем категорию Java и тип проекта - Настольное приложение Java. В нем поддерживается графический интерфейс пользователя (ГИП) с набором компонент Swing или AWT по выбору.

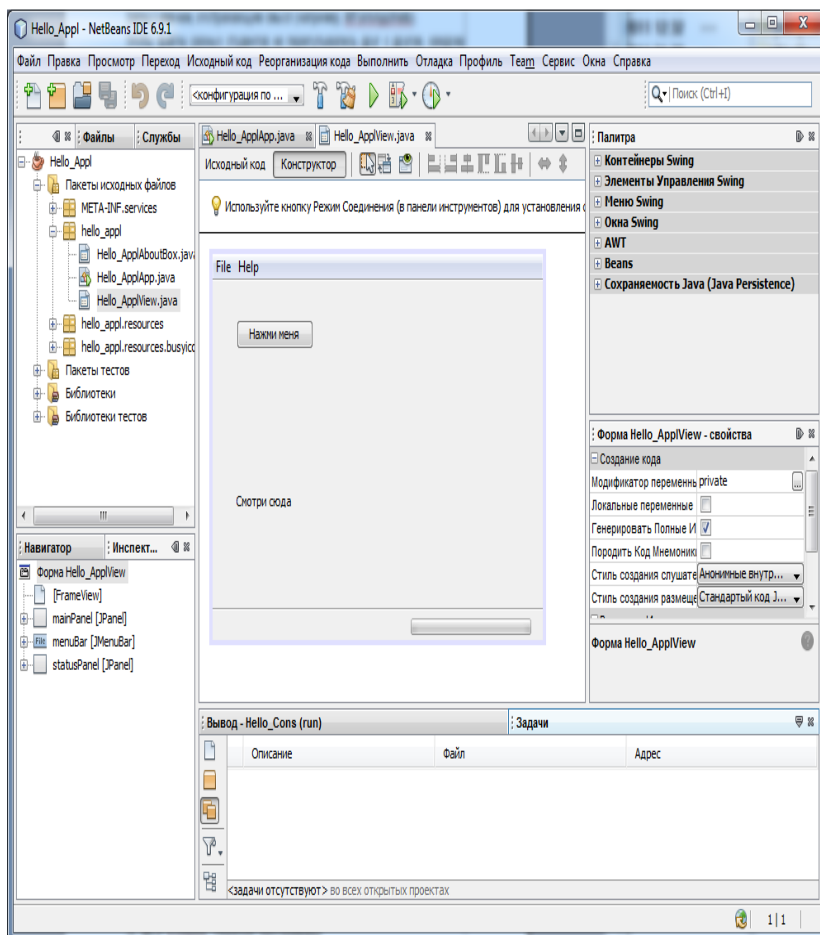


В окне выбираем Настольное приложение Java.



Проекту нужно задать имя (например, Hello_Appl) и расположение (например, E:\Eddy\java\NetBeans\Hello_Appl).

Генерируется окно проекта.



В графическом приложении ИСР использует два окна:

- Исходный код.
- Конструктор. В нем отображается форма приложения. Форма конструируется визуально, а исходный код будет формироваться автоматически.

Справа от Конструктора расположено два окна:

- Палитра доступных компонент.

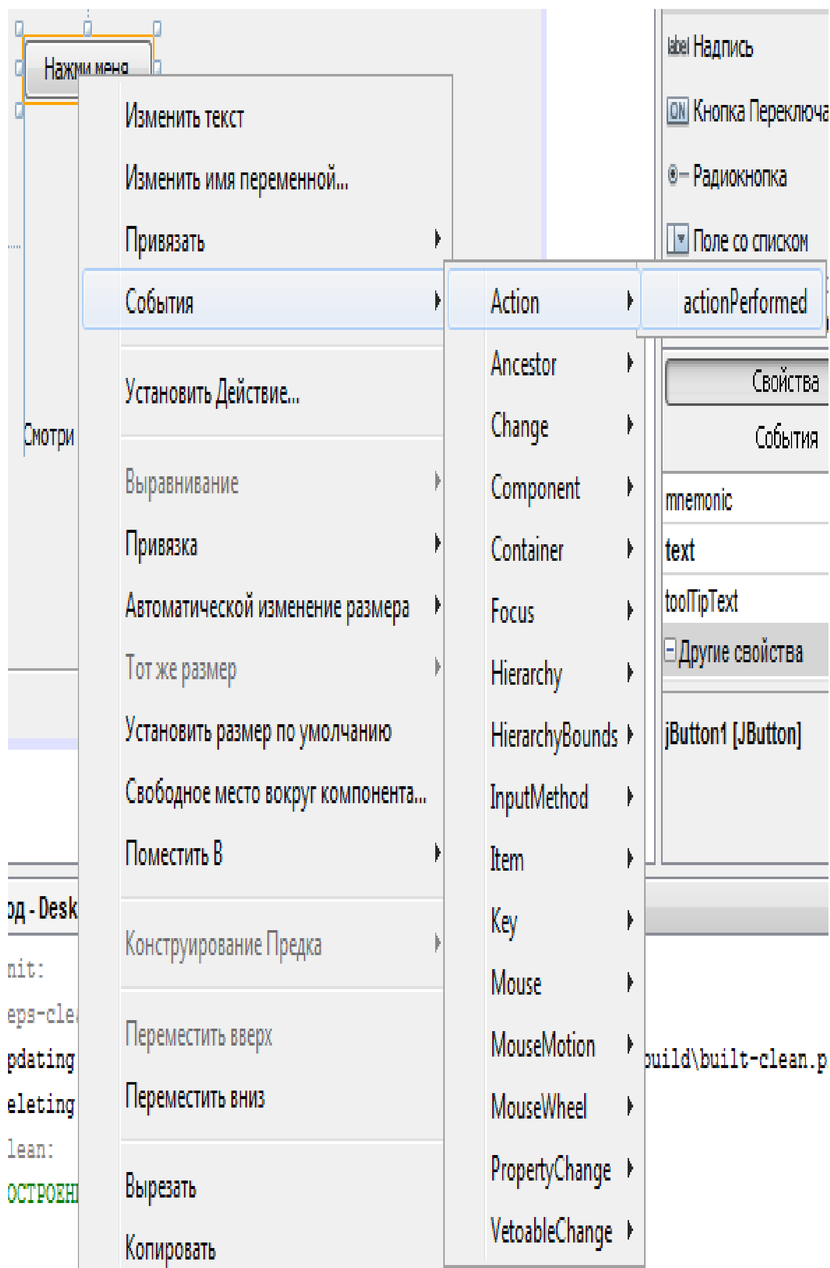
- Свойства элементов формы. В ней отображаются свойства элемента, выделенного в форме.

При визуальном конструировании формы в палитре нужно выделить компоненту и переместить ее в форму в желаемое место. Используем два компонента:

- Кнопка Button, при нажатии которой будет происходить вывод фразы. Ее имя jButton1.
- Метка Label, в которую будет выводиться фраза. Имя метки jLabel1.

В окне Свойства для кнопки Button задаем надпись «Нажми меня», а для метки Label стартовую надпись «Смотри сюда».

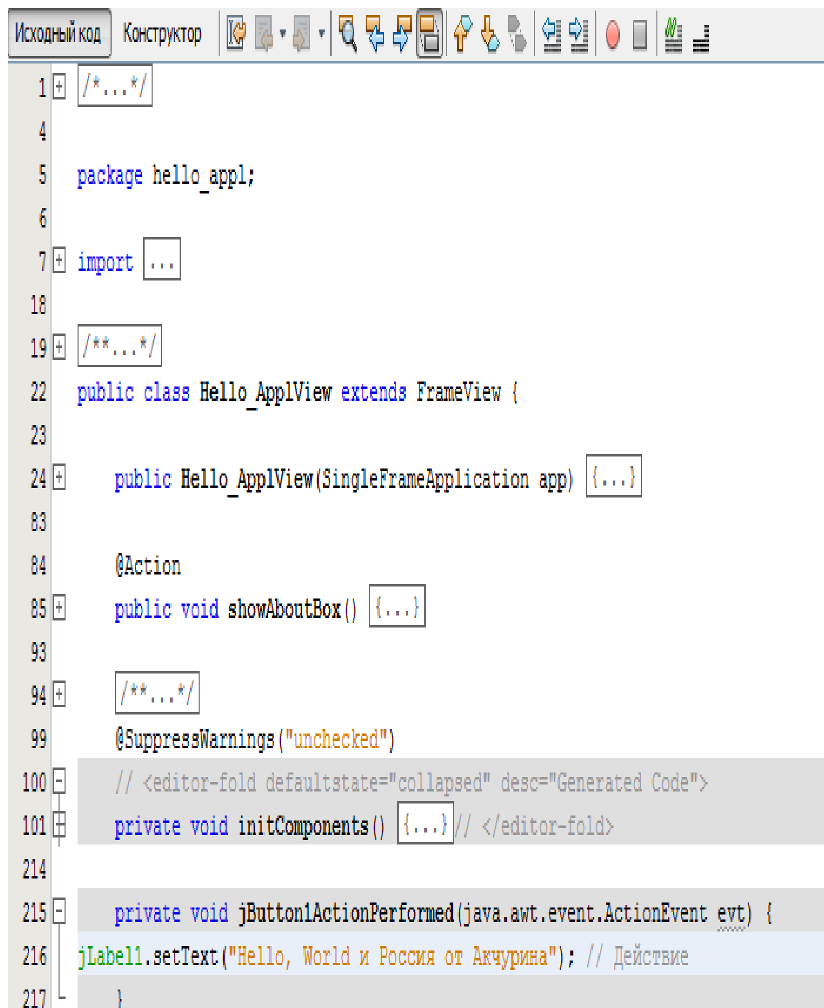
Для кнопки задаем отслеживаемое событие. Для этого в выпадающем меню кнопки выбираем это событие командой События=>Action=>actionPerformed. Это означает реакцию на приоритетное действие, для кнопки это однократный щелчок мышью.


Нажми меня
Изменить текст
Изменить имя переменной...
Привязать
События
Action
actionPerformed
Установить Действие...
Выравнивание
Привязка
Автоматической изменение размера
Тот же размер
Установить размер по умолчанию
Свободное место вокруг компонента...
Поместить В
Конструирование Предка
Переместить вверх
Переместить вниз
Вырезать
Копировать
Ancestor
Change
Component
Container
Focus
Hierarchy
HierarchyBounds
InputMethod
Item
Key
Mouse
MouseMotion
MouseWheel
PropertyChange
VetoableChange
Надпись
Кнопка Переключа
Радиокнопка
Поле со списком
Свойства
События
mnemonic
text
toolTipText
Другие свойства
jButton1 [JButton]
Смотри

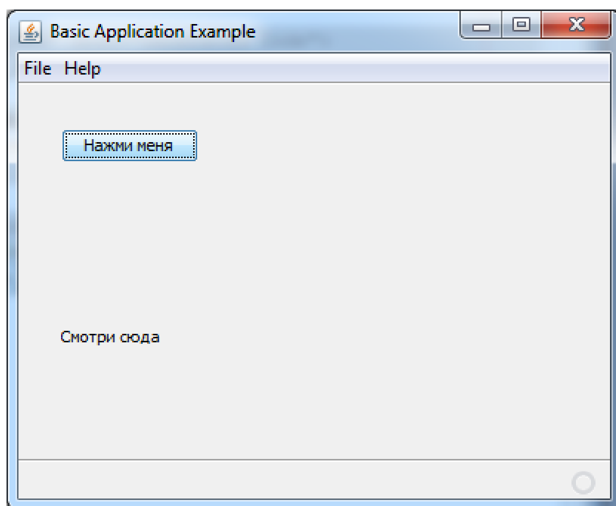
В код программы автоматически заносится обработчик события, в код которого нужно задать действие: jLabel1.setText(“Hello World и Россия от Акчурина”).

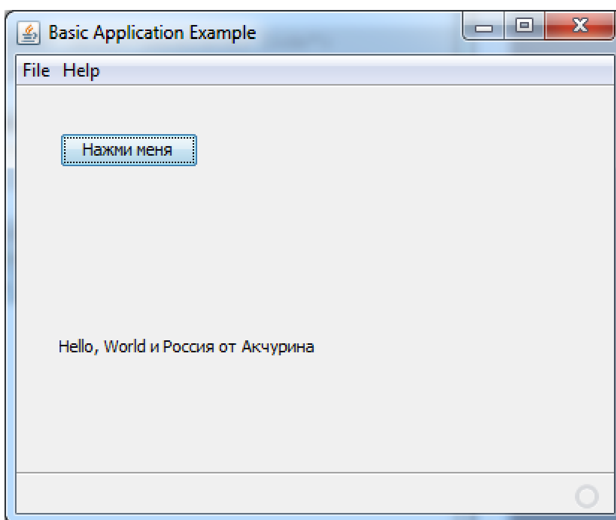
Код очень длинный, он включает действия эмулируемее системно. Его не надо редактировать.

Фрагмент листинга программы



Результаты прогона





4. Основы языка Java

4.1. Первая программа

Чтобы понять принципы работы программы Java, давайте рассмотрим традиционную программу "Hello World!" и разберем каждую строку ее кода на Java.

Для упорядочения и оформления кода в языке Java используются классы. В действительности весь выполняемый код Java должен содержаться в классе, что справедливо и для короткой программы типа "Hello". Ниже приведен код программы, отображающей в окне консоли сообщение "Hello World!".

```
Class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Вот и все, всего 5 строчек! Но даже на этом простом примере можно заметить целый ряд существенных особенностей языка Java.

- Всякая программа представляет собой один или несколько классов, в этом простейшем примере только один класс (**Class**).
- Начало класса отмечается служебным словом **Class**, за которым следует имя класса, выбираемое произвольно, в данном случае Helloworld. Все, что содержится в классе, записывается в фигурных скобках и составляет **тело класса** (Class body).
- Все действия производятся с помощью методов обработки информации, коротко говорят просто метод (method). Это название употребляется в языке Java вместо названия "функция", применяемого в других языках.
- Методы различаются по именам. Один из методов обязательно должен называться **main**, с него начинается выполнение программы. В нашей простейшей программе только один метод, а значит, имя ему main.
- Как и положено функции, метод всегда выдает в результате (чаще говорят, возвращает (returns)) только одно значение, тип которого обязательно указывается перед именем метода. Метод может и не возвращать никакого значения, играя роль процедуры, как в нашем случае. Тогда вместо типа возвращаемого значения записывается слово **void**, как это и сделано в примере.
- После имени метода в скобках, через запятую, перечисляются аргументы (arguments) метода. Для каждого аргумента указывается его тип и, через пробел, имя. В примере только один аргумент, его тип — массив, состоящий из строк символов. Строка символов — это встроенный в Java API тип

string, а квадратные скобки — признак массива. Имя массива может быть произвольным, в примере выбрано имя args.

- Перед типом возвращаемого методом значения могут быть записаны **модификаторы** (modifiers). В примере их два: слово public означает, что этот метод доступен отовсюду; слово static обеспечивает возможность вызова метода main () в самом начале выполнения программы. Модификаторы вообще необязательны, но для метода main () они необходимы

Добавим комментарии к нашему примеру.

Листинг. Первая программа с комментариями

```
/**
* Разъяснение содержания и особенностей программы...
* @author Имя Фамилия (автора)
* @version 1.0 (это версия программы)
*/
Class HelloWorld{        // HelloWorld — это только имя
    // Следующий метод начинает выполнение программы
    public static void main(String[] args)
    {
        /* Следующий метод просто выводит свой аргумент
        * на экран дисплея */
        System.out.println("Hello, 21st Century World!");
        // Следующий вызов закомментирован,
        // метод не будет выполняться
        // System.out.println("Farewell, 20th Century!");
    }
}
```

Звездочки в начале строк не имеют никакого значения, они написаны просто для выделения комментария. Пример, конечно, перегружен пояснениями (это плохой стиль), здесь просто показаны разные формы комментариев

4.2. Пространства имен

Пространства имен представляют собой способ организации различных типов, присутствующих в программах Java. Их можно сравнить с папкой в компьютерной файловой системе. Подобно папкам, пространства имен определяют для классов уникальные полные имена. Программа Java содержит одно или несколько пространств имен, каждое из которых либо определено программистом, либо определено как часть написанной ранее библиотеки классов.

Например, пространство имен System содержит класс out, который включает методы для чтения и записи в окне консоли.

При написании класса вне объявления пространства имен компилятор предоставит ему заданное по умолчанию пространство имен.

Для использования метода println, определенного в классе out, который содержится в пространстве имен System, без предварительного определения пространства имен следует использовать строку кода

```
System.out.println("Hello World и Россия от Акчурина");
```

Необходимо помнить, что всем методам, содержащимся в out, должно предшествовать System. Это быстро становится утомительным, поэтому в начало исходного файла Java целесообразно вставить директиву import, задающую пространство имен

```
import System;
```

Директива устанавливает, что предполагается пространство имен System и впоследствии можно написать короче

```
out.println("Hello World и Россия от Акчурина");
```

4.3. Алфавит

В языке Java используется кодировка unicod. Это означает:

- **Чувствительность к регистру**, M и m - это разные переменные.
- Допустимо использовать для идентификаторов символы **кириллицы**.

Алфавит Java включает:

- строчные и прописные буквы латинского алфавита;
- строчные и прописные буквы национального алфавита;
- цифры от 0 до 9;
- символ «_» ;
- набор специальных символов: " { }, 1 [ ] + — %/ \; ': ? < > = ! & # ~ *-
- прочие символы.

Алфавит служит для построения слов, которые называют лексемами. Различают типы лексем:

- идентификаторы;
- ключевые слова;
- знаки (символы) операций;
- литералы;
- разделители;
- пробельные символы.

Почти все типы лексем (кроме ключевых слов и идентификаторов) имеют собственные правила словообразования, включая собственные подмножества алфавита.

Лексемы обособляются разделителями или пробельными символами.

4.4. Комментарии

В текст программы можно вставить комментарии, которые компилятор не будет учитывать. Они очень полезны для пояснений по ходу программы. В период отладки можно выключать из действий одуе или несколько инструкций, пометив их символами комментария, как говорят программисты, "закомментарив" их. Комментарии вводятся таким образом:

- за двумя наклонными чертами подряд //, без пробела между ними, начинается комментарий, продолжающийся до конца строки;
- за наклонной чертой и звездочкой /* начинается комментарий, который может занимать несколько строк, до звездочки и наклонной черты */ (без пробелов между этими знаками).

Комментарии очень удобны для чтения и понимания кода, они превращают программу в документ, описывающий ее действия. Программу с хорошими комментариями называют самодокументированной. Поэтому в Java введены комментарии третьего типа, а в состав JDK — программа javadoc, извлекающая эти комментарии в отдельные файлы формата HTML и создающая гиперссылки между ними:

- за наклонной чертой и двумя звездочками подряд, без пробелов, /** начинается комментарий, который может занимать несколько строк до звездочки (одной) и наклонной черты */ и обрабатываться программой javadoc. В такой комментарий можно вставить указания программе javadoc, которые начинаются с символа @.

Именно так создается документация к JDK.

4.5. Разделители

В Java используются следующие разделители фрагментов кода:

- Круглые скобки ( ). В методах содержат списки аргументов.
- Фигурные скобки { }. Используются для ограничения блоков кода.
- Квадратные скобки [ ]. Используются для объявления массивов и указания элементов массивов.

4.6. Пробельные символы

Они применяются для визуального оформления текста. Компилятор их игнорирует. Java - язык свободной формы. Текст в нем не нужно особо структурировать. Элементы кода отделяются друг от друга пробельными символам: пробел, символ табуляции, символ новой строки.

Там, где допустим пробельный символ, их количество и типы могут быть любыми. Примеры правильного применеия:

- Main(); Две лексемы – имя методв и пара круглых скобок не требуют пробельного символа.
- Main(); Две лексемы – имя методв и пара круглых скобок можно записать с пробельным символом.

Пробельные символы применяются для улучшения визуального восприятия кода. Ниже 3 листинга для одного и того же кода. Компилятор реагирует одинаково.

Листинг 1. Скобка { в конце строки заголовка,

```
Class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Такой стиль используется чаще всего. Он сокращает число строк кода, но иерархия инструкций не ксегда визульно просматривается. Целесообразно использовать для длинных программ.

Листинг 2. Скобки { в отдельной строке

```
Class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello World!");
    }
}
```

Такой стиль используется реже. Он увеличивает число строк кода, но иерархия инструкций всегда визульно просматривается. Целесообразно использовать для коротких программ.

Листинг3. Без иерархии

```
Class HelloWorld
{
public static void main(String[] args)
{
System.out.println("Hello World!");
}
}
```

Такой стиль не надо использовать. Иерархия инструкций визульно не просматривается.

4.7. Идентификаторы

Это имена объектов, на них накладываются ограничения.

- Первый символ – не цифра (чтобы не воспринимался, как числовая константа).
- Следующие символы – буквы, цифры, символ подчеркивания и знак $.

В ИСР предусмотрены встроенные средства контроля правильности идентификаторов. ИСР не примет неправильный идентификатор, вы получите сообщение о допущенной ошибке.

В языке Java для многословных имен не принято применять символы подчеркивания. Рекомендуется разделять слова, используя в начале слова заглавные буквы.

Принято:

- Имена классов начинать с заглавной буквы.
- Имена методов с прописной буквы.
- В многословных именах начинать отдельные слова с прописной буквы.

Внимание. Следите за высотой символов в именах классов и методов. Язык Java чувствителен к высоте букв.

4.8. Ключевые слова

Это предварительно определенные зарезервированные идентификаторы, имеющие специальные значения для компилятора. Ключевые слова используются для инструкций (команд) Java.

Их нельзя использовать в программе в качестве идентификаторов, если только они не содержат префикс @.

Например, @if является допустимым идентификатором, но if таковым не является, поскольку if — это ключевое слово. Идентификаторы с символом @ применять не рекомендуется.

Перечень ключевых слов:

abstract	else	int	static
Boolean	extends	interface	super
break	false	long	switch
byte	final	native	synchronized
byvalue	finally	new	this
case	float	null	throw
cast	for	operator	throws
catch	future	outer	transient
char	generic	package	true
Class	goto	private	try
const	if	protected	var
continue	implements	public	void
default	import	rest	volatile
do	inner	return	while
double	instanceof	short	

Ключевые слова byvalue, cast, const, future, generic, goto, inner, operator, outer, rest, var зарезервиваны, но сейчас не используются.

4.9. Константы

В языке Java можно записывать константы разных типов в разных видах.

4.9.1. Целые

Целые константы можно записывать в трех системах счисления:

- В десятичной форме: +5, -7, 12345678.
- В восьмеричной форме, начиная с нуля: 027, -0326, 0777. В записи таких констант недопустимы цифры 8 и 9. Число, начинающееся с нуля, записано в восьмеричной форме, а не в десятичной. Не путайте с боквой О.
- В шестнадцатеричной форме, начиная с нуля и латинской буквы x или X: 0xff0a, 0xFC2D, 0x45a8, 0X77FF. Здесь строчные и прописные буквы не различаются.

Целые константы хранятся в формате типа int. В конце целой константы можно записать букву прописную L или строчную l, тогда константа будет сохраняться в длинном формате типа long: +25L, -0371, OxffL, OXDFDF1.

Совет. Не используйте при записи длинных целых констант строчную латинскую букву l, ее легко спутать с единицей.

4.9.2. Вещественные

Вещественные константы записываются только в десятичной системе счисления в двух формах:

- С фиксированной точкой: 37.25, -128.678967, +27.035.
- С плавающей точкой: 2.5e34, -0.345e-25, 37.2E+4. Можно писать строчную или прописную латинскую букву E, пробелы и скобки недопустимы.

В конце вещественной константы можно поставить букву F или f, тогда константа будет сохраняться в формате типа float: 3.5f, -45.67F, 4.7e-5f.

Можно приписать и букву D (или d): 0.045D, -456.77889d, означающую тип double, но это излишне, поскольку действительные константы и так хранятся в формате типа double.

4.9.3. Символы

Для записи одиночных символов используются следующие формы.

- Печатные символы можно записать в апострофах: ' a ', ' N ', ' ? '.
- Управляющие символы записываются в апострофах с обратной наклонной чертой (эскейп-последовательности):
  - ' \n ' — символ перевода строки (newline) с кодом ASCII 10;
  - ' \r ' — символ возврата каретки CR с кодом 13;
  - ' \f ' — символ перевода страницы FF с кодом 12;
  - ' \b ' — символ возврата на шаг BS с кодом 8;
  - ' \t ' — символ горизонтальной табуляции HT с кодом 9;
  - ' \\ ' — обратная наклонная черта;
  - ' \" ' — кавычка;
  - ' \' ' — апостроф.
- Код любого символа в кодировке Unicode набирается в апострофах после обратной наклонной черты и латинской буквы u ровно четырьмя шестнадцатеричными цифрами: ' \u0053 ' — буква S, ' \u0416 ' — буква Ж.

Символы хранятся в формате типа char

Примечание. Прописные русские буквы в кодировке Unicode занимают диапазон от ' \u0410 ' — заглавная буква А, до ' \u042F ' — заглавная Я, строчные буквы от ' \u0430 ' — а, до ' \044F ' — я.

В какой бы форме ни записывались символы, компилятор переводит их в Unicode, включая и исходный текст программы.

Замечание. Компилятор и исполняющая система Java работают только с кодировкой Unicode.

4.9.4. Строки

Строки символов заключаются в двойные кавычки. Управляющие символы и коды записываются в строках точно так же, с обратной наклонной чертой, но без апострофов, и оказывают то же действие. Строки могут располагаться только на одной строке исходного кода, нельзя открывающую кавычку поставить на одной строке, а закрывающую — на следующей. Вот примеры:

```
"Это строка\nс переносом"
"\"Спартак\" — Чемпион!"
```

Замечание. Строки символов нельзя начинать на одной строке исходного кода, а заканчивать на другой.

Для строковых констант определена операция сцеплений, обозначаемая плюсом.

" Сцепление " + "строк" дает в результате строку "Сцепление строк".

Чтобы записать длинную строку в виде одной строковой константы, надо после закрывающей кавычки на первой и следующих строках поставить плюс +; тогда компилятор соберет две (или более) строки в одну строковую константу, например:

```
"Одна строковая константа, записанная "+
"на двух строках исходного текста"
```

Совет. Используйте Unicode напрямую только в крайних случаях.

4.10. Переменные

Переменная представляет числовое или строковое значение или объект класса. Значение, хранящееся в переменной, может измениться, однако имя остается прежним.

Переменная представляет собой один тип поля. Переменная может быть объявлена в любом месте кода. При объявлении нужно указать тип переменной, задавать ее значение не обязательно. Компилятор сделает инициализацию по умолчанию.

Следующий код является простым примером объявления целочисленной переменной, присвоения ей значения и последующего присвоения нового значения.

```
int x = 1;  // x получает значение 1
```

```
x = 2;     // x получает значение 2
```

В Java переменные объявляются с определенным типом данных и именем. Тип указывает точный объем памяти, который следует выделить для хранения значения при выполнении приложения. При преобразовании переменной из одного типа в другой язык Java следует определенным правилам.

Замечание. Не указывайте в именах знак доллара. Компилятор Java использует его для записи имен вложенных классов.

Вот примеры правильных идентификаторов:

```
a1    my_var   var3_5   _var   veryLongVarName
aName   theName   a2Vh36kBnMt456dX
```

В именах лучше не использовать строчную букву l (маленькая L), которую легко спутать с единицей, и букву O, которую легко принять за нуль.

Служебные слова Java, такие как Class, void, static, зарезервированы, их нельзя использовать в качестве идентификаторов своих объектов.

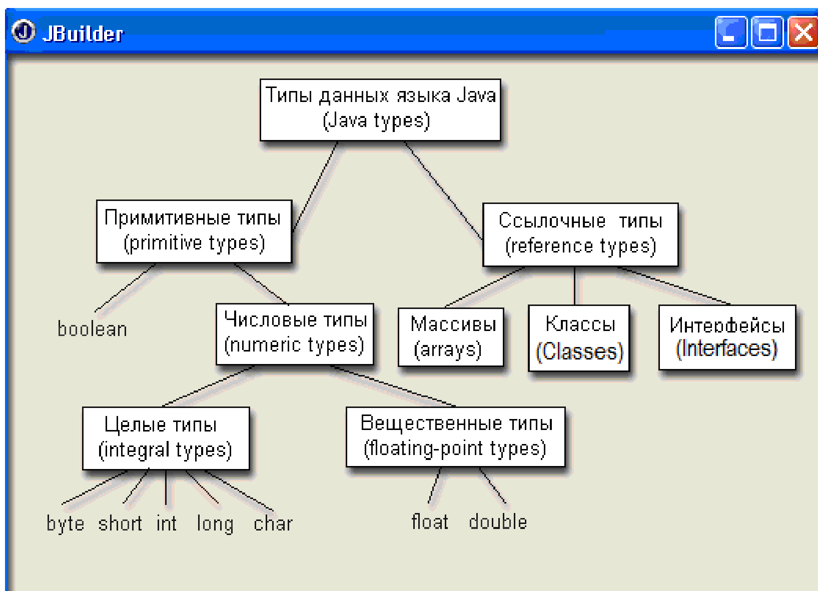
Составное имя (qualified name) - это несколько идентификаторов, разделенных точками, без пробелов, например, уже встречавшееся нам имя System.out.println().

5. Типы

5.1. Иерархия типов данных

Java — язык со строгой типизацией (strongly typed language).

Иерархия типов данных в Java



Все типы исходных данных, встроенные в язык Java, делятся на две группы:

- примитивные типы (primitive types),
- ссылочные типы (reference types).

В переменных примитивных типов хранятся данные, а в переменных ссылочного типа ссылки на фактические данные. Ссылочные типы также называются объектами..

Ссылочные типы делятся на массивы (arrays), классы (Classes) и интерфейсы (interfaces).

Примитивных типов всего 8. Их можно разделить на:

- логический тип boolean.
- 5 типов целых чисел - byte, short, int, long, char.
- 2 типа вещественных чисел - float и double.

По имени переменной невозможно определить ее тип, все переменные обязательно должны быть описаны перед их использованием.

Синтаксис объявления переменной:

Тип Имя1 = Значение, Имя2 = Значение;

Записывается имя типа, затем через пробел список имен переменных, разделенных запятой. Для всех или некоторых переменных можно указать начальные значения после знака равенства, которыми могут служить любые константные выражения того же типа. Описание каждого типа завершается точкой с запятой. В программе может быть сколько угодно описаний каждого типа.

Типичная программа Java использует типы из библиотеки классов, а также пользовательские типы, моделирующие принципы, относящиеся к проблемной области программы.

К сведениям, хранимым в типе, может относиться следующее:

- Место для хранения переменной типа.
- Максимальное и минимальное значения, которые могут быть представлены.
- Содержащиеся члены (методы, поля, события и т.д.).
- Базовый тип, которому он наследует.
- Расположение, в котором будет выделена память для переменных во время выполнения.
- Разрешенные виды операций.

Компилятор использует сведения о типе, чтобы убедиться, что все операции, выполняемые в коде, являются строго типизированными. Например, при объявлении переменной численного типа int, компилятор позволяет использовать переменную и операции вычитания. При попытке выполнить эти же операции с переменной логического типа bool компилятор вызовет ошибку.

Существует возможность преобразовать тип значения в ссылочный тип и обратно в тип значения с помощью упаковки-преобразования и распаковки-преобразования. За исключением упакованного типа значения преобразовать ссылочный тип в тип значения невозможно.

5.2. Преобразования типов

Все вычисления происходят с использованием типа double. Другие типы чисел могут применяться для уменьшения занимаемой памяти. При их использовании перед вычислением они преобразуются в тип double. Различают преобразования:

- Неявные преобразования используются для совместимых типов. Значения источника полностью отображаются приемником. Например, при преобразовании int в double (int – подмножество double). Такое преобразование выполняется автоматически, его не надо заказывать.
- Явные преобразования используются для несовместимых типов. Например, при преобразовании double в int (int – подмножество double). Значения источника не полностью отображаются приемником. Если типы несовместимы, но ошибка допустима, то преобразование возможно, но его нужно явно заказать: для этого перед преобразуемым выражением добавляется префикс идентификации конечного типа (в круглых скобках).

Например,

```
double db=12.94;
int i = (int) db;
```

5.3. Логический тип

5.3.1. Задание

Значения логического типа Boolean возникают в результате различных сравнений, вроде 2 > з, и используются, главным образом, в условных операторах и операторах циклов. Логических значений всего два: true (истина) и false (ложь). Это служебные слова Java. Описание переменных этого типа выглядит так:

```
Boolean b = true, bb = false, bool2;
```

Над логическими данными можно выполнять операции присваивания, например, bool2 = true, в том числе и составные с логическими операциями; сравнение на равенство b == bb и на неравенство b != bb, а также логические операции.

5.4. Логические операции

Определены логические операции:

Операция	Описание
! B	Отрицание (NOT)
B1 & B2	Конъюнкция, операция И (AND)
B1 && B2	Условная конъюнкция (Cond-AND)
B1 | B2	Дизъюнкция, операция ИЛИ (OR)
B1 || B2	Условная дизъюнкция (OR)
B1 ^ B2	Исключающее ИЛИ (XOR)

Они выполняются над логическими данными, их резу льтатом будет тоже логическое значение true или false. Словами эти правила можно выразить так:

- отрицание меняет значение истинности;
- конъюнкция истинна, только если оба операнда истинны;
- дизъюнкция ложна, только если оба операнда ложны;
- исключающее ИЛИ истинно, только если значения операндов различны.

Кроме перечисленных четырех логических операций есть еще две логические операции сокращенного (или условного) вычисления. В них удвоенные знаки амперсанда и вертикальной черты следует записывать без пробелов.

Правый операнд условных операций вычисляется только в том случае, если от него зависит результат операции, т.е. если левый операнд конъюнкции имеет значение true, или левый операнд дизъюнкции имеет значение false.

Замечание. Практически всегда в Java используются именно сокращенные логические операции.

5.5. Целые типы

5.5.1. Задание

Спецификация языка Java определяет разрядность (количество байтов, выделяемых для хранения значений типа в оперативной памяти) и диапазон значений каждого типа. Для целых типов они приведены в таблице.

Тип	Разрядность (байт)	Диапазон
byte	1	от -128 до 127
short	2	от -32768 до 32767
int	4	от -2147483648 до 2147483647
long	8	от -9223372036854775808 до 9223372036854775807
char	2	от '\u0000' до '\uFFFF', в десятичной форме от 0 до 65535

Хотя тип char занимает два байта, в арифметических вычислениях он участвует как тип int, ему выделяется 4 байта, два старших байта заполняются нулями.

Примеры определения переменных целых типов:

```
byte b1 = 50, b2 = -99, b3;
short det = 0, ind = 1;
int i = -100, j = 100, k = 9999;
long big = 50, veryBig = 2147483648L;
char c1 = 'A', c2 = '?', newLine = '\n';
```

Целые типы хранятся в двоичном виде с использованием дополнительного кода. Последнее означает, что для отрицательных чисел хранится не их двоичное представление, а дополнительный код этого двоичного представления. Дополнительный же код получается так: в двоичном представлении все нули меняются на единицы, а единицы на нули, после чего к результату прибавляется единица в двоичной арифметике.

Над целыми типами можно производить массу операций. Их набор восходит к языку C, он оказался удобным и кочует из языка в язык почти без изменений. Особенности применения этих операций в языке Java показаны на примерах.

5.5.2. Арифметика с целыми числами

К арифметическим операциям относятся:

Операция	Описание
+ C1	Унарный плюс
++ C1	Префиксный инкремент. Увеличение C1 на 1 перед использованием
-- C1	Префиксный декремент Уменьшение C1 на 1 перед использованием
C1++	Постфиксный инкремент. Увеличение C1 на 1 после использования
C1--	Постфиксный инкремент. Уменьшение C1 на 1 после использования
C1 + C2	Сложение
- C1	Унарный минус
C1 - C2	Вычитание
C1 * C2	Умножение
C1 / C2	Деление целочисленное Остаток отбрасывается
C1 % C2	Остаток от деление Целая часть результата отбрасывается

Операции инкремент и декремент означают увеличение или уменьшение значения переменной на единицу и применяются только к переменным, но не к константам или выражениям, нельзя написать 5++ или (a + b)++.

5.5.3. Сдвиги

Операции сдвига применяются к битам двоичного представления целого числа. Используется представление в дополнительном коде с раширенем знака (бит знака повторяется во всех старших битах числа до первого значащего).

Число	Прямой код	Дополнительный	С расширенеием
50	0..0110010	0..0110010	0..0110010
-50	1..0110010	1..1001101	1..1001101

В языке Java есть три операции сдвига двоичных разрядов:

Операция	Описание
B << H	Для B сдвиг влево на H разрядов Арифметический сдвиг. Бит знака не трогается
B >> H	Для B сдвиг вправо на H разрядов. Арифметический сдвиг. Бит знака сохраняется
B >>> H	Для B беззнаковый сдвиг вправо на H разрядов Логический сдвиг. Левые позиции заполняются нулями

Эти операции своеобразны тем, что левый и правый операнды в них имеют разный смысл. Слева стоит значение целого типа, а правая часть показывает, на сколько двоичных разрядов сдвигается значение, стоящее в левой части.

Пример. Числа B1=50 и B2=-50 подвергаются разным сдвигам с H=2

```
Int B1, B2, B3, B4, B5, B6 H=2;
B1 = 50; // двоичное         00000000000000000000000000110010 (50)
B2 = -50; // двоичное        11111111111111111111111111001110 (-50)
B3 = B1<<H; // двоичное      00000000000000000000000001101000 (200)
B4 = B2<<H; // двоичное      11111111111111111111111100111000 (-200)
B5 = B1>>>H; // двоичное     00000000000000000000000000001100 (12)
B6 = B2>>>H; // двоичное     00111111111111111111111111110011 (не -12)
```

В операциях сдвига влево и арифметическом сдвиге влево результат равен умножению на 2 в степени H.

При безнаковом сдвиге вправо для положительных чисел результат равен делению на 2 в степени H, а для отрицательных чисел нет

Замечание. Будьте осторожны при использовании сдвигов вправо.

5.5.4. Побитовые операции

Иногда приходится изменять значения отдельных битов в целых данных. Это выполняется с помощью побитовых (bitwise) операций путем наложения маски. В языке Java есть 4 побитовые операции:

Символ	Операция

<table>
<tr><td>~</td><td>дополнение (complement)</td></tr>
<tr><td>&</td><td>побитовая конъюнкция (bitwise AND)</td></tr>
<tr><td>|</td><td>побитовая дизъюнкция (bitwise OR)</td></tr>
<tr><td>^</td><td>побитовое исключающее ИЛИ (bitwise XOR).</td></tr>
</table>

Они выполняются поразрядно, после того как оба операнда будут приведены к одному типу int или long, так же как и для арифметических операций, а значит, и к одной разрядности. Операции над каждой парой битов выполняются согласно таблице истинности.

<table>
<tr><td>nl</td><td>n2</td><td>~nl</td><td>nl & n2</td><td>nl | n2</td><td>nl ^ n2</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td></tr>
</table>

Двоичное представление каждого результата занимает 32 бита.

Заметьте, что дополнение ~x всегда эквивалентно (-x) - 1.

5.5.5. Операции присваивания

Простая операция присваивания (simple assignment operator) записывается знаком равенства =, слева от которого стоит переменная, а справа выражение, совместимое с типом переменной:

```
x = 3.5;
y = 2 * (x - 0.567) / (x + 2);
```

Операция присваивания действует так: выражение, стоящее после знака равенства, вычисляется и приводится к типу переменной, стоящей слева от знака равенства. Результатом операции будет приведенное значение правой части.

Кроме простой операции присваивания есть еще 11 составных операций присваивания (compound assignment operators):

<table>
<tr><th>Операция</th><th>Результат</th></tr>
<tr><td>C2 += C1</td><td>C2 = Старое C2 + C1</td></tr>
<tr><td>C2 -= C1</td><td>C2 = Старое C2 - C1</td></tr>
<tr><td>C2 *= C1</td><td>C2 = Старое C2 * C1</td></tr>
<tr><td>C2 /= C1</td><td>C2 = Старое C2 / C1</td></tr>
<tr><td>C2 %= C1</td><td>C2 = Старое C2 % C1</td></tr>
<tr><td>C2 &= C1</td><td>C2 = Старое C2 & C1</td></tr>
<tr><td>C2 |= C1</td><td>C2 = Старое C2 | C1</td></tr>
<tr><td>C2 ^= C1</td><td>C2 = Старое C2 ^ C1</td></tr>
</table>

C2 <<= C1	C2 = Старое C2 << C1
C2 >>= C1	C2 = Старое C2 >> C1
C2 >>>= C1	C2 = Старое C2 >>> C1

Символы в составной операции записываются без пробелов, нельзя переставлять их местами.

5.6. Вещественные типы

5.6.1. Задание вещественного типа

Вещественных типов в Java два: float и double. Они характеризуются разрядностью, диапазоном значений и точностью представления, отвечающим стандарту IEEE 754-1985 с некоторыми изменениями. К обычным вещественным числам добавляются еще три значения:

- Положительная бесконечность, выражаемая константой POSITIVE_INFINITY и возникающая при переполнении положительного значения, например, в результате операции умножения 3.0*6e307.
- Отрицательная бесконечность NEGATIVE_INFINITY.
- "Не число", записываемое константой NaN (Not a Number) и возникающее при делении вещественного числа на нуль или умножении нуля на бесконечность.

Кроме того, стандарт различает положительный и отрицательный нуль, возникающий при делении на бесконечность соответствующего знака, хотя сравнение 0.0 == -0.0 дает true.

Операции с бесконечностями выполняются по обычным математическим правилам.

Во всем остальном вещественные типы — это вещественные значения, к которым применимы все арифметические операции и сравнения, перечисленные для целых типов. Характеристики вещественных типов приведены в таблице.

Тип	Разрядность, байт	Диапазон	Точность
float	4	$3{,}4e-38 < \lvert x \rvert < 3{,}4e38$	7—8 цифр
double	8	$1{,}7e-308 < \lvert x \rvert < 1{,}7e308$	17 цифр

Примеры определения вещественных типов:

```
float x = 0.001, y = -34.789;
double 21 = -16.2305, z2;
```

Поскольку к вещественным типам применимы все арифметические операции и сравнения, целые и вещественные значения можно смешивать в операциях. При этом правило приведения типов дополняется такими условиями:

- если в операции один операнд имеет тип double, то и другой приводится к типу double;
- если один операнд имеет тип float, то и другой приводится к типу float;
- в противном случае действует правило приведения целых значений.

5.6.2. Операции над вещественными числами

К операциям над вещественными числами относятся:

Операция	Описание
+ C1	Унарный плюс
C1 + C2	Сложение
- C1	Унарный минус
C1 - C2	Вычитание
C1 * C2	Умножение
C1 / C2	Деление

5.7. Операции сравнения

В языке Java 6 обычных операций сравнения целых чисел по величине:

- больше;
- меньше < ;
- больше или равно >= ; >
- меньше или равно <= ;
- равно == ;
- не равно !=.

Операция	Описание
A > B	Если A больше B, то результат = true
A < B	Если A меньше B, то результат = true
A >= B	Если A больше или равно B, то результат = true
A <= B	Если A меньше или равно B, то результат = true
A== B	Если A равно B, то результат = true
A!= B	Если A не равно B, то результат = true

Сдвоенные символы записываются без пробелов, их нельзя переставлять местами, запись => будет неверной.

6. Выражения

Из констант и переменных, операций над ними, вызовов методов и скобок составляются выражения (expressions). Разумеется, все элементы выражения должны быть совместимы, нельзя написать, например, 2 + true. При вычислении выражения выполняются четыре правила:

- Операции одного приоритета вычисляются слева направо: x + y + z вычисляется как (x + y) + z. Исключение: операции присваивания вычисляются справа налево: x = y = z вычисляется как x = (y = z).
- Левый операнд вычисляется раньше правого.
- Операнды полностью вычисляются перед выполнением операции.
- Перед выполнением составной операции присваивания значение левой части сохраняется для использования в правой части.

Следующие примеры показывает особенности применения первых трех правил. Пусть

```
int a = 3, b = 5;
```

Тогда результатом выражения ь + (Ь = 3) будет число 8; но результатом выражения (Ь = 3) + b будет число 6. Выражение b += (Ь = 3) даст в результате 8, потому что вычисляется как первое из приведенных выше выражений.

Четвертое правило можно продемонстрировать так. При тех же определениях а и ь в результате вычисления выражения ь += а += ь += 7 получим 20. Хотя операции присваивания выполняются справа налево и после первой, правой, операции значение ь становится равным 12, но в последнем, левом, присваивании участвует старое значение ь, равное 5. А в результате двух последовательных вычислений

```
a += b += 7;
b += a;
```

получим 27, поскольку во втором выражении участвует уже новое значение переменной ь, равное 12.

Выражения могут иметь сложный и запутанный вид. В таких случаях возникает вопрос о приоритете операций, о том, какие операции будут выполнены в первую очередь. Естественно, умножение и деление производится раньше сложения и вычитания. Остальные правила перечислены в следующем разделе.

Порядок вычисления выражения всегда можно отрегулировать скобками, их можно вставить сколько угодно. Но здесь важно соблюдать "золотую середину". При большом количестве скобок снижается наглядность выражения и лег-

ко ошибиться в расстановке скобок. Если выражение со скобками корректно, то компилятор может отследить только парность скобок, но не правильность их расстановки.

Приоритет операций. Операции перечислены в порядке убывания приоритета. Операции на одной строке имеют одинаковый приоритет.

1. Постфиксные операции ++ и —.
2. Префиксные операции ++ и —, дополнение ~ и отрицание !.
3. Приведение типа (тип).
4. Умножение *, деление / и взятие остатка %.
5. Сложение + и вычитание -.
6. Сдвиги <<, >>, >>>.
7. Сравнения >, <, >=, <=.
8. Сравнения ==, !=.
9. Побитовая конъюнкция &.
10. Побитовое исключающее ИЛИ ^.
11. Побитовая дизъюнкция |.
12. Конъюнкция &&.
13. Дизъюнкция | |.
14. Условная операция ?:.
15. Присваивания =, +=, -=, *=, /=, %=, &=, ^=, |=, <<, >>, >>>.

Здесь перечислены не все операции языка Java, список будет дополняться по мере изучения новых операций.

7. Решения и ветвления

7.1. Безусловный переход вызовом функций

Когда компилятор находит в основном тексте программы имя функции, то происходит приостановка выполнения текущего кода программы и осуществляется переход к найденной функции. Когда функция выполнится и завершит свою работу, то произойдет возврат в основной код программы, на ту инструкцию, которая следует за именем функции.

Имя функции должно содержать пару круглых скобок (), даже если у функции нет аргументов. Это признак функции или метода.

7.2. Инструкция if; else

Применяется для ветвления по двум ветвям. Синтаксис инструкции:

```
if (условие) {
        Блок инструкций 1;
    }
else {
        Блок инструкций 2;
    }
```

Фраза else может отсутствовать.

Если условие выполняется, то исполняется Блок инструкций 1, в противном случае исполняется Блок инструкций 2. Если в блоке только одна инструкция, то фигурные скобки можно пропустить.

Пример. В нем сравниваются значения One, Two. В зависимости от результата выводится одно из двух сообщений.

```
if (a < x) {
    x = a + b;
}
else {
    x = a — b;
}
```

Если в блоке всего одна инструкция, то обрамляющие блок фигурные скобки можно опустить. Однако рекомендуется применят ь скобки всегда, имея в виду, что блок может быть расширен.

7.3. Вложенные инструкции if; else

Применяются для множественного ветвления. Синтаксис инструкции:

```
if (условие_1) {
        Блок инструкций 1;
    }
else
    if (условие_2)  {
            Блок инструкций 2;
        }
    else {
            Блок инструкций 3;
        }
```

Фраза else может отсутствовать.

Если условие_1 выполняется, то исполняется Блок инструкций 1, в противном случае проверяется условие_2. Если оно выполняется, то исполняется Блок инструкций 2, в противном случае Блок инструкций 3.

Если в блоке всего одна инструкция, то обрамляющие блок фигурные скобки можно опустить. Однако рекомендуется применят ь скобки всегда, имея в виду, что блок может быть расширен.

7.4. Инструкции switch, case

Когда вы имеете сложный набор условий, то использование вложенных инструкций if…else приводит к громоздкому коду. Лучше воспользоваться инструкцией switch. Инструкция switch выбирает нужное действие из списка возможных, размещенных во фразах выбора case.

Синтаксис инструкции

```
switch (Условие) {
    case константа_1: инструкция действия; инструкция прерывания;
    case константа_2: инструкция действия; инструкция прерывания;
    ……..
    default:: инструкция;
}
```

Условие (помещено в круглые скобки) это выражение, которое возвращает целочисленную константу для выбора действия из пронумерованных вариантов.

Далее следует блок из секций.

- Секция выбора — **case**. Она нужна для определения действия, которое будет выполняться при совпадении значения Условия с константой в секции case. В этой секции после двоеточия (:) следуют инструкции действий

(хотя бы одна), а также инструкция прерывания действия (она обязательна, иначе будет сквозное выполнение секций выбора).

- Секция действия по умолчанию — **default**. Она может отсутствовать. Она выполняется в том случае, если со значением константы Условие не совпала ни одна константа из секции выбора.

Если результат Условия совпадет с константным значением секции case, то будет выполняться соответствующий ему блок инструкций. В качестве инструкции прерывания действия используют инстукцию break, которая прерывает выполнение инструкции switch.

Пример. Программа запрашивает номер пользователя. В зависимости от введенного номера выводится строка из списка.

```
Class MyClass {
    static void Main(string[] args) {
        int user = 1;
        System.out.print("Ваш номер = ");
        user = Convert.ToInt32(Console.ReadLine());
        switch (user)       {
            case 1: System.out.println ("Здравствуйте Userl"); break;
            case 2: System.out.println ("Здравствуйте User2"); break;
            case 3: System.out.println ("Здравствуйте User3"); break;
            default: System.out.println ("Здравствуйте новичок"); break;
        }
    }
}
```

8. Циклы

Циклом называется группа инструкций, повторяющихся многократно с разными данными. Выбор типа цикла зависит от задачи программирования и личных предпочтений кодирования. Для циклов применяются инструкции: goto, for, while, do-while

Одним из основных отличий Java от других языков, таких как C++, является цикл foreach, разработанный для упрощения итерации по массиву или коллекции.

8.1. Инструкция for

Это цикл с заданным числом повторений. В нем изменение индекса цикла заложено в инструкцию. Задаются - начальное значение индекса, условие выполнения, правило изменения индекса после итерации. Разделители для параметров инструкции for – точка с запятой (;).

Синтаксис инструкции:

```
for (индекс цикла = начало; условие выполнения; изменение индекса)
{
    Инструкции тела цикла;
}
```

Пример. Программа использует цикл, в котором в консоль выводится последовательность чисел от 0 до 9.

```
public Class Labels {
    public static int Main()  {
        for (int i = 0; i < 10; i++){
            System.out.println ("i = {0 } ", i);
        }
    }
}
```

8.2. Инструкция while

Это цикл с неизвестным числом повторений с предусловием. Тело цикла повторяется, пока выполняется условие. Тело цикла первый раз выполняется с проверкой условия. Ее синтаксис выглядит следующим образом:

```
while (Условие)
{
    Инструкции тела цикла;
}
```

Пример. Программа использует цикл, в котором в консоль выводится последовательность чисел от 0 до 9.

```
public Class CycleWhile
{
    public static int Main()  {
        int i = 0;
        while (i < 10) {
            System.out.println("i = "+ i);
            i = i + 1;
        }
    }
}
```

8.3. Инструкция do-while

Это цикл с неизвестным числом повторений с постусловием. Тело цикла повторяется, пока выполняется условие. Тело цикла первый раз выполняется без проверки условия. Выход из цикла при не выполнении условия.

Эта циклическая инструкция работает по принципу: «Пока выполняется условие — повторить». Ее синтаксис выглядит следующим образом:

```
do
{
    Инструкции тела цикла;
}
while (Условие)
```

Пример. Программа использует цикл, в котором в консоль выводится последовательность чисел.

```
public Class DoWhile {
    public static int Main()  {
        int i = 0;
        do  {
            System.out.println("i =  "+ i);
            i = i + 1;
            }
        while (i < 10)
    }
}
```

8.4. Безусловные переходы

Бывают ситуации, когда необходимо прекратить выполнение цикла досрочно (до того как перестанет выполняться условие), не прерывая при этом цикла. Для таких случаев удобно использовать инструкции break и continue.

8.4.1. Инструкция break

Если вы хотите на каком-то шаге цикла его прекратить, не выполняя до конца описанные в нем действия, то лучше всего использовать break. Следующий пример иллюстрирует его работу.

Программа выполняет поиск не простых чисел (которые имеют делитель без остатка) в интервале от 2 до 11. В программе используется два цикла for.

Первый цикл перебирает все числа делимого от 11 до 2. Переменная i инициализируется значением 11 и затем уменьшается на 1 с каждой итерацией. Второй цикл перебирает все числа делителя от (j = i -1) до 2.

В теле внутреннего цикла проверяется условие: делится ли число i на число j без остатка. Если условие выполняется, то число i нельзя отнести к разряду простых чисел, и флаг, определяющий число как простое, устанавливается в false. Дальнейший поиск не имеет смысла. Цикл завершается инструкцией break.

```
Class Program {
    static void Main()    {
        bool IsPrimeNumber = true;  // устанавливаем флаг простого числа
        for (int i = 11; i > 1; i--)  {
            for (int j = i - 1; j > 1; j--) {
                IsPrimeNumber = true;
                // если есть делитель с нулевым остатком, сбрас флага
                if (i % j == 0) {
                    IsPrimeNumber = false; // проверка бессмысленна
                    System.out.println ("не простое число"+ i);
                    System.out.println ("Нажмите Enter");
                }
                if (IsPrimeNumber == false) break;
            }
        }
    }
}
```

8.4.2. Инструкция continue

Она в отличие от break не прерывает хода выполнения цикла. Она лишь приостанавливает текущую итерацию и переходит к следующей итерации.

Пример. В нем в консоль выводятся нечетные числа в заданном диапазоне. Поверка осуществляется проверкой остатка от деления на 2, для нечетных чисел он не равен 0. В цикле перебираются все числа от 1 до 100. Если очередное число четное, то итерация завершается с пропуском последующих инструкций тела цикла и переходом к следующей итерации.

```
Class PoiskNechet {
    static void Main() {
        for ( int i = 100; i > 0; i--) {
            if ( i%2 ==0) {
                continue;
            }
            System.out.println ("{0} - нечетное число", i);
            System.out.println("Нажмите Enter");
        }
    }
}
```

8.5. Вечные циклы

При написании приложений с использованием циклов следует остерегаться зацикливания программы. Зацикливание — это ситуация, при которой условие выполнения цикла всегда истинно и выход из цикла невозможен.

9. Обработка ошибок и исключений

9.1. Введение

Если во время выполнения программы что-то работает неправильно, создается исключение. Исключение останавливает текущий поток программы и если никакие меры не предпринимаются, программа просто прекращает выполнение. Причиной исключений могут быть ошибки в программе (например, деление числа на ноль) или неожиданный ввод (например, выбор несуществующего файла). Задачей программиста является предоставление программе возможности устранить проблемы, не приводя к сбою.

Мы можем перехватить и обработать исключение в программе. При описании обработки применяется бейсбольная терминология. Говорят, что исполняющая система или программа "**выбрасывает**" (throws) объект-исключение. Этот объект "пролетает" через всю программу, появившись сначала в том методе, где произошло исключение, а программа в одном или нескольких местах пытается (try) его "перехватить" (catch) и обработать. Обработку можно сделать полностью в одном месте, а можно обработать исключение в одном месте, выбросить снова, перехватить в другом месте и обрабатывать дальше.

Хорошо написанные ООП программы обязательно должны обрабатывать все возникающие в них исключительные ситуации.

9.2. Классы исключений

В Java определена иерархия классов стандартных исключений. Все классы-исключения расширяют класс Throwable - непосредственное расширение класса Object. У класса Throwable и у всех его расширений 2 конструктора:

- Throwable() - конструктор по умолчанию;
- Throwable (String message) - создаваемый объект будет содержать произвольное сообщение message.

Записанное в конструкторе сообщение можно получить затем методом getMessage(). Если объект создавался конструктором по умолчанию, то данный метод возвратит null. Метод toString() возвращает краткое описание события, именно он работал в предыдущих листингах. 3 метода выводят сообщения обо всех методах, встретившихся по пути "полета" исключения:

- printStackTrace() - выводит сообщения в стандартный вывод, как правило, это консоль;
- printStackTrace(PrintStream stream) - выводит сообщения в байтовый поток stream;

- printStackTrace(PrintWriter stream) - выводит сообщения в символьный поток stream.

У класса Throwable два непосредственных наследника - классы Error и Exception. Они не добавляют новых методов, а служат для разделения классов-исключений на два больших семейства - семейство классов-ошибок (Error) и семейство собственно классов-исключений (Exception). Класс Exception имеет свои расширения, основными являются расширения класса RuntimeException.

Иерархия классов исключений:

- Error. Это класс серьезных ошибок, которые программист не может перехватывать. Расширеня класса:
  - **AbstractMethodError**. Вызван абстрактный метод. Это может произойти лишь в очень редких случаях, так что вы никогда не столкнетесь с этим исключением.
  - **ClassFormatError**. Загружаемый класс или интерфейс имеет неверный формат (обычно это связано с использованием “преобразованных” (mangled) имен).
  - **IllegalAccessError**. Исключение неразрешенного доступа.
  - **IncompatibleClassChangeError**. При загрузке класса или интерфейса было обнаружено изменение, несовместимое с информацией об этом классе или интерфейсе. Например, в период времени между компиляцией класса и компиляцией использующей его программы, из класса был удален незакрытый метод.
  - **InstantiationError**. Интерпретатор попытался создать объект абстрактного класса или интерфейса.
  - **InternalError**. Произошел внутренний сбой runtime-системы. В нормальных условиях такая ошибка не должна возникнуть.
  - **LinkageError**. Исключения класса LinkageError и его подклассов означают, что класс тем или иным образом зависит от другого класса и что связь между ними не может быть установлена.
  - **NoClassDefFoundError** extends LinkageError. Нужный класс не найден.
  - **NoSuchFieldError**. Поле отсутствует в классе или интерфейсе.
  - **NoSuchMethodError**. Метод отсутствует в классе или интерфейсе.
  - **OutOfMemoryError**. Нехватка памяти.
  - **StackOverflowError**. Переполнение стека. Может свидетельствовать о бесконечной рекурсии.
  - **ThreadDeath**. Исключение ThreadDeath возбуждается потоком-“жертвой” при его уничтожении методом thread.stop. Если исключение ThreadDeath перехватывается, его необходимо возбудить повторно,

чтобы поток был уничтожен. Если ThreadDeath не перехватывается, то обработчик ошибок верхнего уровня не выводит никаких сообщений.

- **UnknownError**. Произошла неизвестная, но серьезная ошибка.
- UnsatisfiedLinkError. Ошибка связывания внутри родного метода. Обычно это означает, что библиотека, реализующая родной метод, содержит неопределенные символы, которые не были найдены ни в одной библиотеке.
- **VerifyError**. Произошла ошибка верификации — то есть во время загрузки класс не прошел проверку, в ходе которой обычно выясняется не нарушает ли класс каких-нибудь требований безопасности Java.
- **VirtualMachineError**. Нарушена работа виртуальной машины, или наблюдается нехватка ресурсов.

- RuntimeException. Это класс ошибок среды выполнения, которые программист может перехватывать и обрабатывать. Расширеня класса:
  - **ArithmeticException**. Основной класс исключений, происходящих при выполнении арифметических операций.
  - **ClassCastExeption**. Попытка сохранения в массиве неверного типа.
  - **IllegalArgumentException**. Метод получил неверный аргумент (например, метод String.equals вызван для объекта, который не относится к типу String).
  - **IllegalMonitorStateException**. Механизм wait/notify (ждать/уведомить) использован за пределами синхронного кода.
  - **IllegalThreadStateException**. Состояние потока не допускает выполнения требуемой операции.
  - **IndexOutOfBoundsException**. Runtime-система генерирует это исключение при выходе индекса массива или объекта String за пределы диапазона допустимых значений.
  - **NegativeArraySizeException**. Попытка создания массива отрицательного размера.
  - **NullPointerException**. Для доступа к полю или методу использована null-ссылка. Это же исключение сигнализирует о передаче методу параметра null, если для данного параметра это значение является недопустимым.
  - **NumberFormatException**. Неверное содержимое строки, в которой должно было находиться число. Исключение возбуждается такими методами, как Integer.parseInt.
  - **SecurityException**. Попытка выполнения действия, запрещенного системой безопасности - обычно объектом SecurityManager для текущего runtime-контекста.

Классы-ошибки, расширяющие класс Error, свидетельствуют о возникновении сложных ситуаций в виртуальной машине Java. Их обработка требует глубоко-

го понимания всех тонкостей работы JVM. Имена классов-ошибок, по соглашению, заканчиваются словом Error.

Классы-исключения, расширяющие класс Exception, отмечают возникновение обычной нештатной ситуации, которую можно и даже нужно обработать. Такие исключения следует выбросить оператором throw. Классов-исключений очень много, более 200. Они разбросаны буквально по всем пакетам J2SDK. В большинстве случаев вы способны подобрать готовый класс-исключение для обработки исключительных ситуаций в своей программе. При желании можно создать и свой класс-исключение, расширив класс Exception или любой его подкласс.

Среди классов-исключений выделяется класс RuntimeException - прямое расширение класса Exception. В нем и его подклассах отмечаются исключения, возникшие при работе JVM, но не столь серьезные, как ошибки. Их можно обрабатывать, но лучше доверить это JVM, поскольку чаще всего это просто ошибка в программе, которую надо исправить.

Имена классов-исключений, по соглашению, заканчиваются словом Exception.

9.3. Перехват и огбработка исключений

9.3.1. Блоки try, catch

Блоки try{} и catch(){} используются вместе. Если предполагается, что блок кода может вызвать исключение, то воспользуйтесь блоком try{} для инициализации исключения и блоком catch(){} для кода обработчика исключения,. При отсутствии блоков try и catch произойдет сбой программы.

Синтаксис:

```
try
{
   // Проверяемый код здесь
}
catch (КлассИсключения ИмяИсключения)
{
   // Код обработчика исключения здесь.
}
```

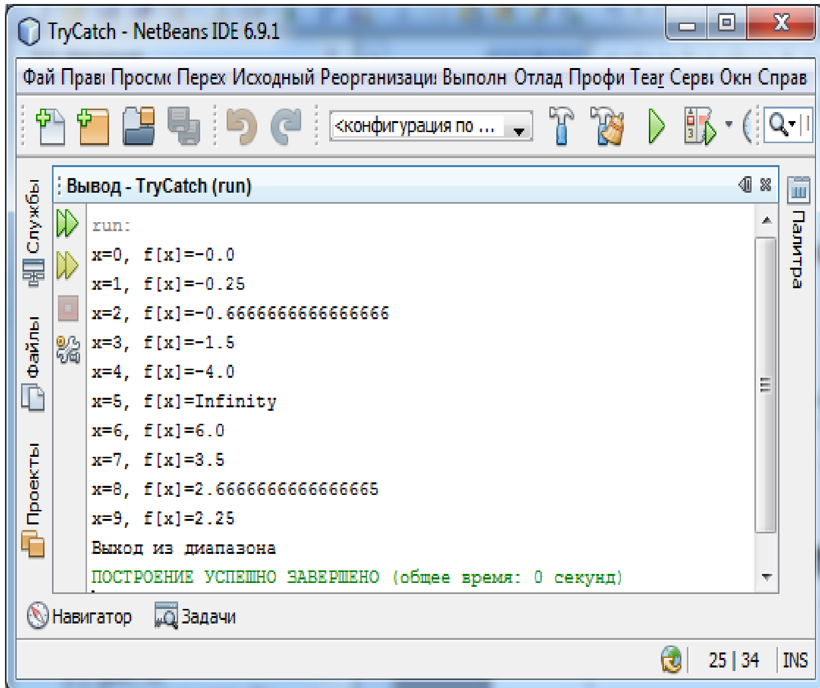
Если в конструкции обработки исключений используется несколько блоков catch(){}, классы исключений нужно перечислять в них последовательно, от менее общих к более общим. Если более общий идет первым, то менее общие не будут исполнены.

Пример. Обработка исключений попытки деления на 0 и выхода из жиапазона массива. Вычисляется функция f[x] = 1 / (x-5).

Листинг программы

```
package trycatch;
public class Main {
    public static void main(String[] args) {
        int k=10;
        double[] f = new double[10];          // Массив размером 10 (0-9)
        try {                                 // Инициализация исключения
            for (int x = 0; x < k + 1; x++) {
                f[x] = (double) x/(x-5);    // Вычисление элемента массива
                System.out.println("x="+x+", f[x]=" + f[x]);
            }
        }
            // Перехватить попытку деления на 0
        catch (ArithmeticException ae) {
            System.out.println("Попытка деления на 0");
        }
            // Перехватить выход из диапазона
        catch (ArrayIndexOutOfBoundsException ai) {
            System.out.println("Выход из диапазона");
        }
    }
}
```

Вывод в консоль



Результаты:

- Деление на 0 – для вещественных чисел не ошибка. Ответ бесконечность.
- Выход за пределы массива при x=10.

9.3.2. Блоки try, catch, finally

Код, содержащийся в блоке finally{}, выполняется всегда, вне зависимости от возникновения исключения. Используется, например, чтобы гарантировать возвращение ресурсов, убедиться, что файл закрыт.

Синтаксис:

```
try
{
   // Проверяемый код здесь
}
catch (КлассИсключения ИмяИсключения)
{
   // Код обработчика исключения здесь.
}
```

```
finally
{
   // Код, выполняемый после try (и возможно catch)
}
```

Итак, каждый блок catch(){} перехватывает один определенный тип исключений. Если требуется одинаково обработать несколько типов исключений, то можно воспользоваться тем, что классы-исключения образуют иерархию.

Листинг. Обработка нескольких типов исключений

```
Class SimpleExt4 {
     public static void main(String[] args) {
          try {
               int n = Integer.parseInt(args[0]);
               System.out.println("After parseInt());
               System.out.println(" 10 / n = " + (10 / n) ) ;
               System.out.println("After results output");
          }
          catch(RuntimeException ae) {
               System.out.println("From Run.Exc. catch: "+ae);
          }
          Finally {
               System.out.println("From finally");
          }
          System.out.println("After all actions");
     }
}
```

В листинге два блока catch(){} заменены одним блоком, перехватывающим исключение класса RuntimeException. Этот блок перехватывает оба исключения, потому что это исключения подклассов класса RuntimeException.

Таким образом, перемещаясь по иерархии классов-исключений, мы можем обрабатывать сразу более или менее крупные совокупности исключений. Рассмотрим подробнее иерархию классов-исключений.

9.3.3. Оператор throw

Этот оператор очень прост: после слова throw через пробел записывается объект класса-исключения. Достаточно часто он создается прямо в операторе throw, например:

```
throw new ArithmeticException();
```

Оператор можно записать в любом месте программы. Он немедленно выбрасывает записанный в нем объект-исключение и дальше обработка этого исключения идет как обычно, будто бы здесь произошло деление на нуль или другое действие, вызвавшее исключение класса ArithmeticException.

9.3.4. Порядок обработки исключений

Блоки catch(){} перехватывают исключения в порядке написания этих блоков. Это правило приводит к интересным результатам.

В листинге мы записали 2 блока перехвата catch(){} и оба блока выполнялись при возникновении соответствующего исключения. Это происходило по тому, что классы-исключения ArithmeticException и ArrayIndexOutOfBoundsException находятся на разных ветвях иерархии исключений. Иначе обстоит дело, если блоки catch(){} перехватывают исключения, расположенные на одной ветви. Например, если есть блок catch(){}, перехватывающий RuntimeException, то следующий за ним блок catch(){}, обрабатывающий выход индекса за пределы, не будет выполняться, поскольку исключение этого типа является, к тому же, исключением общего типа RuntimeException и будет перехватываться предыдущим блоком catch(){}.

```
 try {
     // Операторы, вызывающие исключения
}
catch(RuntimeException re) {
     // Какая-то обработка
}
catch(ArrayindexOutofBoundsException ae) {
     // Никогда не будет выполнен!
}
```

9.3.5. Создание собственных исключений

Прежде всего, нужно четко определить ситуации, в которых будет возникать ваше собственное исключение, и подумать, не станет ли его перехват невольно перехватывать также и другие, не учтенные вами исключения.

Потом надо выбрать суперкласс создаваемого класса-исключения. Им может быть класс Exception или один из его многочисленных подклассов.

После этого можно написать класс-исключение. Его имя, по соглашению, должно завершаться словом Exception. Как правило, этот класс состоит только из 2 конструкторов и переопределения методов toString() и getMessage().

Рассмотрим пример. Пусть метод handle(int cipher) обрабатывает арабские цифры 0—9, которые передаются ему в аргументе cipher типа int. Мы хотим выбросить исключение, если аргумент cipher выходит за диапазон 0—9.

Прежде всего, убедимся, что такого исключения нет в иерархии классов Exception и не отслеживается более общая ситуация попадания целого числа в какой-то диапазон. Это так, поэтому будем расширять наш класс. Назовем его CipherException, прямо от класса Exception. Определим класс CipherException, как показано в листинге, и используем его в классе ExceptDemo.

Листинг. Создание класса-исключения

```
Class CipherException extends Exception {
    private String msg;
    CipherException() {
        msg = null;
    }
    CipherException(String s){ msg = s;}
    public String toString()  {
        return "CipherException (" + msg + ");
    }
}
Class Except Demo {
    static public void handle(int cipher) throws CipherException {
        System.out.pnntln("handle()'s beginning");
        if (cipher < 0 || cipher > 9)
        throw new CipherException("" + cipher);
        System.out.println("handle()'s ending");
    }
    public static void main(String[] args) {
        try  {
            handle(1) ;
            handle(10);
        }
        catch(CipherException ce) {
            System.out.println("caught " + ce) ;
            ce.printStackTrace();
        }
    }
}
```

9.3.6. Заключение

Обработка исключительных ситуаций стала сейчас обязательной частью ООП программ. Применяя методы классов J2SDK и других пакетов, обращайте внимание на то, какие исключения они выбрасывают, и обрабатывайте их. Исключения резко меняют ход выполнения программы, делают его запутанным. Не увлекайтесь сложной обработкой, помните о принципе KISS.

Например, из блока finally{} можно выбросить исключение и обработать его в другом месте. Подумайте, что произойдет в этом случае с исключением, возникшем в блоке try{}? Оно нигде не будет перехвачено и обработано.

10. Работа со строками

Очень большое место в обработке информации занимает работа с текстами. Как и многое другое, текстовые строки в языке Java являются объектами. Они представляются экземплярами классов

- Класс String. Это строки-константы неизменной длины и содержания. Это значительно ускоряет обработку строк и позволяет экономить память, разделяя строку между объектами, использующими ее.
- Класс StringBuffer. Это строки переменной длины и содержания. Длину строк, хранящихся в объектах класса stringBuffer, можно менять, вставляя и добавляя строки и символы, удаляя подстроки или сцепляя несколько строк в одну строку. Во многих случаях, когда надо изменить длину строки типа string, компилятор Java неявно преобразует ее к типу stringBuffer, меняет длину, потом преобразует обратно в тип string

Например, следующее действие

```
String s = "Это" + " одна " + "строка";
```

компилятор выполнит так:

```
String s = new StringBuffer().append("Это").append(" одна ")
    .append("строка").toString();
```

Будет создан объект класса StringBuffer, в него последовательно добавлены строки "Это", " одна ", "строка", и получившийся объект класса StringBuffer будет приведен к типу String методом toString ().

Напомним, что символы в строках хранятся в кодировке Unicode, в которой каждый символ занимает два байта. Тип каждого символа char.

10.1. Класс String

10.1.1. Создание строк

Перед работой со строкой ее следует создать. Это можно сделать разными способами.

Самый простой способ создать строку — это организовать ссылку типа String на строку-константу:

```
String si = "Это строка.";
```

Если константа длинная, можно записать ее в нескольких строках текстового редактора, связывая их операцией сцепления:

```
String s2 = "Длинная строка, " + "записанная в двух строках исходного текста";
```

Замечание. Не забывайте разницу между пустой строкой String s = "", не содержащей ни одного символа, и пустой ссылкой String s = null, не указывающей ни на какую строку и не являющейся объектом.

Самый правильный способ создать объект с точки зрения ООП — это вызвать его конструктор в операции new. Класс String имеет 9 конструкторов:

- string() — создается объект с пустой строкой;
- string(String str) — из одного объекта создается другой, поэтому этот конструктор используется редко;
- string (StringBuf fer str) — преобразованная копия объекта класса BufferString;
- string(byte[] byteArray) — объект создается из массива байтов byteArray;
- string(char [] charArray) — объект создается из массива charArray символов Unicode;
- string(byte [] byteArray, int offset, int count) — объект создается из части массива байтов byteArray, начинающейся с индекса offset и содержащей count байтов;
- string(char [] charArray, int offset, int count) — то же, но массив состоит из символов Unicode;
- string(byte[] byteArray, String encoding) — символы, записанные в массиве байтов, задаются в Unicode-строке, с учетом кодировки encoding ;
- string(byte[] byteArray, int offset, int count, String encoding) — то же самое, но только для части массива.

При неправильном заданий индексов offset, count или кодировки encoding возникает исключительная ситуация.

Конструкторы, использующие массив байтов byteArray, предназначены для создания Unicode-строки из массива байтовых ASCII-кодировок символов. Такая ситуация возникает при чтении ASCII-файлов, извлечении информации из базы данных или при передаче информации по сети.

В самом простом случае компилятор для получения двухбайтовых символов Unicode добавит к каждому байту старший нулевой байт. Получится диапазон '\u0000' — '\u00ff' кодировки Unicode, соответствующий кодам Latin 1. Тексты на кириллице будут выведены неправильно.

Если же на компьютере сделаны местные установки, как говорят на жаргоне "установлена локаль" (locale) (в MS Windows это выполняется утилитой Regional Options в окне Control Panel), то компилятор, прочитав эти установки, создаст символы Unicode, соответствующие местной кодовой странице. В русифицированном варианте MS Windows это обычно кодовая страница CP1251.

Если исходный массив с кириллическим ASCII-текстом был в кодировке CP1251, то строка Java будет создана правильно. Кириллица попадет в свой диапазон '\u0400'—'\u04FF' кодировки Unicode.

Но у кириллицы есть еще, по меньшей мере, 4 кодировки.

- В MS-DOS применяется кодировка CP866.
- В UNIX обычно применяется кодировка KOI8-R.
- На компьютерах Apple Macintosh используется кодировка MacCyrillic.
- Есть еще и международная кодировка кириллицы ISO8859-5;

Например, байт 11100011 (0xE3 в шестнадцатеричной форме) в кодировке CP1251 представляет кириллическую букву Г, в кодировке CP866 - букву У, в кодировке KOI8-R - букву Ц, в ISO8859-5 - букву у, в MacCyrillic - букву г.

Если исходный кириллический ASCII-текст был в одной из этих кодировок, а местная кодировка CP1251, то Unicode-символы строки Java не будут соответствовать кириллице.

В этих случаях используются последние два конструктора, в которых параметром encoding указывается, какую кодовую таблицу использовать конструктору при создании строки.

10.1.2. Сцепление строк

Со строками можно производить операцию сцепления строк (concatenation), обозначаемую знаком плюс (+). Эта операция создает новую строку, просто составленную из состыкованных первой и второй строк. Ее можно применять и к константам, и к переменным. Например:

```
String attention = "Внимание: ";
String s = attention + "неизвестный символ";
```

Вторая операция - присваивание (+=) применяется к переменным в левой части:

```
attention += s;
```

Поскольку операция (+) перегружена со сложения чисел на сцепление строк, встает вопрос о приоритете этих операций. У сцепления строк приоритет выше, чем у сложения, поэтому, записав "2" + 2 + 2, получим строку " 222 ". Но, записав 2 + 2 + "2", получим строку "42", поскольку действия выполняются слева направо. Если же запишем "2" + (2 + 2), то получим "24".

10.1.3. Длина строки

Для того чтобы узнать длину строки, т.е. количество символов в ней, надо обратиться к методу length():

```
String s = "Write once, run anywhere.";
int len = s.length();
```

или еще проще

```
int len = "Write once, run anywhere.".length();
```

поскольку строка-константа - полноценный объект класса String? К ней можно применять методы класса. Заметьте, что строка - это не массив, у нее нет поля length. Поэтому использован метод класса.

10.1.4. Символы строки

Удаление символа

Метод deieteCharAt(int ind) удаляет символ с указанным индексом ind. Длина строки уменьшается на единицу. Если индекс ind отрицателен или больше длины строки, возникает исключительная ситуация.

Выборка 1 символа из строки.

charAt(int ind).

Выборка 1 символа с индексом ind (нумерация с 0). Если индекс ind отрицателен или не меньше, чем длина строки, возникает исключительная ситуация. Например, после определения

```
String s = "Write once, run anywhere.";
char ch = s.charAt(3);
```

переменная ch будет иметь значение 't'.

Копирования символов строки в массив символов.

Метод toCharArray(). Все символы строки в виде массива символов можно получить методом toCharArray(), возвращающим массив символов. Размер массива = числу символов в строке-исходнике.

Вставка символов из строки в существующий массив символов.

```
getChars(int begin, int end, char[] dst, int ind).
```

Здесь

- int begin, начальный номер в строке исходнике.

- int end, конечный номер в строке исходнике.
- char[] dst, символьный массив получатель.
- int ind). начальный номер в символьный массиве получателе.

В массив dist будет записано (end – begin) символов, которые займут позиции в массиве, начиная с индекса ind. Этот метод создает исключительную ситуацию в следующих случаях:

- ссылка dst = null;
- индекс begin отрицателен;
- индекс begin больше индекса end ;
- индекс end больше длины строки;
- индекс ind отрицателен;
- ind + (end — begin) > dst.length.

Например, после выполнения

```
char[] ch = ('К', 'о', 'р', 'о', 'л', 'ь', ' ', 'л', 'е', 'т', 'а'};
    "Забыла найти".getChars(2, 6, ch, 2);
```

Из строки извлекаются символы с номерами от 2 до 5 - была. В результате массив ch будет таков:

```
    ch = ('К', 'о', 'б', 'ы', 'л', 'а ', ' ', 'л', 'е', 'т', 'а'};
```

Поиск символа в строке.

Поиск всегда ведется с учетом регистра букв. Первое появление символа ch в данной строке this можно отследить методом indexOf(int ch), возвращающим индекс этого символа в строке или -1, если символа ch в строке this нет.

Например, "Молоко", indexOf('о') выдаст в результате 1.

Метод выполняет в цикле последовательные сравнения this.charAt(k++> == ch, пока не получит значение true. Второе и следующие появления символа ch в данной строке this можно отследить методом indexOf(int ch, int ind).

Этот метод начинает поиск символа ch с индекса ind. Если ind < о, то поиск идет с начала строки, если ind больше длины строки, то символ не ищется, т.е. возвращается -1.

Например, "Молоко".indexOf('о', indexOf ('о') + 1)   даст в результате 3..

Последнее появление символа ch в данной строке this отслеживает метод lastIndexOf(int ch). Он просматривает строку в обратном порядке. Если символ ch не найден, возвращается -1.

Например, "Молоко".lastIdexOf('о') даст в результате 5.

Предпоследнее и предыдущие появления символа ch в данной строке this можно отследить методом lastIndexOf (int ch, int ind), который просматривает строку в обратном порядке, начиная с индекса ind.

Если ind больше длины строки, то поиск идёт от конца строки, если ind < 0, то возвращается -1.

Замена отдельного символа.

Метод replace(int old, int new) возвращает новую строку, в которой все вхождения символа old заменены символом new. Если символа old в строке нет, то возвращается ссылка на исходную строку.

Например, после выполнения " Рука в руку сует хлеб", replace ('у', 'е') получим строку " Река в реке сеет хлеб".

Регистр букв при замене учитывается.

Удаление пробелов в начале и конце строки.

Метод trim() возвращает новую строку, в которой удалены начальные и конечные символы с кодами, не превышающими '\u0020 '.

10.1.5. Подстроки

Добавление подстроки.

В классе stringBuffer есть 10 методов append(), добавляющих подстроку в конец строки. Они не создают новый экземпляр строки, а возвращают ссылку на ту же самую, но измененную строку.

Основной метод append(string str) присоединяет строку str в конец данной строки. Если ссылка str = null, то добавляется строка "null".

6 методов append(type elem) добавляют примитивные типы Boolean, char, int, long, float, double, преобразованные в строку.

2 метода присоединяют к строке массив str и подмассив sub символов, преобразованные в строку: append(char [] str) и append (char [], sub, int offset, int len).

10 метод добавляет просто объект append(Object obj). Перед этим объект obj преобразуется в строку своим методом toString().

Выделение подстроки.

Метод substring(int begin, int end) выделяет подстроку от символа с индексом begin включительно до символа с индексом end исключительно. Длина подстроки будет равна end - begin.

Метод substring(int begin) выделяет подстроку от индекса begin включительно до конца строки.

Если индексы отрицательны, индекс end больше длины строки или begin больше чем end, то возникает исключительная ситуация.

Например, после выполнения

```
String s = "Write.once, run anywhere.";
String sub1 = s.substring(6, 10);
String sub2 = s.substring(16);
```

получим в строке sub1 значение "once ", а в sub2 — значение "anywhere ".

Вставка подстроки.

10 методов insert() предназначены для вставки строки, указанной параметром метода, в данную строку. Место вставки задается первым параметром метода ind. Это индекс элемента строки, перед которым будет сделана вставка. Он должен быть неотрицательным и меньше длины строки, иначе возникнет исключительная ситуация. Строка раздвигается, емкость буфера при необходимости увеличивается. Методы возвращают ссылку на ту же,но преобразованную строку.

Основной метод insert(int ind, String str) вставляет строку str в данную строку перед ее символом с индексом ind. Если ссылка str = null вставляется строка "null".

Например, после выполнения

```
String s = new StringBuffer("Это большая строка"). insert(4, "не").toString();
```

получим s = "Это небольшая строка".

Метод sb.insert(sb.length(), "xxx") будет работать так же, как sb.append("xxx").

6 методов insert(int ind, type elem) вставляют примитивные типы Boolean, char, int, long, float, double, преобразованные в строку.

2 метода вставляют массив str и подмассив sub символов, преобразованные в строку:

i nsert(int ind, chart] str)
insert(int ind, char[] sub, int offset, int len)

10 метод вставляет просто объект:

```
insert(int ind, Object obj)
```

Объект obj перед добавлением преобразуется в строку методом toString().

Поиск подстроки.

Поиск всегда ведется с учетом регистра букв.

Первое вхождение подстроки sub в данную строку this отыскивает метод indexOf(String sub). Он возвращает индекс первого символа первого вхождения подстроки sub в строку или -1, если подстрока sub не входит в строку this. Например, " Раскраска ".indexof ("рас") даст в результате 1.

Если вы хотите начать поиск не с начала строки, а с какого-то индекса ind, используйте метод indexOf(String sub, int ind). Если ind < 0, то поиск идет с начала строки, если ind больше.длины строки, то символ не ищется, т.е. возвращается -1.

Последнее вхождение подстроки sub в данную строку this можно отыскать методом lastIndexOf(string sub), возвращающим индекс первого символа последнего вхождения подстроки sub в строку this или (-1), если подстрока sub не входит в строку this.

Последнее вхождение подстроки sub не во всю строку this, а только в ее начало до индекса ind можно отыскать методом lastIndexOf(String stf, int ind). Если ind больше длины строки, то.поиск идет от конца строки, если ind < 0, то возвращается -1.

Для того чтобы проверить, не начинается ли данная строка this с подстроки sub, используйте логический метод startsWith(string sub), возвращающий true, если данная строка this начинается с подстроки sub, или совпадает с ней, или подстрока sub пуста.

Можно проверить и появление подстроки sub в данной строке this, начиная с некоторого индекса ind логическим методом startsWith(String sub,int ind). Если индекс ind отрицателен или больше длины строки, возвращается false.

Для того чтобы проверить, не заканчивается ли данная строка this подстрокой sub, используйте логический метод endsWith(String sub). Он возвращает true, если подстрока sub совпадает со всей строкой или подстрока sub пуста.

Например, if (fileName.endsWith(".Java")) отследит имена файлов с исходными текстами Java.

Перечисленные методы создают исключительную ситуацию, если sub = null.

Совет. Если вы хотите осуществить поиск, не учитывающий регистр букв, измените предварительно регистр всех символов строки.

Замена подстроки

Метод replace(int begin, int end. String str) удаляет символы из строки str, начиная с индекса begin включительно до индекса end исключительно. Если end больше длины строки, то метод удаляет символы до конца строки и вставляет вместо них строку str.

Если begin отрицательно, больше длины строки или больше end, возникает исключительная ситуация.

Метод replace() - последовательное выполнение методов delete() и insert().

10.1.6. Сравнение строк

Операция сравнения (==) сопоставляет только ссылки на строки. Она выясняет, указывают ли ссылки на одну и ту же строку. Например, для строк

```
String s1 = "Какая-то строка";
String s2 = "Другая-строка";
```

сравнение s1 == s2 дает в результате false.

Значение true получится, только если обе ссылки указывают на одну и ту же строку, например, после присваивания s1= s2.

Для сравнения не ссылок, а содержимого строк есть несколько методов.

- Логический метод equals(Object obj), переопределенный из класса Object, возвращает true, если аргумент obj не равен null, является объектом класса string, и строка, содержащаяся в нем, полностью идентична данной строке вплоть до совпадения регистра букв. В остальных случаях возвращается значение false.
- Логический метод equalsIgnoreCase(Object obj) работает так же, но одинаковые буквы, записанные в разных регистрах, считаются совпадающими.
- Метод compareTo(string str) возвращает целое число типа int, вычисленное по следующим правилам:
  - Сравниваются символы данной строки this и строки str с одинаковым индексом, пока не встретятся различные символы с индексом, допустим k, или пока одна из строк не закончится.
  - В первом случае возвращается значение this.charAt(k) - str.charAt(k), т.е. разность кодировок Unicode первых несовпадающих символов.
  - Во втором случае возвращается значение this.length() - str.length(), т.е. разность длин строк.
  - Если строки совпадают, возвращается 0.
  - Если значение str равно null, возникает исключительная ситуация.
  - Нуль возвращается в той же ситуации, в которой метод equals() возвращает true.

- Метод compareToIgnoreCase(string str) производит сравнение без учета регистра букв.
- Метод compareTo(Object obj) создает исключительную ситуацию, если obj не является строкой. В остальном он работает как метод compareTo(String str).

Эти методы не учитывают алфавитное расположение символов в локальной кодировке.

Русские буквы расположены в Unicode по алфавиту, за исключением одной буквы. Заглавная буква Ё расположена перед всеми кириллическими буквами, ее код '\ u040l ', а строчная буква е — после всех русских букв, ее код '\ u0451 '.

10.1.7. Регистр букв

Метод toLowerCase() возвращает новую строку, в которой все буквы переведены в нижний регистр, т.е. сделаны строчными.

Метод toUpperCase() возвращает новую строку, в которой все буквы переведены в верхний регистр, т.е. сделаны прописными.

При этом используется локальная кодовая таблица по умолчанию. Если нужна другая локаль, то применяются методы с занием локали:

- toLowerCase(Locale loc)
- toUpperCase(Locale loc).

10.1.8. Преобразование другого типа в строку

В языке Java принято соглашение — каждый класс отвечает за преобразование других типов в тип класса String и должен содержать нужные для этого методы.

Класс String содержит 8 статических методов valueOf(type elem) преобразования в строку примитивных типов Boolean, char, int, long, float, double, массива char[] и просто объекта типа Object.

Метод valueOf(char[] ch, int offset, int len) преобразует в строку подмассив массива ch, начинающийся с индекса offset и имеющий len элементов.

Кроме того, в каждом классе есть метод toString(), переопределенный или просто унаследованный от класса Object. Он преобразует объекты класса в строку. Фактически, метод valueOf() вызывает метод toString() соответствующего класса. Поэтому результат преобразования зависит от того, как реализован метод toString().

10.2. Класс StringBuffer

10.2.1. Введение

Объекты класса StringBuffer — это строки переменной длины. Только что созданный объект имеет буфер определенной емкости (capacity), по умолчанию достаточной для хранения 16 символов. Емкость можно задать в конструкторе объекта.

Как только буфер начинает переполняться, его емкость автоматически увеличивается, чтобы вместить новые символы.

В любое время емкость буфера можно увеличить, обратившись к методу ensureCapacity(int minCapacity). Этот метод изменит емкость, только если minCapacity будет больше длины хранящейся в объекте строки.

Емкость будет увеличена по следующему правилу. Пусть емкость буфера равна N. Тогда новая емкость будет равна

Max(2 * N + 2, minCapacity)

Таким образом, емкость буфера нельзя увеличить менее чем вдвое.

Методом setLength(int newLength) можно установить любую длину строки.

Если она окажется больше текущей длины, то дополнительные символы будут равны ' \u0000'. Если она будет меньше текущей длины, то строка будет обрезана, последние символы потеряются, точнее, будут заменены символом '\u0000'. Емкость при этом не изменится.

Если число newLength окажется отрицательным, возникнет исключительная ситуация.

Совет. Будьте осторожны, устанавливая новую длину объекта.

Количество символов в строке можно узнать, как и для объекта класса String, методом length(), а емкость — методом capacity().

Создать объект класса stringBuffer можно только конструкторами.

10.2.2. Конструкторы

В классе StringBuffer 3 конструктора:

- stringBuffer() - создает пустой объект с емкостью 16 символов;
- stringBuffer.(int capacity) - создает пустой объект емкостью capacity;
- StringBuffer (String str) - создает объект емкостью str.length() + 16, содержащий строку str.

10.2.3. Переворот строки

Метод reverse() меняет порядок расположения символов в строке на обратный порядок. Например, после выполнения

```
String s = new StringBuffer("Это небольшая строка");
reverse().toString();
```

получим s == "акортс яашьлобен отЭ".

10.2.4. Парсинг - синтаксический разбор строки

Задача разбора введенного текста - парсинг (parsing) — вечная задача программирования, наряду с сортировкой и поиском. Написана масса программ-парсеров (parser), разбирающих текст по различным признакам.

Но задача остается. И вот очередной программист, отчаявшись найти что-нибудь подходящее, берется за разработку собственной программы разбора.

В пакет java.utii входит простой класс stringTokenizer, облегчающий разбор строк.

10.3. Класс StringTokenizer

Класс StringTokenizer из пакета java.utii небольшой, в нем 3 конструктора и 6 методов.

Конструктор StringTokenizer(String str) создает объект, готовый разбить строку str на слова, разделенные пробелами, символами табуляций '\t', перевода строки '\n' и возврата каретки '\r'. Разделители не включаются в число слов.

Конструктор StringTokenizer(String str. String delimeters) задает разделители вторым параметром deiimeters, например:

```
StringTokenizer("Казнить,нельзя:пробелов-нет", " \t\n\r,:-");
```

Здесь первый разделитель - пробел. Потом идут символ табуляции, символ перевода строки, символ возврата каретки, запятая, двоеточие, дефис. Порядок расположения разделителей в строке deiimeters не имеет значения. Разделители не включаются в число слов.

Конструктор, который позволяет включить разделители в число слов:

```
StringTokenizer(String str, String deiimeters, Boolean flag);
```

Если параметр flag равен true, то разделители включаются в число слов, если false — нет. Например:

```
StringTokenizer("a - (Ь + c) / Ь * c", " \t\n\r+*-/(), true);
```

В разборе строки на слова участвуют методы:

- Метод nextToken() возвращает в виде строки следующее слово;
- Метод hasMoreTokens() возвращает true, если в строке еще есть слова, и false, если слов больше нет.
- Метод countTokens() возвращает число оставшихся слов.
- Метод nextToken(string newDeiimeters) позволяет "на ходу" менять разделители. Следующее слово будет выделено по новым разделителям newDeiimeters; новые разделители действуют далее вместо старых разделителей, определенных в конструкторе или предыдущем методе nextToken().
- Методы nextEiement() и hasMoreEiements() реализуют интерфейс Enumeration. Они просто обращаются к методам nextToken() и hasMoreTokens().

Листинг. Разбиение строки на слова

```
String s = "Строка, которую мы хотим разобрать на слова";
StringTokenizer st = new StringTokenizer(s, " \t\n\r,.");
while(st.hasMoreTokens())
{
    // Получаем слово и что-нибудь делаем с ним, например,
    // просто выводим на экран
    System.out.println(st.nextToken()) ;
}
```

Полученные слова обычно заносятся в какой-нибудь класс-коллекцию: Vector, Stack или другой, наиболее подходящий для дальнейшей обработки текста контейнер.

11. Массивы

Массив - это структура данных, содержащая несколько переменных одного типа. Массивы могут быть одномерными и многомерными.

Одномерные массивы объявляются со следующим синтаксисом.

type[] ИмяМассива;

- type - имя типа значений элементов.
- [ ] – признак массива. Значение в скобках задает размерность массива..

Элементы массива - это обыкновенные переменные своего типа, с ними можно производить все операции, допустимые для этого типа.

Доступ к элементу массива: ИмяМассива [НомерЭлемента].

Индексация массивов начинается с нуля: массив с элементами n индексируется от 0 до n-1.

При инициализации массива значения его элементов помещаются в фигурные скобки и разделяются запятыми.

Многомерные массивы строятся по иерархическому правилу. Элементами массивов могут быть снова массивы. Синтаксис объявления многмерного массива:

type[] [] [] ИмяМассива;

Число пар квадратных скобках – размерность массива.

Внимание. В многомерных массивах количество компонент по разным измерениям может быть разным.

В примере показано создание одномерного и двумерного массивов.

```
Class TestArraysClass
{
    static void Main()
    {
        int[ ] 1Массив1 = new int[5];                    // 1-массив из 5 чисел
        int[ ] 1Массив2 = new int[] { 1, 3, 5, 7, 9 };   // 1-массив инициализирован
        int[ ][ ]  2Массив1 = new int[2][3];             // 2-массив 2x3 чисел
        int[2][3] 2Массив2= { { 1, 2}, { 3, 4, 5 } };    // 2-массив инициализирован
    }
}
```

Описание массива производится в три этапа.

Первый этап - объявление (declaration). На этом этапе определяется только переменная типа ссылка (reference) на массив, содержащая тип массива. Для этого записывается имя типа элементов массива, квадратными скобками указывается, что объявляется ссылка на массив, а не простая переменная, и перечисляются имена переменных типа ссылка, например,

```
double[] a, b;
```

Здесь определены две переменные - ссылки a и b на массивы типа double. Можно поставить квадратные скобки и непосредственно после имени.

Второй этап - определение (installation). На этом этапе указывается количество элементов массива, называемое его длиной, выделяется место для массива в оперативной памяти, переменная-ссылка получает адрес массива.

Все эти действия производятся еще одной операцией языка Java — операцией new тип, выделяющей участок в оперативной памяти для объекта указанного в операции типа и возвращающей в качестве результата адрес этого участка. Например,

```
a = new double[5];
b = new double[100];
ar = new int[50];
```

Индексы массивов всегда начинаются с 0. Массив a состоит из 5 переменных a[0], a[1], … a[4]. Элемента a[5] в массиве нет. Индексы можно задавать любыми целочисленными выражениями, кроме типа long, например, a[i+j], a[i%5], a[++i]. Исполняющая система Java следит за тем, чтобы значения этих выражений не выходили за границы длины массива.

Третий этап - инициализация (initialization). На этом этапе элементы массива получают начальные значения. Например,

```
a[0] = 0.01; a[1] = -3.4; a[2] = 2:.89; a[3] = 4.5; a[4] = -6.7;
```

Первые два этапа можно совместить:

```
doublet] a = new double[5], b = new double[100];
int i = 0, ar[] = new int[50], k = -1;
```

Можно сразу задать и начальные значения, записав их в фигурных скобках через запятую в виде констант или константных выражений. При этом даже необязательно указывать количество элементов массива, оно будет равно количеству начальных значений;

```
double[] a = {0.01, -3.4, 2.89, 4.5, -6.7};
```

Можно совместить второй и третий этап:

```
a = new double [ ] {0.1, 0.2, -0.3, 0.45, -0.02};
```

Ссылка на массив не является частью описанного массива, ее можно перебросить на другой массив того же типа операцией присваивания. Например, после присваивания a = b обе ссылки a и b указывают на один и тот же массив из 100 вещественных переменных типа double и содержат один и тот же адрес.

Ссылка может присвоить "пустое" значение null, не указывающее ни на какой адрес оперативной памяти:

```
ar = null;
```

После этого массив, на который указывала данная ссылка, теряется, если на него не было других ссылок.

Кроме простой операции присваивания, со ссылками можно производить еще только сравнения на равенство, например, a = b, и неравенство, a != b. При этом сопоставляются адреса, содержащиеся в ссылках, мы можем узнать, не ссылаются ли они на один и тот же массив.

Замечание. Массивы в Java всегда определяются динамически, хотя ссылки на них задаются статически.

Кроме ссылки на массив, для каждого массива автоматически определяется целая константа с одним и тем же именем length. Она равна длине массива. Для каждого массива имя этой константы уточняется именем массива через точку.

Последний элемент массива a можно записать так: a [a. length - 1], предпоследний — a [a. length - 2] и т.д..

12. Пакеты

В стандартную библиотеку Java API входят сотни классов. Каждый программист в ходе работы добавляет к ним десятки своих. Множество классов становится необозримым. Принято классы объединять в библиотеки. Но библиотеки классов, кроме стандартной, не являются частью языка.

Разработчики Java включили в язык дополнительную конструкцию — пакеты (packages). Все классы Java распределяются по пакетам. Кроме классов пакеты могут включать в себя интерфейсы и вложенные подпакеты (subpackages). Образуется древовидная структура пакетов и подпакетов.

Эта структура в точности отображается на структуру файловой системы. Все файлы с расширением Class (содержащие байт-коды), образующие пакет, хранятся в одном каталоге файловой системы. Подпакеты собраны в подкаталоги этого каталога.

Каждый пакет образует одно пространство имен (namespace). Это означает, что все имена классов, интерфейсов и подпакетов в пакете должны быть уникальны. Имена в разных пакетах могут совпадать, но это будут разные программные единицы. Таким образом, ни один класс, интерфейс или подпакет не может оказаться сразу в двух пакетах. Если надо использовать два класса с одинаковыми именами из разных пакетов, то имя класса уточняется именем пакета: пакет.класс. Такое уточненное имя называется полным именем класса (fully qualified name).

Пакетами пользуются еще и для того, чтобы добавить к уже имеющимся правам доступа к членам класса private, protected и public еще один, "пакетный" уровень доступа.

Если член класса не отмечен ни одним из модификаторов private, protected, public, то, по умолчанию, к нему осуществляется пакетный доступ (default access), а именно, к такому члену может обратиться любой метод любого класса из того же пакета. Пакеты ограничивают и доступ к классу целиком — если класс не помечен модификатором public, то все его члены, даже открытые, public, не будут видны из других пакетов.

Чтобы создать пакет надо просто в первой строке Java-файла с исходным кодом записать строку package имя;, например:

```
package mypack;
```

Тем самым создается пакет с указанным именем mypack и все классы, записанные в этом файле, попадут в пакет mypack. Повторяя эту строку в начале каждого исходного файла, включаем в пакет новые классы.

Имя подпакета уточняется именем пакета. Чтобы создать подпакет с именем, например, subpack, следует в первой строке исходного файла написать

```
package mypack.subpack;
```

и все классы этого файла и всех файлов с такой же первой строкой попадут в подпакет subpack пакета mypack.

Компилятор Java может сам создать каталог с тем же именем mypack, а в нем подкаталог subpack, и разместить в них Class-файлы с байт-кодами.

Полные имена в пакете будут выглядеть так:

```
mypack.A, mypack.subpack.в.
```

Фирма SUN рекомендует записывать имена пакетов строчными буквами, тогда они не будут совпадать с именами классов, которые, по соглашению, начинаются с прописной. Кроме того, фирма SUN советует использовать в качестве имени пакета или подпакета доменное имя своего сайта, записанное в обратном порядке, например:

```
com.sun.developer
```

Компилятор всегда создает для таких классов безымянный пакет (unnamed package), которому соответствует текущий каталог (current working directory) файловой системы. Вот поэтому у нас Class-файл всегда оказывался в том же каталоге, что и соответствующий Java-файл.

Безымянный пакет служит обычно хранилищем небольших пробных или промежуточных классов. Большие проекты лучше хранить в пакетах.

12.1. Импорт классов и пакетов

Инструкция **import** позволяет сообщить компилятору, где искать классы.

Компилятор будет искать классы только в одном пакете, в том, что указан в первой строке файла. Для классов из другого пакета надо указывать полные имена. Если они короткие,то мы могли бы писать их в листинге - вместо Base его полное имя p1.Base.

Но если полные имена длинные, а используются классы часто, то набирать полные имена утомительно. Вот тут-то мы и пишем инструкции import, указывая компилятору полные имена классов.

Правила использования инструкции import очень просты: пишется слово import и, через пробел, полное имя класса, завершенное точкой с запятой. Сколько классов надо указать, столько инструкций import и пишется.

Это тоже может стать утомительным и тогда используется вторая форма инструкции import - указывается имя пакета или подпакета, а вместо короткого имени класса ставится звездочка *. Этой записью компилятору предписывается просмотреть весь пакет.

```
Import p1.*;
```

Напомним, что импортировать можно только открытые классы, помеченные модификатором public.

Для классов стандартной библиотеки можно пакеты не указывать инструкцией import. Пакет java.iang просматривается всегда, его необязательно импортировать. Остальные пакеты стандартной библиотеки надо указывать в инструкциях import, либо записывать полные имена классов.

Инструкция import вводится только для удобства программистов и слово "импортировать" не означает никаких перемещений классов.

12.2. Java файлы

Теперь можно описать структуру исходного файла с текстом программы на языке Java.

- В первой строке файла может быть необязательный оператор package.
- В следующих строках могут быть необязательные операторы import.
- Далее идут описания классов и интерфейсов.

Еще два правила:

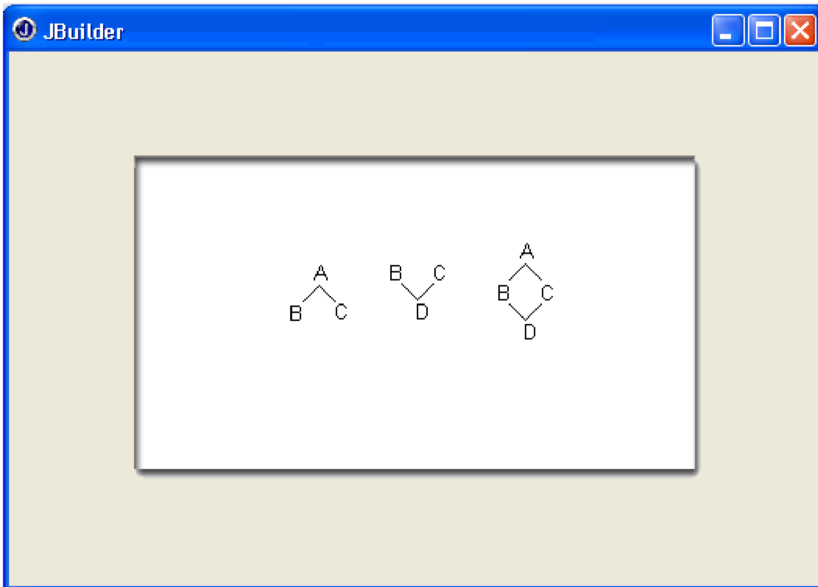
- Среди классов файла может быть только один открытый public класс.
- Имя файла должно совпадать с именем открытого класса, если последний существует.

Отсюда следует, что, если в проекте есть несколько открытых классов, то они должны находиться в разных файлах.

Соглашение "Code Conventions" рекомендует открытый класс, который, если он имеется в файле, описывать первым.

13. Интерфейсы

Получить расширение можно только от одного класса, каждый класс B или C происходит из неполной семьи A, как показано на рисунке слева. Все классы происходят только от класса Object. Но часто возникает необходимость породить класс D от двух классов B и C, как показано на рисунке в центре. Это называется множественным наследованием (multiple inheritance). В множественном наследовании нет ничего плохого. Трудности возникают, если классы B и C сами порождены от одного класса A, как показано на рисунке справа. Это так называемое "ромбовидное" наследование.



Пусть в классе A определен метод f(), к которому мы обращаемся из некоего метода класса D. Можем мы быть уверены, что метод f() выполняет то, что написано в классе A, т.е. это метод A.f()? Может, он переопределен в классах B и C? Если так, то каким вариантом мы пользуемся: B.f() или C.f()?

В разных языках программирования этот вопрос решается по-разному, главным образом, уточнением имени метода f().

Создатели языка Java после долгих споров и размышлений поступили радикально - запретили множественное наследование вообще. При расширении класса после слова extends можно написать только одно имя суперкласса. С

помощью уточнения super можно обратиться только к членам непосредственного суперкласса.

Но что делать, если все-таки при порождении надо использовать несколько предков? Например, у нас есть общий класс автомобилей Automobile, от которого можно породить класс грузовиков Truck и класс легковых автомобилей Car. Но вот надо описать пикап Pickup. Этот класс должен наследовать свойства и грузовых, и легковых автомобилей.

В таких случаях используется еще одна конструкция языка Java - интерфейс. Внимательно проанализировав ромбовидное наследование, теоретики ООП выяснили, что проблему создает только реализация методов, а не их описание.

Интерфейс (Interface), в отличие от класса, содержит только константы и заголовки методов, без их реализации.

Интерфейсы размещаются в тех же пакетах и подпакетах, что и классы, и компилируются тоже в Class-файлы.

Описание интерфейса начинается со слова Interface, перед которым может стоять модификатор public, означающий, как и для класса, что интерфейс доступен всюду. Если же модификатора public нет, интерфейс будет виден только в своем пакете.

После слова Interface записывается имя интерфейса, потом может стоять слово extends и список интерфейсов-предков через запятую. Таким образом, интерфейсы могут порождаться от интерфейсов, образуя свою, независимую от классов, иерархию, причем в ней допускается множественное наследование интерфейсов. В этой иерархии нет корня, общего предка.

Затем, в фигурных скобках, записываются в любом порядке константы и заголовки методов. Можно сказать, что в интерфейсе все методы абстрактные, но слово abstract писать не надо. Константы всегда статические, но слова static и final указывать не нужно.

Все константы и методы в интерфейсах всегда открыты, не надо даже указывать модификатор public.

Таким образом, интерфейс — это только набросок, эскиз. В нем указано, что делать, но не указано, как это делать.

Как же использовать интерфейс, если он полностью абстрактен, в нем нет ни одного полного метода?

Использовать нужно не интерфейс, а его реализацию (implementation). Реализация интерфейса — это класс, в котором расписываются методы одного или

нескольких интерфейсов. В заголовке класса после его имени или после имени его суперкласса, если он есть, записывается слово implements и, через запятую, перечисляются имена интерфейсов.

Вот как можно реализовать иерархию автомобилей:

- Interface Automobile{... }
- Interface Car extends Automobile{... }
- Class Truck implements Automobile!... }
- Class Pickup extends Truck implements Car{... }

или так:

- Interface Automobile{... }
- Iinterface Car extends Automobile{... }
- Interface Truck extends Automobile{... }
- Class Pickup implements Car, Truck{... }

Реализация интерфейса может быть неполной, некоторые методы интерфейса расписаны, а другие нет. Такая реализация - абстрактный класс, его обязательно надо пометить модификатором abstract.

Как реализовать в классе pickup метод f(), описанный и в интерфейсе car, и в интерфейсе Truck с одинаковой сигнатурой? Ответ простой - никак. Такую ситуацию нельзя реализовать в классе Pickup. Программу надо спроектировать по-другому.

Итак, интерфейсы позволяют реалиовать средствами Java чистое объектно-ориентированное проектирование, не отвлекаясь на вопросы реализации проекта.

Мы можем, приступая к разработке проекта, записать его в виде иерархии интерфейсов, не думая о реализации, а затем построить по этому проекту иерархию классов, учитывая ограничения одиночного наследования и видимости членов классов.

Листинг показывает, как можно собрать с помощью интерфейса хор домашних животных.

Листинг. Использование интерфейса для организации полиморфизма

```
interface Голос {
    void голос();
}
Class Собака implements Голос {
    public void voice () {
```

```
            System.out.println("Гав-Гав");
        }
}
Class Кошка implements Голос {
        public void voice () {
            System.out.println("Мяу");
        }
}
```

Что же лучше использовать: абстрактный класс или интерфейс? На этот вопрос нет однозначного ответа.

Создавая абстрактный класс, вы волей-неволей погружаете его в иерархию классов, связанную условиями одиночного наследования и единым предком — классом Object. Пользуясь интерфейсами, вы можете свободно проектировать систему, не задумываясь об этих ограничениях.

С другой стороны, в абстрактных классах можно сразу реализовать часть методов. Реализуя же интерфейсы, вы обречены на скучное переопределение всех методов.

Вы, наверное, заметили и еще одно ограничение: все реализации методов интерфейсов должны быть открытыми, public, поскольку при переопределении можно лишь расширять доступ, а методы интерфейсов всегда открыты.

Вообще же наличие и классов, и интерфейсов дает разработчику богатые возможности проектирования. В нашем примере, вы можете включить в хор любой класс, просто реализовав в нем интерфейс voice.

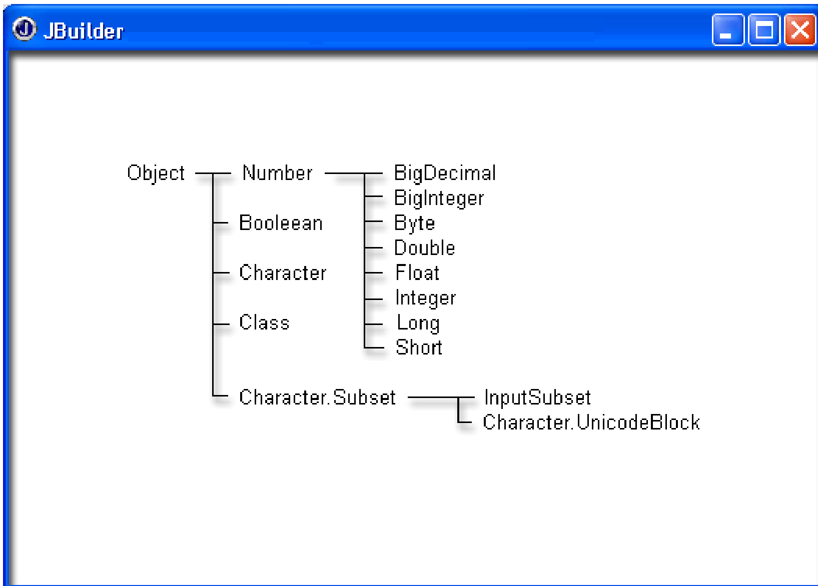
14. Классы оболочки

Java полностью ООП язык. Это означает, что все, что только можно, в Java представлено объектами.

8 примитивных типов нарушают это правило. Они оставлены в Java из-за многолетней привычки к числам и символам. Да и арифметические действия удобнее и быстрее производить с обычными числами, а не с объектами классов.

Но и для этих типов в языке Java есть соответствующие классы классы-оболочки (wrapper) примитивных типов. Конечно, они предназначены не для вычислений, а для действий, типичных при работе с классами - создания объектов, преобразования объектов, получения численных значений объектов в разных формах и передачи объектов в методы по ссылке.

На рисунке показана одна из ветвей иерархии классов Java. Для каждого примитивного типа есть соответствующий класс. Числовые классы имеют общего предка - абстрактный класс Number, в котором описаны 6 методов, возвращающих числовое значение, содержащееся в классе, приведенное к соответствующему примитивному типу: byteValue(), doubleValue(), floatValue(), intValue(), longValue (), shortValue (). Эти методы переопределены в каждом из 6 числовых классов-оболочек.



Помимо метода сравнения объектов equals(), переопределенного из класса Object, все описанные в этой главе классы, кроме Boolean и Class, имеют метод compareTo(), сравнивающий числовое значение, содержащееся в данном объекте, с числовым значением объекта - аргумента метода compareTo(). В результате работы метода получается целое значение:

- нуль, если значения равны;
- отрицательное число (-1), если числовое значение в данном объекте меньше, чем в объекте-аргументе;
- положительное число (+1), если числовое значение в данном объекте больше числового значения, содержащегося в аргументе.

14.1. Числовые классы

В каждом из 6 числовых классов-оболочек есть статические методы преобразования строки символов типа string, представляющей число, в соответствующий примитивный тип:

- Byte.parseByte(),
- Double.parseDouble(),
- Float.parseFloat(),
- Integer.parseInt(),
- Long.parseLong(),
- Short.parseShort().

Исходная строка типа String, как всегда в статических методах, задается как аргумент метода. Эти методы полезны при вводе данных в поля ввода, обработке параметров командной строки, т.е. всюду, где числа представляются строками цифр со знаками плюс или минус и десятичной точкой.

В каждом из этих классов есть статические константы MAX_VALUE и MIN_VALUE, показывающие диапазон числовых значений соответствующих примитивных типов. В классах Double и Float есть еще константы POSITIVE_INFINITY (+бесконечность), NEGATIVE_INFINITY (-бесконечность), NaN (не число), и логические методы проверки isNan(), isInfinite().

Если вы хорошо знаете двоичное представление вещественных чисел, то можете воспользоваться статическими методами floatToIntBits() и doubieToLongBits(), преобразующими вещественное число в целое. Вещественное число задается как аргумент метода. Затем вы можете изменить отдельные биты побитными операциями и преобразовать измененное целое число обратно в вещественное значение методами intBitsToFioat() и longBitsToDouble().

Статическими методами toBinaryString(), toHexString() и toOctalString() классов Integer и Long можно преобразовать целые значения типов int и long, заданные как аргумент метода, в строку символов, показывающую двоичное, шестнадцатеричное или восьмеричное представление числа.

14.2. Класс Boolean

Это очень небольшой класс, предназначенный главным образом для того, чтобы передавать логические значения в методы по ссылке.

Конструктор Boolean(String s) создает объект, содержащий значение true, если строка s равна "true" в любом сочетании регистров букв, и значение false — для любой другой строки.

Логический метод booieanValue() возвращает логическое значение, хранящееся в объекте.

14.3. Класс Character

В этом классе собраны статические константы и методы для работы с отдельными символами.

- Метод digit(char ch, in radix) переводит цифру ch системы счисления с основанием radix в ее числовое значение типа int.
- Метод forDigit(int digit, int radix) производит обратное преобразование целого числа digit в соответствующую цифру (тип char) в системе счисления с основанием radix. Основание системы счисления должно находиться в диапазоне от Character.MIN_RADIX до Character.MAX_RADIX.
- Метод toString() переводит символ, содержащийся в классе, в строку с тем же символом.
- Методы toLowerCase(), toUpperCase(), toTitlCase() возвращают символ, содержащийся в классе, в указанном регистре. Последний из этих методов предназначен для правильного перевода в верхний регистр четырех кодов Unicode, не выражающихся одним символом.

Множество статических логических методов проверяют различные характеристики символа, переданного в качестве аргумента метода:

- isDefined() - выясняет, определен ли символ в кодировке Unicode;
- isDigit() — проверяет, является ли символ цифрой Unicode;
- isIdentifierIgnorable() - выясняет, нельзя ли использовать символ в идентификаторах;
- isIsoControl() - определяет, является ли символ управляющим;
- isJavaIdentifierPart() - выясняет, можно ли использовать символ в идентификаторах;

- isJavaIdentifierStart() - определяет, может ли символ начинать идентификатор;
- isLetter() - проверяет, является ли символ буквой Java;
- IsLetterOrDigit() - проверяет, является ли символ буквой или цифрой Unicode;
- isLowerCase() - определяет, записан ли символ в нижнем регистре;
- isSpaceChar() -выясняет, является ли символ пробелом в смысле Unicode;
- isTitleCase() - проверяет, является ли символ титульным;
- isUnicodeIdentifierPart() - выясняет, можно ли использовать символ в именах Unicode;
- isUnicodeIdentifierStart() - проверяет, является ли символ буквой Unicode;
- isUpperCase() - проверяет, записан ли символ в верхнем регистре;
- isWhiteSpace() - выясняет, является ли символ пробельным.

Точные диапазоны управляющих символов, понятия верхнего и нижнего регистра, титульного символа и пробельных символов лучше всего посмотреть по документации Java API.

14.4. Класс BigInteger

Все примитивные целые типы имеют ограниченный диапазон значений. В целочисленной арифметике Java нет переполнения, целые числа приводятся по модулю, равному диапазону значений.

Для того чтобы можно было производить целочисленные вычисления с любой разрядностью, в состав Java API введен класс BigInteger, хранящийся в пакете java.math. Этот класс расширяет класс Number, следовательно, в нем переопределены методы doubleValue(), floatValue(), intValue(), longValue(). Методы byteVaiue() и shortvalue() не переопределены, а прямо наследуются от класса Number.

Действия с объектами класса BigInteger не приводят ни к переполнению, ни к приведению по модулю. Если результат операции велик, то число разрядов просто увеличивается. Числа хранятся в двоичной форме в дополнительном коде.

Перед выполнением операции числа выравниваются по длине распространением знакового разряда.

6 конструкторов класса создают объект класса BigDecimai из строки символов (знака числа и цифр) или из массива байтов.

2 константы — ZERO и ONE — моделируют нуль и единицу в операциях с объектами класса BigInteger.

Метод toByteArray() преобразует объект в массив байтов.

Большинство методов класса BigInteger моделируют целочисленные операции и функции, возвращая объект класса BigInteger:

- abs() - возвращает объект, содержащий абсолютное значение числа, хранящегося в данном объекте this ;
- add(x) - операция this + x ;
- and(x) - операция this & x ;
- andNot(x) - операция this & (~x) ;
- divide (x) - операция this / x ;
- divideAndRemainder(x) — возвращает массив из двух объектов класса BigInteger, содержащих частное и остаток от деления this на x ;
- gcd(x) - наибольший общий делитель абсолютных значений объекта this и аргумента x ;
- max(x) - наибольшее из значений объекта this и аргумента x ;
- min(x) - наименьшее из значений объекта this и аргумента x ;
- mod(x) - остаток от деления объекта this на аргумент метода x ;
- modInverse(x)- остаток от деления числа, обратного объекту this, на аргумент x ;
- modPow(n, m) - остаток от деления объекта this, возведенного в степень n, на m ;
- multiply(x) - операция this * x ;
- negate() - перемена знака числа, хранящегося в объекте;
- not() - операция ~this ;
- or(x) - операция this | x ;
- pow(n) - возведение числа, хранящегося в объекте, в степень n ;
- remainder(x) - операция this % x ;
- shiftLeft (n) - операция this « n ;
- shiftRight (n) - операция this » n;
- signum() - функция sign (x) ;
- subtract (x) - операция this - x ;
- xor(x) - операция this ^ x.

14.5. Класс BigDecimal

Класс BigDecimal расположен в пакете java.math.

Каждый объект этого класса хранит 2 целочисленных значения: мантиссу вещественного числа в виде объекта класса BigInteger, и неотрицательный десятичный порядок числа типа int.

Мантисса может содержать любое количество цифр, а порядок ограничен значением константы integer.MAX_VALUE. Результат операции над объектами класса BigDecimal округляется по одному из 8 правил, определяемых следующими статическими целыми константами:

- ROUND_CEILING - округление в сторону большего целого;
- ROUND_DOWN - округление к нулю, к меньшему по модулю целому значению;
- ROUND_FLOOR - округление к меньшему целому;
- ROUND_HALF_DOWN - округление к ближайшему целому, среднее значение округляется к меньшему целому;
- ROUND_HALF_EVEN - округление к ближайшему целому, среднее значение округляется к четному числу;
- ROOND_HALF_UP - округление к ближайшему целому, среднее значение округляется к большему целому;
- ROUND_UNNECESSARY - предполагается, что результат будет целым, и округление не понадобится;
- ROUND_UP - округление от нуля, к большему по модулю целому значению.

В классе BigDecimal 4 конструктора:

- BigDecimal(BigInteger bi) - объект будет хранить большое целое bi, порядок равен нулю;
- BigDecimal(BigInteger mantissa, int scale) - задается мантиса mantissa и неотрицательный порядок scale объекта; если порядок scale отрицателен, возникает исключительная ситуация;
- BigDecimal(double d) - объект будет содержать вещественное число удвоенной точности d ; если значение d бесконечно или NaN, то возникает исключительная ситуация;
- BigDecimal(String val) - число задается строкой символов val, которая должна содержать запись числа по правилам языка Java.

В классе переопределены методы doubleValue(), floatValue(), intValue(), longValue().

Большинство методов этого класса моделируют операции с вещественными числами. Они возвращают объект класса BigDecimal. В операциях буква x обозначает объект класса BigDecimal, буква n — целое значение типа int, буква r - способ округления, одну из восьми перечисленных выше констант:

- abs() - абсолютное значение объекта this ;
- add(x) - операция this + x ;

- divide(x, r) - операция this / x с округлением по способу r ;
- divide(x, n, r) - операция this / x с изменением порядка и округлением по способу r ;
- max(x) - наибольшее из this и x ;
- min(x) - наименьшее из this и x ;
- movePointLeft(n) - сдвиг влево на n разрядов;
- movePointRight(n) - сдвиг вправо на n разрядов;
- multiply(x) - операция this * x ;
- negate() - возврзщает объект с обратным знаком;
- scale() - возвращает порядок числз;
- setscaie(n) - устзнавливает новый порядок n ;
- setscaie(n, r) - устанавливает новый порядок п и округляет число при необходимости по способу r ;
- signum() - знак числа, хранящегося в объекте;
- subtract(x) - операция this - x ;
- toBiginteger() - округление числа, хранящегося в объекте;
- unscaiedvalue() -возвращает мантиссу числа.

14.6. Класс Class

Класс Object, стоящий во главе иерархии классов Java, представляет все объекты, действующие в системе, является их общей оболочкой. Всякий объект можно считать экземпляром класса Object.

Класс с именем Class представляет характеристики класса, экземпляром которого является объект. Он хранит информацию о том, не является ли объект на самом деле интерфейсом, массивом или примитивным типом, каков суперкласс объекта, каково имя класса, какие в нем конструкторы, поля, методы и вложенные классы.

В классе Class нет конструкторов, экземпляр этого класса создается исполняющей системой Java во время загрузки класса и предоставляется методом getCiass() класса Object, например:

```
String s = "Это строка";
Class c = s.getClass();
```

Статический метод forName(string Class) возвращает объект класса Class для класса, указанного в аргументе, например:

```
Class cl = Class.forName("Java,lang.String");
```

Но этот способ создания объекта класса Class считается устаревшим (deprecated). В новых версиях JDK для этой цели используется специальная конструкция — к имени класса через точку добавляется слово Class:

Class c2 = Java.lang.String.Class;

Логические методы isArray(), isInterface(), isPrimitive() позволяют уточнить, не является ли объект массивом, интерфейсом или примитивным типом.

Если объект ссылочного типа, то можно извлечь сведения о вложенных классах, конструкторах, методах и полях методами getDeciaredCiasses(), getDeclaredConstructors(), getDeclaredMethods(), getDeclaredFields(), в виде массива классов, соответственно, Class, Constructor, Method, Field. Последние 3 класса расположены в пакете java.lang.reflect и содержат сведения о конструкторах, полях и методах аналогично тому, как класс Class хранит сведения о классах.

Методы getClasses(), getConstructors(), getInterfaces(), getMethods(), getFieids() возвращают такие же массивы, но не всех, а только открытых членов класса.

Метод getSuperciass() возвращает суперкласс объекта ссылочного типа, getPackage() - пакет, getModifiers() - модификаторы класса в битовой форме. Модификаторы можно затем расшифровать методами класса Modifier из пакета Java.lang.reflect.

15. Классы коллекции

До сих пор мы пользовались массивами. Они удобны, если необходимо быстро обработать однотипные элементы, например, просуммировать числа, найти наибольшее и наименьшее значение, отсортировать элементы. Но уже для поиска нужных сведений в большом объеме информации массивы неудобны. Для этого лучше использовать бинарные деревья поиска.

Кроме того, массивы всегда имеют постоянную, предварительно заданную, длину, в массивы неудобно добавлять элементы. При удалении элемента из массива оставшиеся элементы следует перенумеровывать.

При решении задач, в которых количество элементов заранее неизвестно, элементы надо часто удалять и добавлять, надо искать другие способы хранения.

15.1. Класс Vector

В языке Java с самых первых версий есть класс Vector, предназначенный для хранения переменного числа элементов самого общего типа Object.

Класс Vector является примером того, как можно объекты класса Object, а значит, любые объекты, объединить в коллекцию. Этот тип коллекции упорядочивает и даже нумерует элементы. В векторе есть первый элемент, есть последний элемент. К каждому элементу обращаются непосредственно по индексу. При добавлении и удалении элементов оставшиеся элементы автоматически перенумеровываются.

В классе Vector из пакета java.util хранятся элементы типа Object, а значит, любого типа. Количество элементов может быть любым и наперед не определяться. Элементы получают индексы 0, 1, 2,.... К каждому элементу вектора можно обратиться по индексу, как и к элементу массива.

Кроме количества элементов, называемого размером (size) вектора, есть еще размер буфера - емкость (capacity) вектора. Обычно емкость совпадает с размером вектора, но можно ее увеличить методом ensureCapacity(int minCapacity) или сравнять с размером вектора методом trimToSize().

В Java 2 класс Vector переработан, чтобы включить его в иерархию классов-коллекций. Поэтому многие действия можно совершать старыми и новыми методами. Рекомендуется использовать новые методы, поскольку старые могут быть исключены из следующих версий Java.

Создание вектора.

В классе 4 конструктора:

- vector() - создает пустой объект нулевой длины.
- vector(int capacity) - создает пустой объект указанной емкости capacity.
- vector(int capacity, int increment) - создает пустой объект указанной емкости capacity и задает число increment, на которое увеличивается емкость при необходимости.
- vector(Collection c) - вектор создается по указанной коллекции. Если capacity отрицательно, создается исключительная ситуация. После создания вектора его можно заполнять элементами.

Методы класса

- Метод add(Object element) позволяет добавить элемент в конец вектора, то же делает старый метод addElement(Object element).
- Методом add(int index, Object element) или старым методом insertElementAt (Object element, int index) можно вставить элемент в указанное место index. Элемент, находившийся на этом месте, и все последующие элементы сдвигаются, их индексы увеличиваются на единицу.
- Метод addAll(Collection coll) позволяет добавить в конец вектора все элементы коллекции coll.
- Методом addAll(int index, Collection coll) возможно вставить, начиная с позиции index все элементы коллекции coll.
- Метод set(int index, Object element) заменяет элемент, стоявший в векторе в позиции index, на элемент element (то же позволяет выполнить старый метод setElementAt(Object element, int index)).
- Метод ИмяВектора.size() возвращает количество элементов в векторе.
- Метод ИмяВектора.capacity() возвращает емкость вектора.
- Метод ИмяВектора.isEmpty() возвращает true, если в векторе нет ни одного элемента.
- Метод ИмяВектора.firstEiement() возвращает первый элемент (индекс 0).
- Метод ИмяВектора.lastEiement() возвращает последний элемент..
- Метод ИмяВектора.get(int index) возвращает элемент с заданным индексом.
- То же самое делает старый метод ИмяВектора.elementAt(int index).

Эти методы возвращают объект класса Object. Перед использованием его следует привести к нужному типу.

Получить все элементы вектора в виде массива типа Object[] можно методами:

- ИмяВектора.toArray().
- ИмяВектора.toAr ray(Object [] A). Этот метод заносит все элементы вектора в массив A, если в нем достаточно места.

Проверить наличие элемента в векторе.

- Логический метод ИмяВектора.contains(Object element) возвращает true, если элемент element находится в векторе.
- Логический метод ИмяВектора.containsAll(Collection C) возвращает true, если вектор содержит все элементы указанной коллекции.

4 метода позволяют отыскать позицию указанного элемента element:

- ИмяВектора.indexof(Object element) ведет поиск, начиная с индекса 0 включительно,.возвращает индекс первого появления элемента в векторе.
- ИмяВектора.indexOf(Object element, int begin) ведет поиск, начиная с индекса begin включительно, возвращает индекс первого появления элемента в векторе.
- ИмяВектора.lastindexOf(Object element) ведет поиск, начиная с индекса begin включительно, возвращает индекс последнего появления элемента в векторе;
- lastindexOf(Object element, int start) ведет поиск от индекса start включительно к началу вектора, возвращает индекс первого появления элемента в векторе.

Если элемент не найден, возвращается (-1).

Два метода в Java 2 для удаления 1 элемента:

- Логический метод ИмяВектора.remove(Object element) удаляет из вектора первое вхождение указанного элемента element. Метод возвращает true, если элемент найден и удаление произведено.
- Метод ИмяВектора.remove(int index) удаляет элемент из позиции index и возвращает его в качестве своего результата типа Object.

Аналогичные действия позволяют выполнить старые методы типа void:

- ИмяВектора.removeElement(Object element).
- ИмяВектора.removeElementAt(int index), не возвращающие результата.

Удалить диапазон элементов можно методом removeRange(int begin, int end), не возвращающим результата. Удаляются элементы от позиции begin включительно до позиции end исключительно.

Два метода в Java 2 для удаление группы элементов:

- Удалить из данного вектора все элементы коллекции coil возможно логическим методом ИмяВектора.removeAll(Collection coll).
- Удалить последние элементы можно, урезав вектор методом ИмяВектора.setSizefint(newSize) с заданием нового размера..

- Удалить все элементы, кроме входящих в указанную коллекцию coil, разрешает логический метод ИмяВектора.retainAll(Collection coll).
- Удалить все элементы вектора можно методом ИмяВектора.clear() или старым методом ИмяВектора.removeAHElements(), или обнулив размер-вектораметодом ИмяВектора.setSize(0).

15.2. Класс Stack

Класс stack из пакета java.util объединяет элементы в стек.

Стек (Stack) реализует порядок работы с элементами подобно магазину винтовки - первым выстрелит патрон, положенный в магазин последним Такой порядок обработки называется LIFO (Last In - First Out, первым пришел - последним ушел).

Перед работой создается пустой стек конструктором stack().

Затем на стек кладутся и снимаются элементы, причем доступен только "верхний" элемент, тот, что положен на стек последним.

Дополнительно к методам класса vector класс stack содержит 5 методов, позволяющих работать с коллекцией как со стеком:

- push(Object item) - помещает элемент item в стек;
- pop() - извлекает верхний элемент из стека;
- peek() - читает верхний элемент, не извлекая его из стека;
- empty() - проверяет, не пуст ли стек;
- search(Object item) - находит позицию элемента item в стеке. Верхний элемент имеет позицию 1 (не 0), под ним элемент 2 и т.д. Если элемент не найден, возвращается (-1).

15.3. Класс Hashtable

Класс Hashtable расширяет абстрактный класс Dictionary. В объектах этого класса хранятся пары "ключ + значение".

Из таких пар "Фамилия И.О. + номер" состоит, например, телефонный справочник. Класс упрощает поиск данных, так как ищется пара, а не каждый компонент отдельно.

Каждый объект класса Hashtable имеет параметры

- Размер (size) - количество пар.
- Емкость (capacity) - размер буфера.
- Показатель загруженности (load factor) - процент заполненности буфера, по достижении которого увеличивается его размер.

Создание хэш-таблицы.

Для создания объектов класс Hashtable предоставляет 4 конструктора:

- ИмяХэша.Hashtable() - создает пустой объект с начальной емкостью в 101 элемент и показателем загруженности 0,75.
- ИмяХэша.Hashtable(int capacity) - создает пустой объект с начальной емкостью capacity и показателем загруженности 0,75.
- ИмяХэша.Hashtable(int capacity, float loadFactor) - создает пустой Объект с начальной емкостью capacity и показателем загруженности loadFactor
- ИмяХэша.Hashtable(Map f) - создает объект класса Hashtable, содержащий все элементы отображения f, с емкостью, равной удвоенному числу элементов отображения f, но не менее 11, и показателем загруженности 0,75.

Заполнение таблицы.

- Метод ИмяХэша put(Object key, Object value) добавляет пару "key + value", если ключа key не было в таблице, и меняет значение value ключа key, если он уже есть в таблице. Возвращает старое значение ключа или pull, если его не было. Если хотя бы один параметр равен null, возникает исключительная ситуация;
- Метод void ИмяХэша.putAii(Map f) добавляет все элементы отображения f. В объектах-ключах key должны быть реализованы методы hashCode() и equals().

Получить значение по ключу.

- Метод ИмяХэша.get (Object key) возвращает значение элемента с ключом key в виде объекта класса Object. Для дальнейшей работы его следует преобразовать к конкретному типу.

Проверка наличия ключа или значения.

- Метод ИмяХэша.containsKey(Object key) возвращает true, если в таблице есть ключ key.
- Метод ИмяХэша.containsvalue(Object value).
- Старый метод ИмяХэша.contains(Object value) возвращают true, если в таблице есть ключи со значением value.
- Метод ИмяХэша.isEmpty() возвращает true, если в таблице нет элементов.

Получить все элементы таблицы.

- Метод ИмяХэша.values() представляет все значения value таблицы в виде интерфейса Collection. Все модификации в объекте collection изменяют таблицу, и наоборот.

- Метод ИмяХэша.keyset() предоставляет все ключи key таблицы в виде интерфейса set. Все изменения в объекте set корректируют таблицу, и наоборот.
- Метод ИмяХэша.entrySet() представляет все пары " key— value " таблицы в виде интерфейса Set. Все модификации в объекте set изменяют таблицу, и наоборот.
- Метод ИмяХэша.toString() возвращает строку, содержащую все пары.

Старые методы ИмяХэша.elements() и ИмяХэша.keys() возвращают значения и ключи в виде интерфейса Enumeration (перечисление).

Удалить элементы.

- Метод ИмяХэша.remove(Object key) удаляет пару с ключом key, возвращая значение этого ключа, если оно есть, и null, если пара с ключом key не найдена.
- Метод ИмяХэша.clear() удаляет все элементы, очищая таблицу.

Листинг. Телефонный справочник

```
import java.util.*;
Class PhoneBook {
    public static void main(String[] args) {   // arg[0] из командной строки
        Hashtabie Хэш = new Hashtabie(); // Создана хэш-табдица
        String Имя = null;                // Ссылка на Имя
        Хэш.put("John", "123-45-67");
        Хэш.put ("Lemon", "567-34-12");
        Хэш.put("Bill", "342-65-87");
        Хэш.put("Gates", "423-83-49");
        Хэш.put("Batman", "532-25-08");
        try      {
            Имя = args[0];  // Занести в Имя введенное в командной строке
        }
        catch(Exception e) {     // Перехват исключения
            System.out.println("Используйте имя из справочника");
            return;
        }
        if (Хэш.containsKey(Имя))
            System.out.println(Имя + "'Имя, телефон = " + Хэш.get(Имя));
        else
            System.out.println("Сожалею, нет такого имени");
    )
}
```

15.4. Класс Properties

Класс Properties расширяет класс Hashtabie. Он предназначен в основном для ввода и вывода пар свойств системы и их значений. Пары хранятся в виде строк типа String. В классе Properties два конструктора:

- Properties() - создает пустой объект;
- Properties(Properties default) - создает объект с заданными парами свойств default.

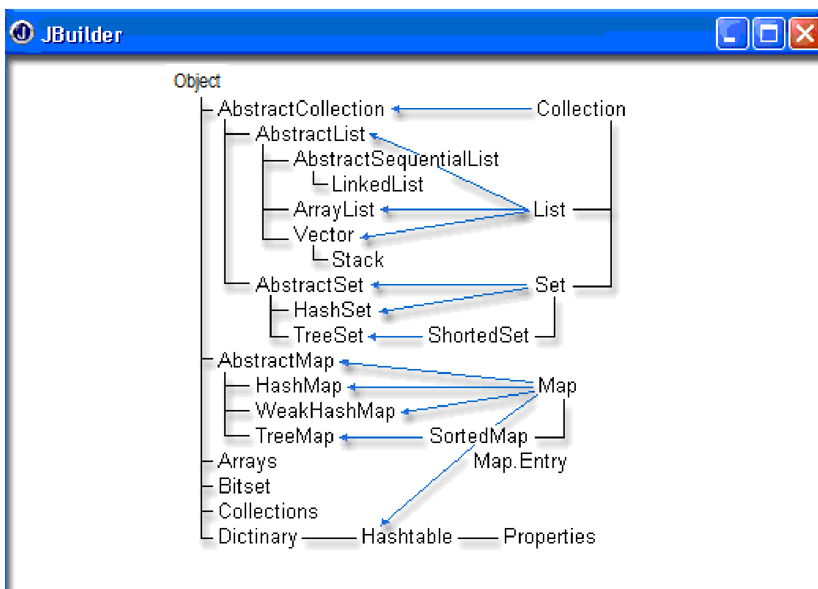
Кроме унаследованных от класса Hashtabie методов в классе Properties есть еще следующие методы:

- Метод String getProperty(string key) - возвращает значение по ключу key ;
- Метод String getProperty(String.key, String defaultValue) - возвращает значение по ключу key ; если такого ключа нет, возвращается defaultValue
- Метод setProperty(String key, String value) - добавляет новую пару, если ключа key нет, и меняет значение, если ключ key есть.
- Метод load(Inputstream in) - загружает свойства из входного потока in.
- Методы list(PrintStream out) и list(PrintWriter out) - выводят свойства в выходной поток out.
- Метод store (OutputStream out, String header) - выводит свойства в выходной поток out с заголовком header.

Листинг. Вывод системных свойств

```
Class Prop {
    public static void main(String[] args) {
        System.getProperties().list(System.out);
    }
}
```

Примеры классов Vector, Stack, Hashtabie, Properties показывают удобство классов-коллекций. Поэтому в Java 2 разработана целая иерархия коллекций. Она показана ниже. Линии указывают классы, реализующие интерфейсы. Все коллекции разбиты на 3 группы, описанные в интерфейсах List, Set и Map.



15.5. Интерфейс Collection

Интерфейс Ccollection из пакета java.util описывает общие свойства коллекций List (Список) и Set (Множество). Он содержит методы добавления и удаления элементов, проверки и преобразования элементов:

- Метод Boolean.add(Object obj) обавляет элемент obj в конец коллекции; возвращает false, если такой элемент в коллекции уже есть, а коллекция не допускает повторяющиеся элементы; возвращает true, если добавление прошло удачно.
- Метод Boolean.addAll(Collection coll) добавляет все элементы коллекции coll в конец данной коллекции.
- Метод void clear() удаляет все элементы коллекции.
- Метод Boolean.contains(Object obj) проверяет наличие элемента obj в коллекции.
- Метод Boolean.containsAll(Collection coll) проверяет наличие всех элементов коллекции coll в данной коллекции.
- Метод Boolean.isEmpty() проверяет, пуста ли коллекция.
- Метод Iterator.iterator() — возвращает итератор данной коллекции.
- Метод Boolean remove(Object obj) удаляет указанный элемент из коллекции; возвращает false, если элемент не найден, или true, если удаление прошло успешно.

- Метод Boolean.removeAll(Collection coil) удаляет элементы указанной коллекции, лежащие в данной коллекции;
- Метод Boolean.retainAll(Collection coll) удаляет все элементы данной коллекции, кроме элементов коллекции coll ;
- Метод int.size() возвращает количество элементов в коллекции;
- Метод Object [].toArray() возвращает все элементы коллекции в виде массива;
- Метод Object.toArray(Object[] A) записывает все элементы коллекции в массив A, если в нем достаточно места.

15.6. Интерфейс List

Интерфейс List из пакета java.util, расширяющий интерфейс Collection, описывает методы работы с упорядоченными коллекциями. Иногда их называют последовательностями (sequence). Элементы такой коллекции пронумерованы, начиная от нуля, к ним можно обратиться по индексу. В отличие от коллекции Set элементы коллекции List могут повторяться.

Класс Vector одна из реализаций интерфейса List.

Интерфейс List добавляет к методам интерфейса Collection методы, использующие индекс (index) элемента:

- Метод void add(int index, Object obj) - вставляет элемент obj в позицию index, старые элементы, начиная с позиции index, сдвигаются, их индексы увеличиваются на единицу.
- Метод Boolean.addAll(int index, Collection coll) - вставляет все элементы коллекции coil.
- Метод Object.get(int index) - возвращает элемент, находящийся в позиции index.
- Метод int.indexOf(Object obj) - возвращает индекс первого появления элемента obj в коллекции.
- Метод int.lastindexOf(Object obj) - возвращает индекс последнего появления элемента obj в коллекции.
- Метод Listiterator.listiterator() - возвращает итератор коллекции.
- Метод Listiterator.listiterator(int index) - возвращает итератор конца коллекции от позиции index.
- Метод Object.set(int index, Object obj) - заменяет элемент, находящийся в позиции index, элементом obj.
- Метод List.subList(int from, int to) - возвращает часть коллекции от позиции from включительно до позиции to исключительно.

15.7. Интерфейс Set и SortedSet.

Интерфейс Set из пакета java.util, расширяющий интерфейс Collection, описывает неупорядоченную коллекцию, **не содержащую** повторяющихся элементов. Это соответствует математическому понятию множества (Set). Такие коллекции удобны для проверки наличия или отсутствия у элемента свойства, определяющего множество. Новые методы в интерфейс Set не добавлены, просто метод add() не станет добавлять еще одну копию элемента, если такой элемент уже есть в множестве. Этот интерфейс расширен интерфейсом sortedset.

Интерфейс SortedSet из пакета java.util, расширяющий интерфейс Set, описывает упорядоченное множество, отсортированное по естественному порядку возрастания его элементов или по порядку, заданному реализацией интерфейса comparator.

Элементы не нумеруются, но есть понятие первого, последнего, большего и меньшего элемента. Дополнительные методы интерфейса отражают эти понятия:

- Метод Comparator.comparator() - возвращает способ упорядочения коллекции;
- Метод Object first() - возвращает первый, меньший элемент коллекции.
- Метод SortedSet.headset(Object toEiement) - возвращает начальные, меньшие элементы до элемента toEiement исключительно.
- Метод Object.last() - возвращает последний, больший элемент коллекции.
- Метод SortedSet subset(Object fromElement, Object toEiement) - возвращает подмножество коллекции от элемента fromElement включительно до элемента toEiement исключительно.
- Метод SortedSet.tailSet(Object fromElement) - возвращает последние, большие элементы коллекции от элемента fromElement включительно.

15.8. Интерфейс Map

Интерфейс Map из пакета java.util описывает коллекцию, состоящую из пар "ключ + значение". У каждого ключа только одно значение, что соответствует математическому понятию однозначной функции или **отображения** (Map).

Такую коллекцию часто называют еще словарем (dictionary) или ассоциативным массивом (associative array).

Обычный массив – это простейший пример словаря с заранее заданным числом элементов. Это отображение множества первых неотрицательных целых чисел на множество элементов массива, множество пар "индекс массива + ссылка на элемент массива".

Класс HashTable одна из реализаций интерфейса мар.

Интерфейс Map содержит методы, работающие с ключами и значениями:

- Метод Boolean.containsKey(Object key) - проверяет наличие ключа key.
- Метод Boolean.containsValue(Object value) - проверяет наличие значения value.
- Метод Set.entryset() - представляет коллекцию в виде множества, каждый элемент которого пара из данного отображения, с которой можно работать методами вложенного интерфейса Map.Entry.
- Метод Object.get(Object key) - возвращает значение, отвечающее ключу key.
- Метод Set.keyset() - представляет ключи коллекции в виде множества.
- Метод Object.put(Object key, Object value) - добавляет пару "key + value", если такой пары не было, и заменяет значение ключа key, если такой ключ уже есть в коллекции.
- Метод void putAll(Map M) - добавляет к коллекции все пары из отображения M.
- Метод Collection.values() - представляет все значения в виде коллекции.

В интерфейс Map вложен интерфейс Map.Entry, содержащий методы работы с отдельной парой.

Вложенный интерфейс Map.Entry

Этот интерфейс описывает методы работы с парами, полученными методом entrySet():

- Методы g etKey() и getvaiue() - позволяют получить ключ и значение пары;
- Метод setvaiue (Object value) - меняет значение в данной паре.

Интерфейс SortedMap.

Интерфейс SortedMap, расширяющий интерфейс Map, описывает упорядоченную по ключам коллекцию Map. Сортировка производится либо в естественном порядке возрастания ключей, либо в порядке, описываемом в интерфейсе Comparator.

Элементы не нумеруются, но есть понятия большего и меньшего из двух элементов, первого, самого маленького, и последнего, самого большого элемента коллекции. Эти понятия описываются следующими методами:

- Метод comparator.comprator() - возвращает способ упорядочения коллекции.
- Метод Object.firstKey() - возвращает первый, меньший элемент коллекции.

- Метод SortedMap.headMap(Object toKey) - возвращает начало коллекции до элемента с ключом toKey исключительно.
- Метод Object.lastKey() - возвращает последний, больший ключ коллекции.
- Метод SortedMap.subMap(Object fromKey, Object toKey) - возвращает часть коллекции от элемента с ключом fromKey включительно до элемента с ключом toKey исключительно.
- Метод SortedMap.taiiMap(Object fromKey) - возвращает остаток коллекции от элемента fromKey включительно.

Вы можете создать свои коллекции, реализовав рассмотренные интерфейсы. Это дело трудное, поскольку в интерфейсах много методов. Чтобы облегчить эту задачу, в Java API введены частичные реализации интерфейсов — абстрактные классы-коллекции.

15.9. Абстрактные классы-коллекции

Эти классы лежат в пакете java.util:

- Абстрактный класс AbstractGollection реализует интерфейс Collection, но оставляет нереализованными методы iterator(), size().
- Абстрактный класс AbstractList реализует интерфейс List, но оставляет нереализованным метод get(mt) и унаследованный метод size(). Этот класс позволяет реализовать коллекцию с прямым доступом к элементам, подобно массиву
- Абстрактный класс AbstractSequantaaiList реализует интерфейс List, но оставляет нереализованным метод listiterator(int index) и унаследованный метод size(). Данный класс позволяет реализовать коллекции с последовательным доступом к элементам с помощью итератора ListIerator.
- Абстрактный класс AbstractSet реализует интерфейс Set, но оставляет нереализованными методы, унаследованные от AbstractCollection.
- Абстрактный класс AbstractMap реализует интерфейс Map, но оставляет нереализованным метод entrySet().

Наконец, в составе Java API есть полностью реализованные классы-коллекции помимо уже рассмотренных классов Vectdr, Stack, Hashtable и Properties, Это классы ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap, WeakHashMap.

Для работы с этими классами разработаны интерфейсы:

Iterator, Listiterator, Comparator, а также классы Arrays, Collections.

15.10. Интерфейс Iterator

В 70—80-х годах прошлого столетия, после того как была осознана важность правильной организации данных в определенную структуру, большое внимание уделялось изучению и построению различных структур данных: связанных списков, очередей, деков, стеков, деревьев, сетей

Вместе с развитием структур данных развивались и алгоритмы работы с ними: сортировка, поиск, обход, хэширование..

В 90-х годах было решено заносить данные в определенную коллекцию, скрыв ее внутреннюю структуру, а для работы с данными использовать методы этой коллекции.

В частности, задачу обхода возложили на саму коллекцию. В Java API введен интерфейс Iterator, описывающий способ обхода всех элементов коллекции. В каждой коллекции есть метод iterator(), возвращающий реализацию интерфейса Iterator для указанной коллекции. Получив эту реализацию, можно обходить коллекцию в некотором порядке, определенном данным итератором, с помощью методов, описанных в интерфейсе Iterator и реализованных в этом итераторе. Подобная техника использована в классе StringTokenizer.

15.11. Интерфейс ListIterator

Интерфейс ListIterator расширяет интерфейс Iterator, обеспечивая перемещение по коллекции как в прямом, так и в обратном направлении. Он может быть реализован только в тех коллекциях, в которых есть понятия следующего и предыдущего элемента и где элементы пронумерованы.

В интерфейс ListIterator добавлены следующие методы:

- Метод void add(Object element) - добавляет элемент element перед текущим элементом.
- Метод Boolean.hasPrevious() - возвращает true, если в коллекции есть элементы, стоящие перед текущим элементом.
- Метод int.nextIndex() - возвращает индекс текущего элемента; если текущим является последний элемент коллекции, возвращает размер коллекции.
- Метод Object.previous() - возвращает предыдущий элемент и делает его текущим;
- Метод int.previousIndex() - возвращает индекс предыдущего элемента;
- Метод void set(Object element) заменяет текущий элемент элементом element, выполняется сразу после next() или previous().

Как видите, итераторы могут изменять коллекцию, в которой они работают, добавляя, удаляя и заменяя элементы. Чтобы это не приводило к конфликтам, предусмотрена исключительная ситуация, возникающая при попытке использования итераторов параллельно "родным" методам коллекции.

15.12. Классы, создающие списки

Класс ArrayList полностью реализует интерфейс List и итератор типа Iterator. Класс ArrayList очень похож на класс Vector, имеет тот же набор методов и может использоваться в тех же ситуациях.

В классе ArrayList 3 конструктора:

- ArrayList() - создает пустой объект;
- ArrayList(Collection col) - создает объект, содержащий все элементы коллекции col ;
- ArrayList(int initCapacity) - создает пустой Объект емкости initCapacity.

Единственное отличие класса ArrayList от класса Vector заключается в том, что класс ArrayList не синхронизован. Это означает что одновременное изменение экземпляра этого класса несколькими подпроцессами приведет к непредсказуемым результатам.

LinkedList – это двунаправленный список

Класс LinkedList полностью реализует интерфейс List и содержит дополнительные методы, превращающие его в двунаправленный список. Он реализует итераторы типа Iterator и bistIterator.

Этот класс можно использовать для обработки элементов в стеке, деке или двунаправленном списке.

В классе LinkedList 2 конструктора:.

- LinkedList() - создает пустой объект
- LinkedList(Collection col) - создает объект, содержащий все элементы коллекции col.

15.13. Классы отображений

15.13.1. Обычные отображения

Класс HashMap очень похож на класс Hashtabie и может использоваться в тех же ситуациях. Класс, например, полностью реализует интерфейс Map, а также итератор типа Iterator.. Он имеет тот же набор функций и такие же конструкторы:

- HashMap() - создает пустой объект с показателем загруженности 0,75;
- HashMap(int.capacity) - создает пустой объект с начальной емкостью capacity и показателем загруженности 0,75;
- HashMap(int capacity, float loadFactor) - создает пустой объект С начальной емкостью capacity и показателем загруженности loadFactor ;
- HashMap(Map f) - создает объект класса HashMap, содержащий все элементы отображения f, с емкостью, равной удвоенному числу элементов отображения f, но не менее 11, и показателем загруженности 0,75.

Класс WeakHashMap отличается от класса HashMap только тем, что в его объектах неиспользуемые элементы, на которые никто не ссылается, автоматически исключаются из объекта.

15.13.2. Упорядоченные отображения

Класс TreeMap полностью реализует интерфейс sortedMap. Он реализован как бинарное дерево поиска, значит его элементы хранятся в упорядоченном виде. Это значительно ускоряет поиск нужного элемента.

Порядок задается либо естественным следованием элементов, либо объектом, реализующим интерфейс сравнения Comparator.

В этом классе 4 конструктора:

- TreeMap() - создает пустой объект с естественным порядком элементов;
- TreeMap(Comparator с) - создает пустой объект, в котором порядок задается объектом сравнения с ;
- TreeMap (Map f) - создает объект, содержащий все элементы отображения f с естественным порядком его элементов;
- TreeMap(SortedMap sf) - создает объект, содержащий все элементы отображения sf, в том же порядке.

Здесь надо пояснить, каким образом можно задать упорядоченность элементов коллекции

Интерфейс Comparator описывает 2 метода сравнения:

- int compare(Object obji, Object obj2) - возвращает отрицательное число, если objl в каком-то смысле меньше obj2 ; нуль, если они считаются равными; положительное число, если objl больше obj2.
- Boolean equals(Object obj) - сравнивает данный объект с объектом obj, возвращая true, если объекты совпадают в каком-либо смысле, заданном этим методом.

Для каждой коллекции можно реализовать эти два метода, задав конкретный способ сравнения элементов, и определить объект класса SortedMap вторым конструктором. Элементы коллекции будут автоматически отсортированы в заданном порядке.

Листинг показывает один из возможных способов упорядочения комплексных чисел. Здесь описывается класс ComplexCompare, реализующий интерфейс Comparator, В листинге он применяется для упорядоченного хранения множества комплексных чисел.

Листинг. Сравнение комплексных чисел.

```
import java.util.*;
Class ComplexCompare implements Comparator {
    public int compare(Object obj1, Object obj2) {
        Complex zl = (Complex) objl, z2 = (Complex) obj2;
        double re1 = zl.getRe(), im1 = zl.getlm();
        double re2 = z2.getRe(), im2 = z2.getlm();
        if (re1 != re2) return (int)(re1 - re2);
        else if (im1 != im2) return (int)(iml — im2);
        else return 0;
    }
    public Boolean equals(Object z) {
        return compare(this, z) = 0;
    }
}
```

15.14. Классы множеств

15.14.1. Простые множества

Класс HashSet полностью реализует интерфейс Set и итератор типа Iterator. Класс Hashset используется в тех случаях, когда надо хранить только одну копию каждого элемента.

В классе HashSet 4 конструктора:

- Hashset() - создает пустой объект с показателем загруженности 0,75;
- HashSet(int capacity) - создает пустой объект с начальной емкостью capacity и показателем загруженности 0,75;
- HashSet(int capacity, float loadFactor) - создает пустой объект с начальной емкостью capacity и показателем загруженности loadFactor ;
- HashSet(Collection col) - создает объект, содержащий все элементы коллекции col, с емкостью, равной удвоенному числу элементов коллекции coll, но не менее 11, и показателем загруженности 0,75.

15.14.2. Упорядоченные множества

Класс TreeSet полностью реализует интерфейс SortedSet и итератор типа Iterator. Класс TreeSet реализован как бинарное дерево поиска, значит, его элементы хранятся в упорядоченном виде. Это значительно ускоряет поиск нужного элемента.

Порядок задается либо естественным следованием элементов, либо объектом, реализующим интерфейс сравнения Comparator.

Этот класс удобен при поиске элемента во множестве, например, для проверки, обладает ли какой-либо элемент свойством, определяющим множество.

В классе TreeSet 4 конструктора:

- TreeSet() - создает пустой объект с естественным порядком элементов;
- TreeSet(Comparator c) - создает пустой объект, в котором порядок задается объектом сравнения c;
- TreeSet(Collection col) - создает объект, содержащий все элементы коллекции col, с естественным порядком ее элементов;
- TreeSet(SortedMap sf) - создает объект, содержащий все элементы отображения sf, в том же порядке.

15.15. Класс Collections

Коллекции предназначены для хранения элементов в удобном для дальнейшей обработки виде. Очень часто обработка заключается в сортировке элементов и поиске нужного элемента. Эти и другие методы обработки собраны в класс Collections.

Все методы класса Collections статические, ими можно пользоваться, не создавая экземпляры класса Collections. Как обычно в статических методах, коллекция, с которой работает метод, задается его аргументом.

Сортировка может быть сделана только в упорядочиваемой коллекции, реализующей интерфейс List. Для сортировки в классе Collections есть 2 метода:

- static void sort(List col) - сортирует в естественном порядке возрастания коллекцию col, реализующую интерфейс List;
- static void sort(List coll, Comparator c) - сортирует коллекцию col в порядке, заданном объектом c.

После сортировки можно осуществить бинарный поиск в коллекции методами:

- static int binarySearch(List coll, Object element) - отыскивает элемент element в отсортированной в естественном порядке возрастания коллекции

coll и возвращает индекс элемента или отрицательное число, если элемент не найден; отрицательное число показывает индекс, с которым элемент element был бы вставлен в коллекцию, с обратным знаком;

- static int binarySearch(List coll, Object element, Comparator c) — то же, но коллекция отсортирована в порядке, определенном объектом c.

4 метода находят наибольший и наименьший элементы в упорядочиваемой коллекции:

- static Object max(Collection col) - возвращает наибольший в естественном порядке элемент коллекции col;
- static Object max(Collection col, Comparator c) - то же впорядке, заданном объектом c ;
- static Object mm(Collection col) - возвращает наименьший в естественном порядке элемент коллекции col;
- static Object min(Collection col, Comparator c) - то же в порядке, заданном объектом c.

2 метода "перемешивают" элементы коллекции в случайном порядке:

- static void shuffle(List col) - случайные числа задаются по умолчанию;
- static void shuffle (List col, Random r) — случайные числа определяются объектом r.

Метод reverse(List coll) - меняет порядок расположения элементов на обратный.

Метод copy(List from, List to) - копирует коллекцию from в коллекцию to.

Метод fill(List coll, Object element) - заменяет все элементы существующей коллекции col элементом element.

15.16. Утилиты

В этой главе описаны средства, полезные для создания программ работы с массивами, датами, случайными числами.

15.17. Класс Arrays - массивы

В классе Arrays из пакета java.util собрано множество методов для работы с массивами. Их можно разделить на 4 группы:

- Сортировка.
- Поиск.
- Заполнение.
- Копирование.

15.17.1. Сортировка

18 статических методов сортируют массивы с разными типами числовых элементов в порядке возрастания чисел или просто объекты в их естественном порядке. 8 из них имеют простой вид

static void sort(type[] a),

где type может быть один из 7 примитивных типов (byte, short, int, long, char, float, double) или тип Object.

8 методов с теми же типами сортируют часть массива от индекса from включительно до индекса to исключительно:

static void sort(type[] a, int from, int to).

Оставшиеся 2 метода сортировки упорядочивают массив или его часть с элементами типа Object по правилу, заданному объектом с, реализующим интерфейс Comparator:

- static void sort(Object[] a, Comparator c).
- static void sort(Object[] a, int from, int to, Comparator c).

15.17.2. Поиск

После сортировки можно организовать бинарный поиск в массиве одним из 9 статических методов поиска. 8 методов имеют вид

static int binarySearch(type[] a, type element),

где type — один из тех же 8 типов.

9-ый метод отыскивает элемент element в массиве, отсортированном в порядке, заданном объектом с

static int binarySearch(Object[] a, Object element, Comparator c).

Методы поиска возвращают индекс найденного элемента массива. Если элемент не найден, то возвращается отрицательное число, означающее индекс, с которым элемент был бы вставлен в массив в заданном порядке, с обратным знаком.

15.17.3. Заполнение

18 статических методов заполняют массив или часть массива указанным значением value:

- static void fill(type[], type value),
- static void fill(type[], int from, int to, type value),

где type — один из 8 примитивных типов или тип Object.

9 статических логических методов сравнивают массивы:

- static Boolean equals(type[] al, type[] a2)

где type — один из 8 примитивных типов или тип Object.

Массивы считаются равными, и возвращается true, если они имеют одинаковую длину и равны элементы массивов с одинаковыми индексами.

<u>Листинг. Применение методов класса Arrays</u>

```
import java.utii.*;
Class ArraysTest {
    public static void main(String[] args) {
        int[] a = {34, -45, 12, 67, -24, 45, 36, -56};
        Arrays.sort(a) ;
        for (int i = 0; i < a.length; i++)
        System.out.print (a[i]. + " ");
        System.out.println();
        Arrays.fill(a, Arrays.binarySearch(a, 12), a.length, 0);
        for (int i = 6; i < a.length; i++) System.out.print(a[i] + " ");
        System.out.println();
    }
}
```

15.17.4. Копирование массивов

В классе System из пакета java.iang есть статический метод копирования массивов, который использует сама исполняющая система Java. Этот метод действует быстро и надежно, его удобно применять в программах. Синтаксис:

```
static void arrayCopy(Object src, int src_ind, Object dest, int dest_ind, int count)
```

Из массива, на который указывает ссылка src, копируется count элементов, начиная с элемента с индексом src_ind, в массив, на который указывает ссылка dest, начиная с его элемента с индексом dest_ind.

Все индексы должны быть заданы так, чтобы элементы лежали в массивах, типы массивов должны быть совместимы, а примитивные типы обязаны полностью совпадать. Ссылки на массивы не должны быть равны null.

Ссылки src и dest могут совпадать, при этом для копирования создается промежуточный буфер. Метод можно использовать, например, для сдвига элементов в массиве. После выполнения

```
int[] arr = {5, 6, 1, 8, 9, 1, 2, 3, 4, 5, -3, -7};
System.arrayCopy(arr, 2, arr, 1, arr.length — 2);
```

получим (5, 7, 8, 9, 1, 2, 3, 4, 5, -3, -7, -7}.

15.18. Класс Locale - локальные установки

Некоторые данные (даты, время) традиционно представляются в разных местностях по-разному. Например, дата в России выводится в формате число, месяц, год через точку: 27.06.01. В США принята запись месяц/число/год через наклонную черту: 06/27/01.

Совокупность таких форматов для данной местности, как говорят на жаргоне "локаль", хранится в объекте класса Locale из пакета java.util. Для создания такого объекта достаточно выбрать язык language и местность country. Иногда требуется третья характеристика — вариант variant, определяющая программный продукт, например, "WIN", "MAC", "POSIX".

По умолчанию местные установки определяются операционной системой и читаются из системных свойств. Посмотрите на строки:

```
user.language = ru;        // Язык — русский
user.region = RU;          // Местность — Россия
file.encoding = Cpl251;    // Байтовая кодировка — CP1251
```

Они определяют русскую локаль и локальную кодировку байтовых символов. Локаль, установленную по умолчанию на той машине, где выполняется программа, можно выяснить статическим методом Locale.getDefauit().

Чтобы работать с другой локалью, ее надо прежде всего создать. Для этого в классе Locale есть два конструктора:

- Locale(String language, String country).
- Locale(String language, String country. String variant).

Параметр language — это строка из двух строчных букв, определенная стандартом ISO639, например, "ru", "fr", "en". Параметр country — строка из двух прописных букв, определенная стандартом ISO3166, например, "RU", "US".

Локаль часто указывают одной строкой "ru_RU", "en_GB", "en_US" и т.д.

После создания локали можно сделать ее локалью по умолчанию статическим методом:

    Locale.setDefault(Locale newLocale);

Несколько статических методов класса Locale позволяют получить параметры локали по умолчанию, или локали, заданной параметром locale:

- string getCountry() - стандартный код страны из двух букв;
- string getDispiayCountry() - страна записывается словом, обычно выводящимся на экран;
- String getDisplayCountry (Locale locale) - то же для указанной локали.

Такие же методы есть для языка и варианта.

Можно просмотреть список всех локалей, определенных для данной JVM, и их параметров, выводимый в стандартном виде:

- Locale[] getAvailableLocales()
- String!] getISOCountries()
- String[] getISOLanguages()

Установленная локаль в дальнейшем используется при выводе данных в местном формате.

15.19. Класс Date - дата

Методы работы с датами и показаниями времени собраны в два класса: Calendar и Date из пакета java.utii.

Объект класса Date хранит число миллисекунд, прошедших с 1 января 1970 г. 00:00:00 по Гринвичу. Это "день рождения" UNIX, он называется "Epoch".

Класс Date удобно использовать для отсчета промежутков времени в миллисекундах.

Получить текущее число миллисекунд, прошедших с момента Epoch на той машине, где выполняется программа, можно статическим методом

System.currentTimeMillis()

В классе Date несколько конструкторов:

- Date() заносит в создаваемый объект текущее время машины, на которой выполняется программа, по системным часам.
- Date(long miiiisec) - использует указанное число миллисекунд.
- Date(int year, int month, int date, int hrs, int min, int sec) – дата-время с годом, месяцем, днем, часом, минутой, секундой.
- Date(int year, int month, int date, int hrs, int min) – дата-время с годом, месяцем, днем, часом, минутой (секунда = 0).
- Date(int year, int month, int date) – только дата с годом, месяцем, днем.
- Date(String S) – дата из строки S.
- Метод parse(String S) – Дата из строки S с учетом форматов разных стран.

Получить значение, хранящееся в объекте, можно используя:

- Метод int getYear() - год (если он после 1900).
- Метод int getMonth() - месяц.
- Метод int getDate()- число месяца.
- Метод int getDay() - день недели (0 – воскресенье … 6 – суббота).
- Метод int getHous() - час (0 –полночь … 23).
- Метод int getMinutes() - минуты (0…59).
- Метод int getSeconds() - секунды (0…61, есть дополнительные секунды для отображения лишних секунд из-изменения продолжительности астрономического года со временем).
- Метод int getTime() - время в формат UTC.
- Метод int getTimezoneOffset() - смещение часового пояса в минутах.

Изменить значение, хранящееся в объекте, можно используя:

- Метод int setYear() – год (если он после 1900).
- Метод int setMonth() – месяц.
- Метод int setDate() – число месяца.
- Метод int setDay() - день недели (0 – воскресенье … 6 – суббота).
- Метод int setHous() – час (0 –полночь … 23).
- Метод int setMinutes() – минуты (0…59).

- Метод int setSeconds() – секунды (0…61, есть дополнительные секунды для отображения лишних секунд из-изменения продолжительности астрономического года со временем).

3 логических метода сравнивают отсчеты времени:

- Метод boolean before(Date Другая) – возвращает true, если Date меньше Другая.
- Метод boolean after(Date Другая) – возвращает true, если Date больше Другая.
- Метод boolean equels(Date Другая) – возвращает true, если Date = Другая.

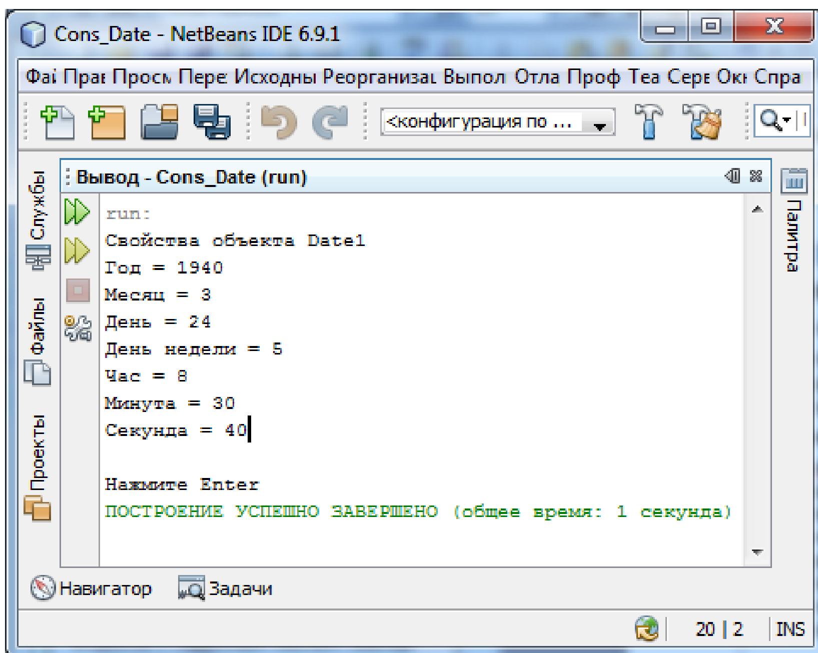
Преобразование миллисекунд, хранящихся в объектах класса Date, в текущее время и дату производится методами класса Calendar.

В примере создается объект Date - Date1, для которого затем в консоль выводятся детали.

<u>Листинг программы</u>

```
package cons_date;
import java.util.Date;
public class Main {
    public static void main(String[] args) {
        Date Date1 = new Date(1940,3,24,8,30,40);
        System.out.println("Свойства объекта Date1");
        System.out.println("Год = " + Date1.getYear());
        System.out.println("Месяц = "+Date1.getMonth());
        System.out.println("День = "+Date1.getDate());
        System.out.println("День недели = " + Date1.getDay());
        System.out.println("Час = " + Date1.getHours());
        System.out.println("Минута = " + Date1.getMinutes());
        System.out.println("Секунда = " + Date1.getSeconds());
        System.out.println();
        System.out.println("Нажмите Enter");
    }
}
```

При прогоне получаем:



15.20. Часовой пояс и летнее время

Методы установки и изменения часового пояса (time zone), а также летнего времени DST (Daylight Savings Time), собраны в абстрактном классе Timezone из пакета java.utii. В этом же пакете есть его реализация — подкласс SimpleTimeZone.

В классе simpieTimeZone 3 конструктора, но чаще всего объект создается статическим методом getDefault(), возвращающим часовой пояс, установленный на машине, выполняющей программу.

В этих классах множество методов работы с часовыми поясами, но в большинстве случаев требуется только узнать часовой пояс на машине, выполняющей программу, статическим методом getDefault(), проверить осуществляется ли переход на летнее время логическим методом useDaylightTime() и установить часовой пояс методом setDefault(TimeZone zone).

15.21. Класс Calendar

Класс Calendar — абстрактный, в нем собраны общие свойства календарей: юлианского, григорианского, лунного. В Java API пока есть только одна его реализация — подкласс GregorianCalendar.

Поскольку calendar — абстрактный класс, его экземпляры создаются 4 статическими методами по заданной локали и/или часовому поясу:

- Calendar getInstance()
- Calendar getInstance(Locale loc)
- Calendar getInstance(TimeZone tz)
- Calendar getInstance(TimeZone tz, Locale loc)

Определены целочисленные константы для работы с месяцами - от JANUARY до DECEMBER, для работы с днями недели - от MONDAY до SUNDAY.

Первый день недели можно узнать методом int getFirstDayOfweek(), а установить - методом setFirstDayOfWeek(int day), например:

```
setFirstDayOfWeek(Calendar.MONDAY)
```

Остальные методы позволяют просмотреть время и часовой пояс или установить их.

15.21.1. Подкласс GregorianCalendar

В григорианском календаре две целочисленные константы определяют эры: BC (before Christ) и AD (Anno Domini).

7 конструкторов определяют календарь по времени, часовому поясу и/или локали:

- GregorianCalendar()
- GregorianCalendar(int year, int month, int date)
- GregorianCalendar(int year, int month, int date, int hour, int minute)
- GregorianCalendar(int year, int month, int date,
- int hour, int minute, int second)
- GregorianCalendar(Locale loc)
- GregorianCalendar(TimeZone tz)
- GregorianCalendar(TimeZone tz, Locale loc)

После создания объекта следует определить дату перехода с юлианского календаря на григорианский календарь методом setGregorianChange(Date date). По умолчанию это 15 октября 1582г. На территории России переход был осуществлен 14 февраля 1918г., значит, создание объекта greg надо выполнить так:

```
GregorianCalendar greg = new GregorianCalendar();
greg.setGregorianChange(new
    GregorianCalendar(1918, Calendar.FEBRUARY, 14).getTime ()) ;
```

Узнать, является ли год високосным в григорианском календаре, можно логическим методом

i sLeapYear ().

Метод get(int field) возвращает элемент календаря, заданный аргументом field. Для этого аргумента в классе Calendar определены следующие статические целочисленные константы:

- ERA – эра.
- YEAR – год.
- DAY_OF_YEAR – день года.
- WEEK_OF_YEAR неделя года.
- MONTH – месяц.
- DAY_OF_MONTH – день месяца
- WEEK_OF_MONTH – неделя месяца.
- DAY_OF_WEEK – день недели.
- DAY_OF_WEEK_IN_MONTH – дней из недели в месяце.
- HOUR_OF_DAY – час дня.
- MINUTE – минута.
- SECOND – секунда.
- MILLISECOND – миллисекунда.
- ZONE_OFFSET – смещение часового пояса.
- DATE – дата.

Несколько методов set(), использующих эти константы, устанавливают соответствующие значения.

15.21.2. Представление даты и времени

Различные способы представления дат и показаний времени можно осуществить методами, собранными в абстрактный класс DateFormat и его подкласс SimpleDateFormat из пакета Java. text.

Класс DateFormat предлагает 4 стиля представления даты и времени:

- стиль SHORT представляет дату и время в коротком числовом виде: 27.04.01 17:32; в локали США: 4/27/01 5:32 PM;
- стиль MEDIUM задает год 4-мя цифрами и показывает секунды: 27.04.2001 17:32:45; в локали США месяц представляется 3-мя буквами;
- стиль LONG представляет месяц словом и добавляет часовой пояс: 27 апрель 2001 г. 17:32:45 GMT+03.-00;
- стиль FULL в русской локзли таков же, как и стиль LONG ; в локали США добавляется еще день недели.

Есть еще стиль DEFAULT, совпадающий со стилем MEDIUM.

При создании объекта класса simpieDateFormat можно задать в конструкторе шаблон, определяющий какой-либо другой формат, например:

```
SimpieDateFormat sdf = new SimpieDateFormat("dd-MM-yyyy hh.iran");
System.out.println(sdf.format(new Date()));
```

Получим вывод в таком виде: 27-04-2001 17.32.

В шаблоне буква d означает цифру дня месяца, м — цифру месяца, у — цифру года, h — цифру часа, m — цифру минут. Остальные обозначения для шаблона указаны в документации по классу SimpieDateFormat.

Эти буквенные обозначения можно изменить с помощью класса DateFormatSymbols.

Не во всех локалях можно создать объект класса SimpieDateFormat. В таких случаях используются статические методы getinstance() класса DateFormat, возвращающие объект класса DateFormat. Параметрами этих методов служат стиль представления даты и времени и, может быть, локаль.

После создания объекта метод format() класса DateFormat возвращает строку с датой и временем, согласно заданному стилю. В качестве аргумента задается объект класса Date.

Например:

```
System.out.println("LONG: " + DateFormat.getDateTimeInstance
    (DateFormat. LONG, DateFormat. LONG). format (new Date ()));
```

или

```
System.out.println("FULL: " + DateFormat.getDateTimeInstance
    (DateFormat.FULL,DateFormat.FULL, Locale.US).format(new Date()));
```

15.22. Классы случайных чисел

Доступные методы определены в классах Math и Random.

15.22.1. Класс Math

Если нужны случайные числа, с которыми не требуются упорядочивания, то можно употребить простой метод

```
    Java.lang.Math.random(),
```

который возвращает значения типа double в пределах от 0 до 1.0.

15.22.2. Класс Random

В классе Random определены методы с более широкими возможностями. Для доступа к ним сначала нужно создать объект Rnd типа Random, а затем употреблять его методы.

```
Random Rnd = new Random();
```

Есть несколько конструкторов:

- Random(). В качестве стартового числа использует значение, полученное из текущего времени.
- Random(long seed). В качестве стартового числа использует значение seed.

Для объекта Rnd доступны:

- Метод Rnd.setSeed(long seed).
- Метод Rnd.nextInt(). Случайное число типа int, равномерно распределенное между величинами Integer.MIN_VALUE и Integer.MAX_VALUE включительно.
- Метод Rnd.nextLong(). Случайное число типа long, равномерно распределенное между величинами Long.MIN_VALUE и Long.MAX_VALUE включительно.
- Метод Rnd.nextFloat(). Случайное число типа float, равномерно распределенное между величинами Float.MIN_VALUE и Float.MAX_VALUE включительно.
- Метод Rnd.nextDouble(). Случайное число типа double, равномерно распределенное между величинами Double.MIN_VALUE и Double.MAX_VALUE включительно.
- Метод Rnd.nextGaussian(). Случайное число типа double, с распределением Гаусса с математическим ожиданием 0.0 и стандартным отклонением 1.0.

15.23. Класс Runtime - взаимодействие с системой

Класс System позволяет осуществить и некоторое взаимодействие с системой во время выполнения программы (run time). Но кроме него для этого есть специальный класс Runtime.

Класс Runtime содержит методы взаимодействия с JVM во время выполнения программы. Каждое приложение может получить только один экземпляр данного класса статическим методом getRuntime(). Все вызовы этого метода возвращают ссылку на один и тот же объект.

Методы freeMemory() и totalMemory() - возвращают количество свободной и всей памяти, находящейся в распоряжении JVM для размещения объектов, в байтах, в виде числа типа long. Не стоит твердо опираться на эти числа, поскольку количество памяти меняется динамически.

Метод exit(int status) - запускает процесс останова JVM и передает операционной системе статус завершения status. По соглашению, ненулевой статус означает ненормальное завершение. Удобнее использовать аналогичный метод класса system, который является статическим.

Метод halt(int status) - осуществляет немедленный останов JVM. Он не завершает запущенные процессы нормально и должен использоваться только в аварийных ситуациях.

Метод loadLibrary(String libName) - позволяет подгрузить динамическую библиотеку во время выполнения по ее имени libName.

Метод load (String fileName) - подгружает динамическую библиотеку по имени файла fileName, в котором она хранится.

Метод gc() - запускает процесс освобождения ненужной оперативной памяти (garbage collection). Этот процесс периодически запускается самой виртуальной машиной Java и выполняется в фоне с небольшим приоритетом, но можно его запустить и из программы. Опять-таки удобнее использовать статический метод System.gc().

Наконец, несколько методов exec() запускают в отдельных процессах исполнимые файлы. Аргументом этих методов служит командная строка исполнимого файла.

Например, Runtime.getRuntime().exec ("notepad") запускает программу notepad.

16. Графический интерфейс

16.1. Принципы построения

В предыдущих главах мы писали программы, связанные с текстовым терминалом и запускающиеся из командной строки. Такие программы называются консольными приложениями. Они разрабатываются для выполнения на серверах, там, где не требуется интерактивная связь с пользователем. Программы, тесно взаимодействующие с пользователем, воспринимающие сигналы от клавиатуры и мыши, работают в графической среде. Каждое приложение, предназначенное для работы в графической среде, должно создать хотя бы одно окно, в котором будет происходить его работа, и зарегистрировать его в графической оболочке операционной системы, чтобы окно могло взаимодействовать с операционной системой и другими окнами: перекрываться, перемещаться, менять размеры, сворачиваться в ярлык.

Есть много различных графических систем: MS Windows, X Window System, Macintosh. В каждой из них свои правила построения окон и их компонентов: меню, полей ввода, кнопок, списков, полос прокрутки. Эти правила сложны и запутанны. Графические API содержат сотни функций.

Для облегчения создания окон и их компонентов написаны библиотеки классов: MFC, Motif, OpenLook, Qt, Tk, Xview, OpenWindows и множество других. Каждый класс такой библиотеки описывает сразу целый графический компонент, управляемый методами этого и других классов.

В технологии Java дело осложняется тем, что приложения Java должны работать в любой или хотя бы во **многих** графических средах. Нужна библиотека классов, независимая от конкретной графической системы. В первой версии JDK задачу рещили следующим образом: были разработаны интерфейсы, содержащие методы работы с графическими объектами. Классы библиотеки AWT реализуют эти интерфейсы для создания приложений. Приложения Java используют данные методы для размещения и перемещения графических объектов, изменения их размеров, взаимодействия объектов.

С другой стороны, для работы с экраном в конкретной графической среде эти интерфейсы реализуются в каждой такой среде отдельно. В каждой графической оболочке это делается по-своему, средствами этой оболочки с помощью, графических библиотек данной операционной системы. Такие интерфейсs были названы peer-интерфейсами.

Библиотека классов Java, основанных на peer-интерфейсах, получила название AWT (Abstract Window Toolkit). При выводе объекта, созданного в приложении Java и основанного на peer-интерфейсе, на экран создается парный ему

(peer-to-peer) объект графической подсистемы операционной системы, который и отображается на экране. Эти объекты тесно взаимодействуют во время работы приложения. Поэтому графические объекты AWT в каждой графической среде имеют вид, характерный для этой среды: в MS Windows, Motif, OpenLook, OpenWindows, везде окна, созданные в AWT, выглядят как "родные" окна.

В версии JDK 1.1 библиотека AWT была переработана. В нее добавлена возможность создания компонентов, полностью написанных на Java и не зависящих от peer-интерфейсов. Такие компоненты стали называть "легкими" (lightweight) в отличие от компонентов, реализованных через peer-интерфейсы, названных "тяжелыми" (heavy).

"Легкие" компоненты везде выглядят одинаково, сохраняют заданный при создании вид (look and feel). Более того, приложение можно разработать таким образом, чтобы после его запуска можно было выбрать какой-то определенный вид: Motif, Metal, Windows XP или какой-нибудь другой, и сменить этот вид в любой момент работы.

Эта интересная особенность "легких" компонентов получила название PL&F (Pluggable Look and Feel) или "plaf.

Была создана обширная библиотека "легких" компонентоэ Java, названная **Swing**. В ней были переписаны все компоненты библиотеки AWT, так что библиотека Swing может использоваться самостоятельно, несмотря на то, что все классы из нее расширяют классы библиотеки AWT.

Библиотека классов Swing поставлялась как дополнение к JDK 1.1. В состав Java 2 SDK она включена как основная графическая библиотека классов, реализующая идею "100% Pure Java", наряду с AWT.

В Java 2 библиотека AWT значительно расширена добавлением новых средств рисования, вывода текстов и изображений, получивших название Java 2D, и средств, реализующих перемещение текста методом DnD (Drag and Drop).

Кроме того, в Java 2 включены новые методы ввода/вывода Input Method Framework и средства связи с дополнительными устройствами ввода/вывода, такими как световое перо или клавиатура Бройля, названные Accessibility.

Все эти средства Java 2: AWT, Swing, Java 2D, DnD, Input Method Framework и Accessibility составили библиотеку графических средств Java, названную JFC (Java Foundation Classes).

Описание каждого из этих средств составит целую книгу, поэтому мы вынуждены ограничиться представлением только основных средств библиотеки AWT.

16.2. Класс Graphics

16.2.1. Графические примитивы

При создании компонента, т.е. объекта класса Component, автоматически формируется его графический контекст (graphics context). В контексте размещается область рисования и вывода текста и изображений. Контекст содержит текущий и альтернативный цвет рисования и цвет фона - объекты класса Color, текущий шрифт для вывода текста - объект класса Font.

В контексте определена система координат, начало которой с координатами (0,0) расположено в верхнем левом углу области рисования, ось Ох направлена вправо, ось Оу - вниз. Точки координат находятся между пикселами.

Управляет контекстом класс Graphics или новый класс Graphics2D, введенный в Java 2. Поскольку графический контекст сильно зависит от конкретной графической платформы, эти классы сделаны абстрактными. Поэтому нельзя непосредственно создать экземпляры класса Graphics или Graphics2D.

Однако каждая виртуальная машина Java реализует методы этих классов, создает их экземпляры для компонента и предоставляет объект класса Graphics методом getGraphics() класса Component или как аргумент методов paint() и update().

Посмотрим сначала, какие методы работы с графикой и текстом предоставляет нам класс Graphics.

При создании контекста в нем задается текущий цвет для рисования, обычно черный, и цвет фона области рисования - белый или серый. Изменить текущий цвет можно методом setcoior (Color newCoior), аргумент newcoior которого - объект класса Color.

Узнать текущий цвет можно методом getColor(), возвращающим объект класса Color.

16.2.2. Задание цвета

Цвет - объект класса Color. Основу класса составляют 7 конструкторов цвета. Самый простой конструктор 1:

    Color(int red, int green, int blue)

создает цвет, получающийся как смесь красной red, зеленой green и синей blue составляющих. Эта цветовая модель называется RGB. Каждая составляющая меняется от 0 (отсутствие составляющей) до 255 (полная интенсивность этой составляющей). Например:

```
Color pureRed = new Color(255, 0, 0);
Color pureGreen = new Color(0, 255, 0);
```

определяют чистый ярко-красный pureRed и чистый ярко-зеленый pureGreen цвета.

В конструкторе 2 интенсивность составляющих можно изменять более гладко вещественными числами от 0.0 (отсутствие составляющей) до 1.0 (полная интенсивность составляющей):

```
Color(float red, float green, float blue)
```

Например:

```
Color someColor = new Color(O.OSf, 0.4f, 0.95f);
```

Конструктор 3

```
Color(int rgb)
```

задает все три составляющие в одном целом числе. В битах 16—23 записывается красная составляющая, в битах 8—15 — зеленая, а в битах 0—7 — синяя составляющая цвета. Например:

```
Color c = new Color(OXFF8F48FF);
```

Здесь красная составляющая задана с интенсивностью 0x8F, зеленая — 0x48, синяя — 0xFF.

Следующие 3 конструктора

- Color(int red, int green, int blue, int alpha)
- Color(float red, float green, float blue, float alpha)
- Color(int rgb, Boolean hasAlpha)

вводят четвертую составляющую цвета, так называемую "альфу", определяющую прозрачность цвета. Эта составляющая проявляет себя при наложении одного цвета на другой. Если альфа равна 255 или 1,0, то цвет совершенно непрозрачен, предыдущий цвет не просвечивает сквозь него. Если альфа равна 0 или 0,0, то цвет абсолютно прозрачен, для каждого пиксела виден только предыдущий цвет.

Последний из этих конструкторов учитывает составляющую альфа, находящуюся в битах 24—31, если параметр hasAlpha равен true. Если же hasAlpha равно false, то составляющая альфа считается равной 255, независимо от того, что записано в старших битах параметра rgb.

Первые 3 конструктора создают непрозрачный цвет с альфой, равной 255 или 1,0.

Конструктор 7

Color(ColorSpace cspace, float[] components, float alpha)

позволяет создавать цвет не только в цветовой модели (color model) RGB, но и в других моделях: CMYK, HSB, CIEXYZ, определенных объектом класса

ColorSpace.

Для создания цвета в модели HSB можно воспользоваться статическим методом

getHSBColor(float hue, float saturation, float brightness).

Если нет необходимости тщательно подбирать цвета, то можно просто воспользоваться одной из 13 статических констант типа Color, имеющихся в классе Color. Вопреки соглашению "Code Conventions" они записываются строчными буквами: black, blue, cyan, darkGray, gray, green, lightgray, magenta, orange, pink, red, white, yellow.

Методы класса Color позволяют получить составляющие текущего цвета: getRed(), getGreen(), getBlue(), getAlpha(), getRGB(), getColorSpace(), getComponents ().

2 метода создают более яркий brighter() и более темный darker() цвета по сравнению с текущим цветом. Они полезны, если надо выделить активный компонент или, наоборот, показать неактивный компонент бледнее остальных компонентов.

2 статических метода возвращают цвет, преобразованный из цветовой модели RGB в HSB и обратно:

float[] RGBtoHSB(int red, int green, int blue, float[] hsb)
int HSBtoRGB(int hue, int saturation, int brightness)

Создав цвет, можно рисовать им в графическом контексте.

16.2.3. Рисованные примитивы

16.2.4. Основной метод рисования

drawLine(int x1, int y1, int x2, int y2)

вычерчивает текущим цветом отрезок прямой между точками с координатами (x1, y1) и (x2, y2).

Одного этого метода достаточно, чтобы нарисовать любую картину по точкам, вычерчивая каждую точку с координатами (x, y) методом drawLine (x1, y1, x2, y2) и меняя цвета от точки к точке. Но никто, разумеется, не станет этого делать. Лучше использовать методы рисование графических примитивов:

- drawRect(int x, int y, int width, int height) - чертит прямоугольник со сторонами, параллельными краям экрана, задаваемый координатами верхнего левого угла (x, y), шириной width пикселов и высотой height пикселов;
- draw3DRect(int x, int y, int width, int height, Boolean raised) - чертит прямоугольник, как будто выделяющийся из плоскости рисования. если аргумент raised равен true, или как будто вдавленный в плоскость, если аргумент raised равен false;
- drawOval(int x, int y, int width, int height) - чертит овал, вписанный в прямоугольник, заданный аргументами метода. Если width == height, то получится окружность;
- drawArc(int x, int y, int width, int height, int startAngle, int arc) - чертит дугу овала, вписанного в прямоугольник, заданный первыми 4 аргументами. Дуга имеет величину arc градусов и отсчитывается от угла startAngle. Угол отсчитывается в градусах от оси Ox. Положительный угол отсчитывается против часовой стрелки, отрицательный - по часовой стрелке;
- drawRoundRect (int x, int y, int width, int height, int arcWidth, int arcHeight) - чертит прямоугольник с закругленными краями. Закругления вычерчиваются четвертинками овалов, вписанных в прямоугольники шириной arcwidth и высотой arcHeight, построенные в углах основного прямоугольника;
- drawPolyline(int[] xPoints, int[] yPoints, int nPoints) - чертит ломаную с вершинами в точках <xPoints[i], ypoints[i]) и числом вершин nPoints;
- drawPolygon(int[] xPoints, int[] yPoints, int nPoints) - чертит замкнутую ломаную, проводя замыкающий отрезок прямой между первой и последней точкой;
- drawFoiygon(Polygon p) - чертит замкнутую ломаную, вершины которой заданы объектом p класса Polygon.

Класс Polygon предназначен для работы с многоугольником, в частности, с треугольниками и произвольными четырехугольниками.

Объекты этого класса можно создать конструкторами:

- Polygon() - создает пустой объект;
- Polygon(int[] xPoints, int[] yPoints, int nPoints) - задаются вершины многоугольника (xPoints[i], yPoints[i]) и их число nPoints. После создания объекта в него можно добавлять вершины методом addPoint(int x, int y).

Логические методы contains() позволяют проверить, не лежит ли в многоугольнике заданная аргументами метода точка, отрезок прямой или целый прямоугольник со сторонами, параллельными сторонам экрана.

Логические методы intersects() позволяют проверить, не пересекается ли с данным многоугольником отрезок прямой, заданный аргументами метода, или прямоугольник со сторонами, параллельными сторонам экрана.

Методы getBounds() и getBounds2D() возвращают прямоугольник, целиком содержащий в себе данный многоугольник.

16.2.5. Примитивы с заливкой

Несколько методов вычерчивают фигуры, залитые текущим цветом:

- fillRect(),
- fill3DRect(),
- filiArc(),
- fillOval(),
- fillPoiygon(),
- fillRoundRect().

У них такие же аргументы, как и у соответствующих методов, вычерчивающих незаполненные фигуры.

Например, если вы хотите изменить цвет фона области рисования, то установите новый текущий цвет и начертите им заполненный прямоугольник величиной во всю область:

```
public void paint(Graphics g) (
    Color initColor = g.getColor();    // Сохраняем исходный цвет
    g.setColor(new Color(0, 0, 255));  // Устанавливаем цвет фона
                        // Заливаем область рисования
    g.fillRect(0, 0, getSizeO.width - 1, getSize().height - 1);'
    g.setColor(initColor);             // Восстанавливаем исходный цвет
                        // Дальнейшие действия
}
```

16.2.6. Вывод текста

Для вывода текста в область рисования текущим цветом и шрифтом, начиная с точки (x, y), в классе Graphics есть несколько методов:

- drawsSring (String s, int x, int y) - выводит строку s;
- drawBytes(byte[] b, int offset, int length, int x, int y) - выводит length элементов массива байтов b, начиная с индекса offset;

- drawChars(chart[] ch, int offset, int length, int x, int y) - выводит length элементов массива символов ch, начиная с индекса offset.
- drawString(AttributedCharacteriterator iter, int x, int y) - выводит текст, занесенный в объект класса, реализующего интерфейс AttributedCharacteriterator. Это позволяет задавать свой шрифт для каждого выводимого симвбла. Точка (x, y) - это левая нижняя точка первой буквы текста на базовой линии (baseline) вывода шрифта.

Установка шрифта.

Метод setFont(Font newFont) класса Graphics устанавливает текущий шрифт для вывода текста.

Метод getFont () возвращает текущий шрифт.

Шрифт - это объект класса Font. Посмотрим, какие возможности предоставляет этот класс.

Объекты класса Font хранят начертания (glyphs) символов, образующие шрифт. Их можно создать конструкторами:

- Font(Map attributes) - задает шрифт с заданными аргументом attributes атрибутами. Ключи атрибутов и некоторые их значения задаются константами класса TextAttnbute из пакета java.awt.font. Этот конструктор характерен для Java 2D и будет рассмотрен далее в настоящей главе.
- Font(String name, int style, int size) - задает шрифт по имени name, со стилем style и размером size типографских пунктов. Этот конструктор характерен для JDK 1.1, но широко используется и в Java 2D в силу своей простоты.

Типографский пункт в России и некоторых европейских странах равен 0,376мм, Точнее, 1/72 части французского дюйма. В англо-американской системе мер пункт равен 1/72 части английского дюйма, 0,351мм. Этот пункт и применяется в компьютерной графике.

Имя шрифта name может быть строкой с физическим именем шрифта, например, "Courier New", или одна из строк "Dialog", "Dialoglnput",' "Monospaced", "Serif", "SansSerif", "Symbol". Это так называемые логические имена шрифтов (logical font names). Если name = null, то задается шрифт по умолчанию.

Стиль шрифта style — это одна из констант класса Font:

- BOLD — полужирный;
- ITALIC — курсив;
- PLAIN — обычный.

При выводе текста логическим именам шрифтов и стилям сопоставляются физические имена шрифтов (font face name) или имена семейств шрифтов (font name). Это имена реальных шрифтов, имеющихся в графической подсистеме операционной системы.

Например, логическому имени "Serif" может быть сопоставлено имя семейства (family) шрифтов Times New Roman, а в сочетании со стилями — конкретные физические имена Times New Roman Bold, Times New Roman Italic. Эти шрифты должны находиться в составе шрифтов графической системы той машины, на которой выполняется приложение.

Список имен доступных шрифтов можно просмотреть следующими операторами:

```
Font[] fnt = Toolkit.getGraphicsEnvironment.getAHFonts();
for (int i = 0; i< fnt.length; i++)
    System.out.println(fnt[i].getFontName());
```

В состав SUN J2SDK входит семейство шрифтов Lucida. Установив SDK, вы можете быть уверены, что эти шрифты есть в вашей системе.

Таблицы сопоставления логических и физических имен шрифтов находятся в файлах с именами

- font.properties;
- font.properties.ar;
- font.properties.ja;
- font.properties.ru.

и т.д.

Нужный файл выбирается виртуальной машиной Java по окончании имени файла. Это окончание совпадает с международным кодом языка, установленного в локали или в системном свойстве user.language. Если у вас установлена русская локаль с международным кодом языка "ru", то для сопоставления будет выбран файл font.properties.ru. Если такой файл не найден, то применяется файл font.properties, не соответствующий никакой конкретной локали.

Поэтому можно оставить в системе только один файл font.properties, переписав в него содержимое нужного файла или создав файл заново. Для любой локали будет использоваться этот файл.

При выводе строки в окно приложения очень часто возникает необходимость расположить ее определенным образом относительно других элементов изображения: центрировать, вывести над или под другим графическим объектом.

Для этого надо знать метрику строки: ее высоту и ширину. Для измерения размеров отдельных символов и строки в целом разработан класс FontMetrics.

В Java 2D класс FontMetrics заменен классом TextLayout. Его мы рассмотрим в конце этой главы, а сейчас выясним, какую пользу можно извлечь из методов класса FontMetrics.

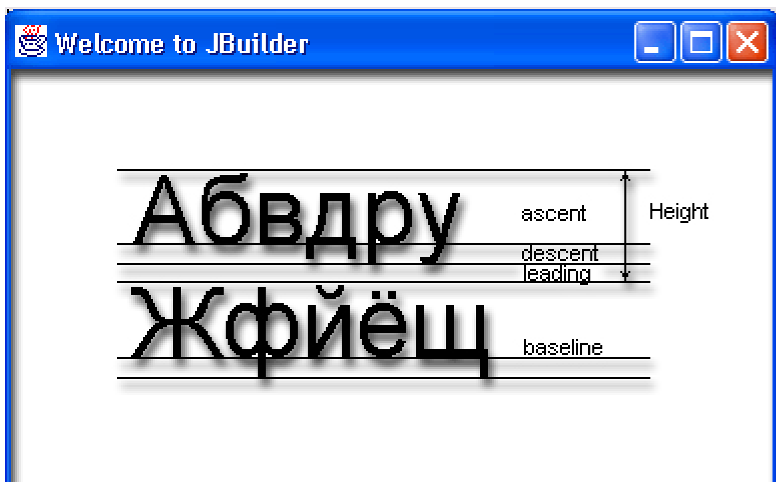
Класс FontMetrics

Класс FontMetrics является абстрактным, поэтому нельзя воспользоваться его конструктором. Для получения объекта класса FontMetrics, содержащего набор метрических характеристик шрифта f, надо обратиться к методу getFontMetrics(f) класса Graphics или класса Component.

Класс FontMetri.cs позволяет узнать ширину отдельного символа ch в пикселах методом charwidth(ch), общую ширину всех символов массива или под-массива символов или байтов методами getchars() и getBytes(), ширину целой строки str в пикселах методом stringwidth(str).

Несколько методов возвращают в пикселах вертикальные размеры шрифта.

- Интерлиньяж (leading) - расстояние между нижней точкой свисающих элементов букв (как р, у) строки и верхней точкой выступающих элементов букв (как б) следующей строки - возвращает метод getLeading().
- Среднее расстояние от базовой линии шрифта (baseline) до верхней точки прописных букв и выступающих элементов той же строки (ascent) возвращает метод getAscent(), а максимальное - метод getMaxAscent().
- Среднее расстояние свисающих элементов от базовой линии той же строки возвращает метод getDescent(), а максимальное - метод getMaxDescent().
- Высоту шрифта (height) - сумму ascent + descent + leading - возвращает метод getHeight(). Высота шрифта равна расстоянию между базовыми линиями соседних строк.



Дополнительные характеристики шрифта можно определить методами класса LineMetrics из пакета java.awt.font. Объект этого класса можно получить несколькими методами getLineMetrics() класса FontMetrics.

В Java 2 класс Graphics, в рамках системы Java 2D, значительно расширен классом Graphics2D.

16.3. Возможности Java 2D

16.3.1. Введение

В систему пакетов и классов Java 2D, основа которой - класс Graphics2D пакета java.awt, внесено несколько принципиально новых положений.

Кроме координатной системы, принятой в классе Graphics и названной координатным пространством пользователя (User Space), введена еще система координат устройства вывода (Device Space): экрана монитора, принтера. Методы класса Graphics2D автоматически переводят (transform) систему координат пользователя в систему координат устройства при выводе графики.

Преобразование координат пользователя в координаты устройства можно задать "вручную", причем преобразованием способно служить любое аффинное преобразование плоскости, в частности, поворот на любой угол и/или сжатие/растяжение. Оно определяется как объект класса AffineTransform. Его можно установить как преобразование по умолчанию методом setTransform(). Возможно выполнять преобразование "на лету" методами transform() и translate() и делать композицию преобразований методом concatenate().

Поскольку аффинное преобразование вещественно, координаты задаются вещественными, а не целыми числами.

Графические примитивы: прямоугольник, овал, дуга и др., реализуют теперь новый интерфейс Shape пакета java.awt. Для их вычерчивания можно использовать новый единый для всех фигур метод draw(), аргументом которого способен служить любой объект, реализовавший интерфейс Shape. Введен метод fill(), заполняющий фигуры - объекты класса, реализовавшего интерфейс Shape.

Для вычерчивания (stroke) линий введено понятие пера (реп). Свойства пера описывает интерфейс Stroke. Класс Basicstroke реализует этот интерфейс. Перо обладает характеристиками:

- оно имеет толщину (width) в один (по умолчанию) или несколько пикселов;
- оно может закончить линию (end cap) закруглением - статическая константа CAP_ROUND, прямым обрезом - CAP_SQUARE (по умолчанию), или не фиксировать определенный способ окончания - CAP_BUTT;
- оно может сопрягать линии (line joins) закруглением - статическая константа JOIN_ROOND, отрезком прямой — JOIN_BEVEL, или просто состыковывать - JOIN_MITER (по умолчанию);
- оно может чертить линию различными пунктирами (dash) и штрих-пунктирами, длины штрихов и промежутков задаются в массиве, элементы массива с четными индексами задают длину штриха, с нечетными индексами - длину промежутка между штрихами.

Методы заполнения фигур описаны в интерфейсе Paint. 3 класса реализуют этот интерфейс. Класс Color реализует его сплошной (solid) заливкой, класс GradientPaint - градиентным (gradient) заполнением, при котором цвет плавно меняется от одной заданной точки к другой заданной точке, класс TexturePaint - заполнением по предварительно заданному образцу (pattern fill).

Буквы текста понимаются как фигуры, т.е. объекты, реализующие интерфейс Shape, и могут вычерчиваться методом draw() с использованием всех возможностей этого метода. При их вычерчивании применяется перо, все методы заполнения и преобразования.

Кроме имени, стиля и размера, шрифт получил много дополнительных атрибутов, например, преобразование координат, подчеркивание или перечеркивание текста, вывод текста справа налево. Цвет текста и его фона являются теперь атрибутами самого текста, а не графического контекста. Можно задать разную ширину символов шрифта, надстрочные и подстрочные индексы. Атрибуты устанавливаются константами класса TextAttribute.

Процесс визуализации (rendering) регулируется правилами (hints), определенными константами класса RenderingHints.

С такими возможностями Java 2D стала полноценной системой рисования, вывода текста и изображений. Посмотрим, как реализованы эти возможности, и как ими можно воспользоваться.

16.3.2. Преобразование координат

Правило преобразования координат пользователя в координаты графического устройства (transform) задается автоматически при создании графического контекста так же, как цвет и шрифт. В дальнейшем его можно изменить методом setTransform() так же, как меняется цвет или шрифт. Аргументом этого метода служит объект класса AffineTransform из пакета java.awt.geom, подобно объектам класса Color или Font при задании цвета или шрифта.

Класс AffineTransform. Аффинное преобразование координат задается основными конструкторами класса AffineTransform:

- AffineTransform(double a, double b, double c, double d, double e, double f)
- AffineTransform (float a, float b, float c, float d, float e, float f)

При этом точка с координатами (x, y) в пространстве пользователя перейдет в точку с координатами (a*x+c*y+e, b*x+d*y+f) в пространстве графического устройства.

Такое преобразование не искривляет плоскость - прямые линии переходят в прямые, углы между линиями сохраняются. Примерами аффинных преобразований служат повороты вокруг любой точки на любой угол, параллельные сдвиги, отражения от осей, сжатия и растяжения по осям.

Следующие 2 конструктора используют в качестве аргумента массив (a, b, c, d, e, f} или (a, b, c, d}, если e = f = 0, составленный из таких же коэффициентов в том же порядке:

- AffineTransform(double[] arr)
- AffineTransform(float[] arr)

Конструктор 5 создает новый объект по другому, уже имеющемуся, объекту:

- AffineTransform(AffineTransform at)

Конструктор 6 (конструктор по умолчанию) создает тождественное преобразование:

- AffineTransform()

Эти конструкторы математически точны, но неудобны при задании конкретных преобразований. Попробуйте рассчитать коэффициенты поворота на 43° вокруг точки с координатами (10, 50).

Во многих случаях удобнее создать преобразование статическими методами, возвращающими объект класса AffineTransform.

- getRotateInstance (double angle) — возвращает поворот на угол angle, заданный в радианах, вокруг начала координат. Положительное направление поворота таково, что точки оси Ox поворачиваются в направлении к оси Oy. Если оси координат пользователя не менялись преобразованием отражения, то положительное значение angle задает поворот по часовой стрелке.
- getRotateInstance(double angle, double x, double y) - такой же поворот вокруг точки с координатами (x, y).
- getScaleInstance(double sx, double sy) - изменяет масштаб по оси Ox в sx раз, по оси Oy в sy раз.
- getShareInstance(double shx, double shy) - преобразует каждую точку (x, y) в точку (x+shx*y, shy*x+y).
- getTranslateInstance(double tx, double ty) - сдвигает каждую точку (x, y) в точку (x+tx, y+ty).
- Метод createInverse() возвращает преобразование, обратное текущему преобразованию.

После создания преобразования его можно изменить методами:

- setTransform(AffineTransform at)
- setTransform(double a, double b, double c, double d, double e, double f)
- setToIdentity()
- setToRotation(double angle)
- setToRotation(double angle, double x, double y)
- setToScale(double sx, double sy)
- setToShare(double shx, double shy)
- setToTranslate(double tx, double ty)

сделав текущим преобразование, заданное одним из этих методов.

Преобразования, заданные методами:

- concatenate(AffineTransform at)
- rotate(double angle)
- rotate(double angle, double x, double y)
- scale(double sx, double sy)
- shear(double shx, double shy)

- translate(double tx, double ty)

выполняются перед текущим преобразованием, образуя композицию преобразований.

Преобразование, заданное методом preconcatenate{AffineTransform at), напротив, осуществляется после текущего преобразования.

Прочие методы класса AffineTransform производят преобразования различных фигур в пространстве пользователя.

16.3.3. Рисование фигур

Характеристики пера для рисования фигур описаны в интерфейсе Stroke. В Java 2D есть пока только один класс, реализующий этот интерфейс — класс BasicStroke.

Конструкторы класса BasicStroke определяют характеристики пера. Основной конструктор

BasicStroke(float width, int cap, int join, float miter, float[] dash, float dashBegin)

задает:

- толщину пера width в пикселах;
- оформление конца линии cap; это одна из констант:
- CAP_ROUND - закругленный конец линии;
- CAP_SQUARE - квадратный конец линии;
- CAP_BUTT - оформление отсутствует;
- способ сопряжения линий join; это одна из констант:
- JOIN_ROUND - линии сопрягаются дугой окружности;
- JOIN_BEVEL - линии сопрягаются отрезком прямой, перпендикуляр-ным биссектрисе угла между линиями;
- JOIN_MITER - линии просто стыкуются; расстояние между линиями miter, начиная с которого применяется сопряжение JOIN_MITER;
- длину штрихов и промежутков между штрихами - массив dash; элементы массива с четными индексами задают длину штриха в пикселах, элементы с нечетными индексами - длину промежутка; массив перебирается циклически;
- индекс dashBegin, начиная с которого перебираются элементы массива dash.

Остальные конструкторы задают некоторые характеристики по умолчанию:

- BasicStroke (float width, int cap, int join, float miter) — сплошная линия;

- BasicStroke (float width, int cap, int join) - сплошная линия с сопряжением JOIN_ROUND или JOIN_BEVEL; для сопряжения JOIN_MITER задается значение miter = 10.0f;
- BasicStroke (float width) - прямой обрез CAP_SQUARE и сопряжение JOIN_MITER со значением miter = 10.0f;
- BasicStroke() - ширина1. 0f.

После создания пера одним из конструкторов и установки пера методом setStroke о можно рисовать различные фигуры методами draw() и fill().

Общие свойства фигур, которые можно нарисовать методом draw() класса Graphics2D, описаны в интерфейсе shape. Этот интерфейс реализован для создания обычного набора фигур - прямоугольников, прямых, эллипсов, дуг, точек - классами Rectangle2D, RoundRectangle2D, Line2D, Ellipse2D, Arc2D, Point2D пакета java.awt.geom. В этом пакете есть еще классы Cubiccurve2D и QuadCurve2D для создания кривых третьего и второго порядка.

Все эти классы абстрактные, но существуют их реализации — вложенные классы Double и Float для задания координат числами соответствующего типа.

16.3.4. Класс GeneralPath

В пакете java.awt.geom есть еще класс GeneralPath. Объекты этого класса могут содержать сложные конструкции, составленные из отрезков прямых или кривых линий и прочих фигур, соединенных или не соединенных между собой. Более того, поскольку этот класс реализует интерфейс shape, его экземпляры сами являются фигурами и могут быть элементами других объектов класса GeneralPath.

Вначале создается пустой объект класса GeneralPath конструктором по умолчанию GeneralPath() или объект, содержащий одну фигуру, конструктором GeneralPath (Shape sh).

Затем к этому объекту добавляются фигуры методом append(Shape sh, Boolean connect)

Если параметр connect равен true, то новая фигура соединяется с предыдущими фигурами с помощью текущего пера.

В объекте есть текущая точка. Вначале ее координаты (0, 0), затем ее можно переместить в точку (x, y) методом moveTo (float x, float y).

От текущей точки к точке (x, y) можно провести:

- отрезок прямой методом lineTo(float x, float y);

- отрезок квадратичной кривой методом quadTo(float xi, float yl, float x, float y),
- кривую Безье методом curveTo(float xl, float yl, float x2, float y2, float x, float y).

Текущей точкой после этого становится точка (x, y). Начальную и конечную точки можно соединить методом ciosePath(). Вот как можно создать треугольник с заданными вершинами:

```
GeneralPath p = new GeneralPath();
p.moveTo(xl, yl); // Переносим текущую точку в первую вершину,
p.lineTo(x2, y2); // проводим сторону треугольника до второй вершины,
p.lineTo(x3, y3); // проводим вторую сторону,
p.closePathf);    // проводим третью сторону до первой вершины
```

Способы заполнения фигур определены в интерфейсе Paint. В настоящее время Java 2D содержит 3 реализации этого интерфейса - классы Color, GradientPaint и TexturePamt. Класс Color нам известен, посмотрим, какие способы заливки предлагают классы GradientPaint И TexturePaint.

16.3.5. Классы GradientPaint и TexturePaint

Класс GradientPaint предлагает сделать заливку следующим образом.

В двух точках M и N устанавливаются разные цвета. В точке M(x1, y1) задается цвет c1, в точке N(x2, y2) - цвет c2. Цвет заливки гладко меняется от c1 к c2 вдоль прямой, соединяющей точки M и N, оставаясь постоянным вдоль каждой прямой, перпендикулярной прямой MN. Такую заливку создает конструктор 1

```
GradientPaint(float x1, float y1, Color c1, float x2, float y2, Color c2)
```

При этом вне отрезка MN цвет остается постоянным: за точкой M— цвет c1, за точкой N — цвет c2.

Конструктор 2 при задании параметра cyclic=true повторяет заливку полосы мы во всей заливаемой фигуре.

GradientPaint(float xl, float yl, Color cl, float x2, float y2, Color c2, Boolean cyclic)

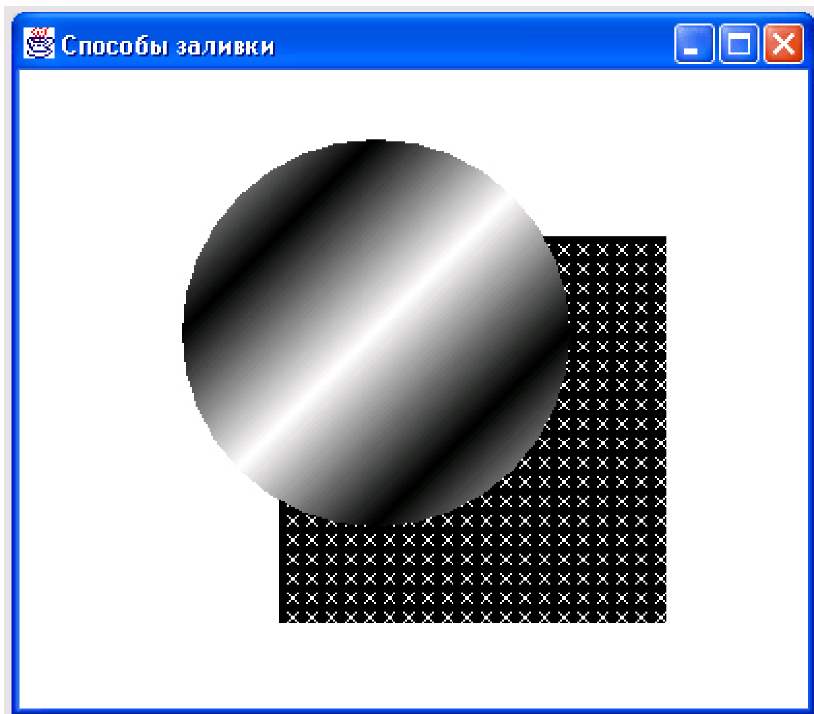
Еще два конструктора задают точки как объекты класса Point2D.

Класс TexturePaint поступает сложнее. Сначала создается буфер — объект класса Bufferedlmage из пакета java.awt. image. Его графический контекст управляется экземпляром класса Graphics2D. Этот экземпляр можно получить методом createGraphics() класса Bufferedlmage. Графический контекст буфера заполняется фигурой, которая будет служить образцом заполнения.

Затем по буферу создается объект класса TexturePaint. При этом еще задается прямоугольник, размеры которого будут размерами образца заполнения. Конструктор выглядит так:

TexturePaint(BufferedImage buffer, Rectangle2D anchor)

После создания заливки — объекта класса color, GradientPaint или TexturePaint - она устанавливается в графическом контексте методом setPaint(Paint p) и используется в дальнейшем методом fill(Shape sh). Пример:



16.3.6. Вывод текста средствами Java 2D

Шрифт - объект класса Font - кроме имени, стиля и размера имеет еще полтора десятка атрибутов: подчеркивание, перечеркивание, наклон, цвет шрифта и цвет фона, ширину и толщину символов, аффинное преобразование, расположение слева направо или справа налево.

Атрибуты шрифта задаются как статические константы класса TextAttribute. Наиболее используемые атрибуты перечислены в таблице.

Атрибут	Значение
BACKGROUND	Цвет фона. Объект, реализующий интерфейс Paint
FOREGROUND	Цвет текста. Объект, реализующий интерфейс Paint
BIDI EMBEDDED	Уровень вложенности просмотра текста. Целое от 1 до 1 5
CHAR_REPLACEMENT	Фигура, заменяющая символ. Объект GraphicAttribute
FAMILY	Семейство шрифта. Строка типа string
FONT	Шрифт. Объект класса Font
JUSTIFICATION	Допуск при выравнивании абзаца. Объект класса Float со значениями от 0,0 до 1,0. Есть две константы: JUSTIFICATION FULL И JUSTIFICATION NONE
POSTURE	Наклон шрифта. Объект класса Float. Есть две константы: POSTURE_OBLIQUE И POSTURE_REGULAR
RUNJHRECTION	Просмотр текста: RUN DIRECTION LTR — слева направо, RUN DIRECTION RTL — справа налево
SIZE	Размер шрифта в пунктах. Объект класса Float
STRIKETHROUGH	Перечеркивание шрифта. Задается константой STRIKETHROUGH ON, по умолчанию перечеркивания нет
SUPERSCRIPT	Подстрочные или надстрочные индексы. Константы: SUPERSCRIPT_NONE, SUPERSCRIPT_SUB, SUPERSCRIPT_SUPER
SWAP COLORS	Замена местами цвета текста и цвета фона. Константа SWAP COLORS ON, по умолчанию замены нет
TRANSFORM	Преобразование шрифта. Объект класса AffineTransform
UNDERLINE	Подчеркивание шрифта. Константы: UNDERLINE_ON, UNDERLINE_LOW_DASHED, UNDERLINE_LOW_DOTTED, UNDERLINE LOW GRAY, UNDERLINE LOW ONE PIXEL, UNDERLINE_LOW_TWO_PIXEL

WEIGHT	Толщина шрифта. Константы: WEIGHT ULTRA LIGHT, WEIGHT _ EXTRA_LIGHT, WEIGHT _ LIGHT, WEIGHT _ DEMILIGHT, WEIGHT _ REGULAR, WEIGHT _ SEMIBOLD, WEIGHT MEDIUM, WEIGHT DEMIBOLD, WEIGHT _ BOLD, WEIGHT HEAVY, WEIGHT _ EXTRABOLD, WEIGHT _ ULTRABOLD
WIDTH	Ширина шрифта. Константы: WIDTH CONDENSED,WIDTH SEMI CONDENSED, WIDTH REGULAR, WIDTH_SEMI_EXTENDED, WIDTH_EXTENDED

К сожалению, не все шрифты позволяют задать все атрибуты. Посмотреть список допустимых атрибутов для данного шрифта можно методом getAvailableAttributes() класса Font.

В классе Font есть конструктор Font(Map attributes), которым можно сразу задать нужные атрибуты создаваемому шрифту. Это требует предварительной записи атрибутов в специально созданный для этой цели объект класса, реализующего интерфейс Map класса HashMap, WeakHashMap или Hashtable. Например:

```
HashMap hm = new HashMap ();
hm.put(TextAttribute.SIZE, new Float(60.Of));
hm.put(TextAttribute.POSTURE, TextAttribute.POSTUREJDBLIQUE);
Font f = new Font(hm);
```

Текст в Java 2D обладает контекстом - объектом класса FontRenderContext, хранящим всю информацию, необходимую для вывода текста. Получить его можно методом getFontRenderContext() класса Graphics2D.

Вся информация о тексте, в том числе и об его контексте, собирается в объекте класса TextLayout. Этот класс в Java 2D заменяет класс FontMetrics.

В конструкторе класса TextLayout задается текст, шрифт и контекст. Начало метода paint() со всеми этими определениями может выглядеть так:

```
public void paint(Graphics gr) {
    Graphics2D g = (Graphics2D)gr;
    FontRenderContext frc = g.getFontRenderContext();
    Font f = new Font("Serif", Font.BOLD, 15);
    String s = "Какой-то текст";
    TextLayout tl = new TextLayout(s, f, frc) ;    // Продолжение метода
}
```

В классе TextLayout есть не только более 20 методов getXxx(), позволяющих узнать различные сведения о тексте, его шрифте и контексте, но и метод

draw(Graphics2D g, float x, float y)

вычерчивающий содержимое объекта класса TextLayout в графический области g, начиная с точки (x, y).

Еще один интересный метод

getOutline(AffineTransform at)

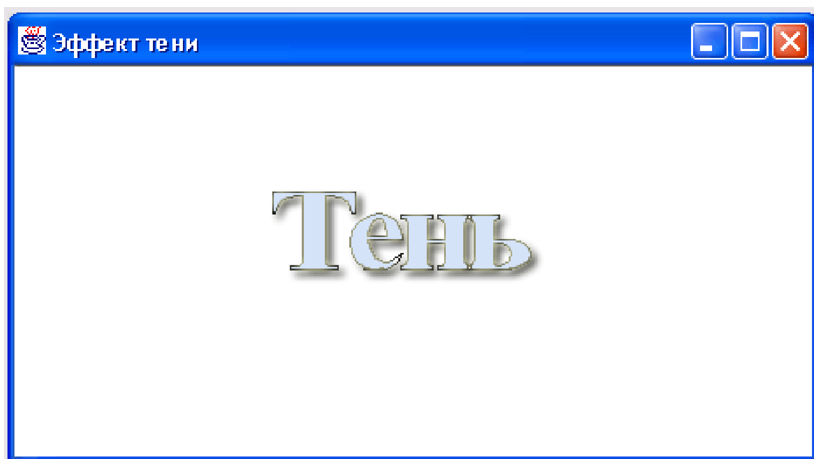
возвращает контур шрифта в виде объекта shape. Этот контур можно затем заполнить по какому-нибудь образцу или вывести только контур, как показано в листинге.

Листинг. Вывод текста средствами Java 2D

```
import java.awt.*;
import j ava.awt.font.*;
import j ava.awt.geom.*;
import java.awt.event.*
Class StillText extends Frame {
    StillText(String s) {
        super(s);
        setSize(400, 200);
        setVisible(true);
        addWindowListener (
            new WindowAdapter()  {
                public void windowClosing(WindowEvent ev) {
                    System.exit(0) ;
                }
            }
        );
    }
}
public void paint(Graphics gr) {
    Graphics2D g = (Graphics2D)gr;
    int w = getSize().width, h = getSize().height;
    FontRenderContext frc = g.getFontRenderContext();
    String s = "Тень";
    Font f = new Font("Serif", Font.BOLD, h/3);
    TextLayout tl = new TextLayout(s, f, frc);
    AffineTransform at = new AffineTransform();
    at.setToTranslation(w/2-tl.getBounds().getwidth()/2, h/2);
```

```
        Shape sh = tl.getOutline(at);
        g.draw(sh);
        AffineTransform atsh =
        new AffineTransform(1, 0.0, 1.5, -1, 0.0, 0.0);
        g.transform(at);
        g.transform(atsh);
        Font df = f.deriveFont(atsh);
        TextLayout dtl = new TextLayout(s, df, frc);
        Shape sh2 = dtl.getOutline(atsh);
        g.fill(sh2); } public static void main(Stnng[] args) {
            new StillText(" Эффект тени");
        }
}
```

Вывод этой программы.



Еще одна возможность создать текст с атрибутами - определить объект класса AttributedString из пакета j ava. text. Конструктор этого класса

AttributedString(String text, Map attributes)

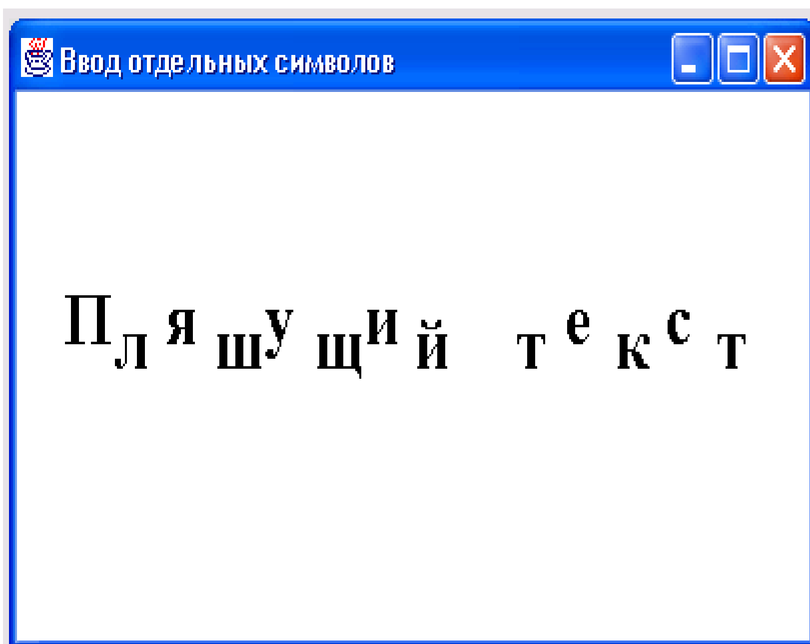
задает сразу и текст, и его атрибуты. Затем можно добавить или изменить характеристики текста одним их трех методов addAttibute().

Если текст занимает несколько строк, то встает вопрос его форматирования. Для этого вместо класса TextLayout используется класс LineBreakMeasurer, методы которого позволяют отформатировать абзац. Для каждого сегмента

текста можно получить экземпляр класса TextLayout и вывести текст, используя его атрибуты.

Для редактирования текста необходимо отслеживать курсором (caret) текущую позицию в тексте. Это осуществляется методами класса TextHitinfo, а методы класса TextLayout позволяют получить позицию курсора, выделить блок текста" и подсветить его.

Наконец, можно задать отдельные правила для вывода каждого символа текста. Для этого надо получить экземпляр класса Glyphvector методом createGiyphvector() класса Font, изменить позицию символа методом setciyphPosition(), задать преобразование символа, если это допустимо для данного шрифта, методом setciyphTransformo, и вывести измененный текст методом drawGiyphVector() класса Graphics2D. Пример:.



16.3.7. Методы улучшения визуализации

Визуализацию (rendering) созданной графики можно усовершенствовать, установив один из методов (hint) улучшения одним из методов класса Graphics2D:

- setRenderingHints(RenderingHints.Key key, Object value)
- setRenderingHints(Map hints)

Ключи - методы улучшения - и их значения задаются константами класса RenderingHints, перечисленными в таблице.

Метод	Значение
KEY_ANTIALIASING	Размывание крайних пикселов линий для гладкости изображения; три значения, задаваемые константами VALUE ANTIALIAS DEFAULT, VALUE ANTIALIAS ON, VALUE~ANTIALIAS OFF
KEY_TEXT_ANTTALIASING	То же для текста. Константы: VALUE TEXT ANTIALIASING_DEFAULT, VALUE TEXT ANTIALIASING ON, VALUE_TEXT_ANTIALIASING_OFF
KEY RENDERING	Три типа визуализации. Константы: VALUE RENDER SPEED, VALUE RENDER QUALITY, VALUE RENDER DEFAULT
KEY COLOR RENDERING	То же для цвета. Константы: VALUE COLOR RENDER_SPEED, VALUE COLOR RENDER QUALITY, VALUE COLOR RENDER DEFAULT
KEY ALPHA INTERPOLATION	Плавное сопряжение линий. Константы: VALUE ALPHA INTERPOLATION_SPEED, VALUE ALPHA INTERPOLATION QUALITY, VALUE_ALPHA_INTERPOLATION_DEFAULT
KEY_INTERPOLATION	Способы сопряжения. Константы: VALUE_INTERPOLATION_BILINEAR, VALUE INTERPOLATION BICUBIC, VALUE INTERPOLATION_NEAREST_NEIGHBOR
KEY_DITHERING	Замена близких цветов. Константы: VALUE DITHER ENABLE, VALUE DITHER DISABLE, VALUE DITHER DEFAULT

Не все графические системы обеспечивают выполнение этих методов, поэтому задание указанных атрибутов не означает, что определяемые ими методы будут применяться на самом деле.

16.4. Библиотека AWT

16.4.1. Компонент и контейнер

Основное понятие графического интерфейса пользователя (ГИП) — компонент (component) графической системы. В русском языке это слово подразумевает просто составную часть, но в ГИП это понятие гораздо конкретнее. Оно означает отдельный, полностью определенный элемент, который можно использовать независимо от других элементов. Например, это поле ввода, кнопка, строка меню, полоса прокрутки, радиокнопка. Само окно приложения — тоже его компонент. Компоненты могут быть и невидимыми. Например, панель, объединяющая компоненты, тоже является компонентом.

В AWT компонентом считается объект класса Component или объект всякого класса, расширяющего класс Component. В классе Component собраны общие методы работы с любым компонентом ГИП. Этот класс - центр библиотеки AWT.

Каждый компонент перед выводом на экран помещается в контейнер (container). Контейнер "знает", как разместить компоненты на экране. В языке Java контейнер - это объект класса Container или всякого его расширения. Прямой наследник этого класса класс JComponent вершина иерархии многих классов библиотеки Swing.

Создав компонент - объект класса Component или его расширения, следует добавить его к предварительно созданному объекту класса container или его расширения одним из методов add().

Класс Container сам является невидимым компонентом, он расширяет класс Component. Таким образом, в контейнер наряду с компонентами можно помещать контейнеры, в которых находятся какие-то другие компоненты, достигая тем самым большой гибкости расположения компонентов.

Основное окно приложения, активно взаимодействующее с операционной системой, необходимо построить по правилам графической системы. Оно должно перемещаться по экрану, изменять размеры, реагировать на действия мыши и клавиатуры. В окне должны быть, как минимум, следующие стандартные компоненты.

- Строка заголовка (title bar), с левой стороны которой необходимо разместить кнопку контекстного меню, а с правой — кнопки сворачивания и разворачивания окна и кнопку закрытия приложения.
- Необязательная строка меню (menu bar) с выпадающими пунктами меню.
- Горизонтальная и вертикальная полосы прокрутки (scrollbars).

- Окно должно быть окружено рамкой (border), реагирующей на действия мыши.

Окно с этими компонентами в готовом виде описано в классе Frame. Чтобы создать окно, достаточно сделать свой класс расширением класса Frame, как показано в листинге. Всего несколько строк текста и окно готово.

```
import j ava.awt.*;
Class TooSimpleFrame extends Frame {
    public static void main(String[] args) {
        Frame fr = new TooSimpleFrame();
        fr.setSize(400, 150);
        fr.setVisible(true);
    }
}
```

Класс TooSimpleFrame обладает всеми свойствами класса Frame, являясь его расширением. В нем создается экземпляр окна fr, и устанавливаются размеры окна на экране - 400x150 пикселов - методом setSize(). Если не задать размер окна, то на экране появится окно минимального размера — только строка заголовка. Конечно, потом его можно растянуть с помощью мыши до любого размера.

Затем окно выводится на экран методом setVisibie(true). Дело в том, что, с точки зрения библиотеки AWT, создать окно значит выделить область оперативной памяти, заполненную нужными пикселами, а вывдит содержимое этой области на экран метод SetVisible(true).

Конечно, такое окно непригодно для работы. Не говоря уже о том, что у него нет заголовка и поэтому окно нельзя закрыть. Хотя его можно перемещать по экрану, менять размеры, сворачивать на панель задач и раскрывать, но команду завершения приложения мы не запрограммировали. Окно нельзя закрыть ни щелчком кнопки мыши на кнопке с крестиком в правом верхнем углу окна, ни комбинацией клавиш <Alt>+<F4>. Приходится завершать работу приложения средствами операционной системы, например, комбинацией клавиш <Ctrl>+<C>.

В листинге ниже к предыдущей программе добавлены заголовок окна и обращение к методу, позволяющему завершить приложение.

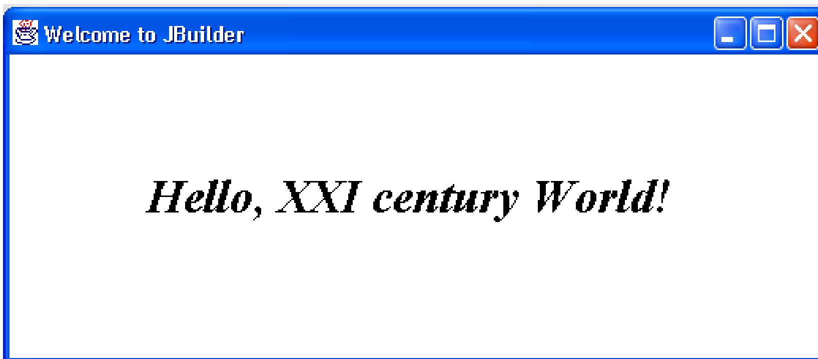
```
import java.awt.*;
import java.awt.event.*;
Class SimpleFrame extends Frame {
    SimpleFrame(String s) {
        super (s);
```

```
        setSize(400, 150);
        setVisible(true);
        addWindowListener (
            new WindowAdapter()  {
                public void windowClosing(WindowEvent ev) {
                    System.exit (0);
                }
            }
        );
    }
    public static void main(String[] args) {
        new SimpleFrame(" Моя программа");
    }
}
```

В программу добавлен конструктор класса SimpleFrame, обращающийся к конструктору своего суперкласса Frame, который записывает свой аргумент s в строку заголовка окна.

В конструктор перенесена установка размеров окна, вывод его на экран и добавлено обращение к методу addWindowListener(), реагирующему на действия с окном. В качестве аргумента этому методу передается экземпляр безымянного внутреннего класса, расширяющего класс WindowAdapter. Этот безымянный класс реализует метод windowClosing(), обрабатывающий попытку закрытия окна. Данная реализация очень проста — приложение завершается статическим методом exit() класса System. Окно при этом закрывается автоматически.

Итак, окно готово. Но оно пока пусто. Выведем в него, по традиции, приветствие "Hello, World!", правда, слегка измененное.



Листинг. Графическая программа с приветствием

```
import java.awt.*;
import java.awt.event.*;
Class HelloWorldFrame extends Frame {
    HelloWorldFrame(String s) {
        super(s);
    }
    public void paint(Graphics g) {
        g.setFont(new Font("Serif", Font.ITALIC | Font.BOLD, 30));
        g.drawstring("Hello, XXI century World!", 20, 100);
    }
    public static void main(String[] args) {
        Frame f = new HelloWorldFrame("Здравствуй, мир XXI века!");
        f.setSize(400, 150);
        f.setvisible(true);
        f.addWindowListener (
            new WindowAdapter()
            {
                public void windowciosing(WindowEvent ev)
                (
                    System.exit(0);
                }
            }
        );
    }
}
```

Для вывода текста мы переопределяем метод paint() класса Component. Класс Frame наследует этот метод с пустой реализацией.

Метод paint() получает в качестве аргумента экземпляр g класса Graphics, умеющего, в частности, выводить текст методом drawString(). В этом методе кроме текста мы указываем положение начала строки в окне - 20 пикселов от левого края и 100 пикселов сверху. Эта точка - левая нижняя точка первой буквы текста H.

Кроме того, мы установили новый шрифт "Serif" большего размера - 30 пунктов, полужирный, курсив.

16.4.2. Иерархия классов AWT

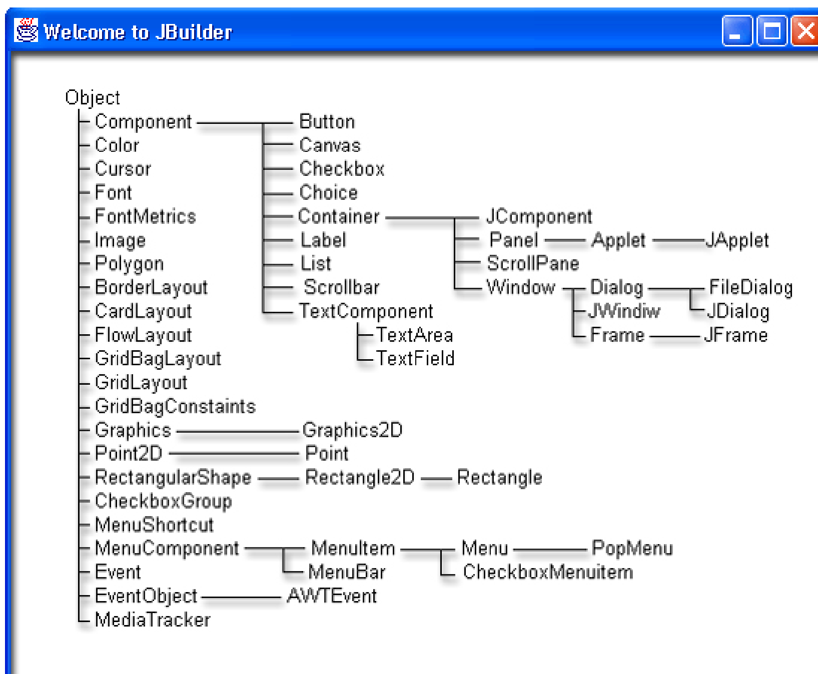
На рисунке показана иерархия основных классов AWT. Основу ее составляют готовые компоненты: Button, Canvas, Checkbox, Choice, Container, Label, List,

Scrollbar, TextArea, TextField, Menubar, Menu, PopupMenu, MenuItem, CheckboxMenuItem. Если этого набора не хватает, то от класса Canvas можно породить собственные "тяжелые" компоненты, а от класса Component - "легкие" компоненты.

Основные контейнеры - это классы Panel, ScrollPane, Window, Frame, Dialog, FileDialog. Свои "тяжелые" контейнеры можно породить от класса Panel, а "легкие" - от класса Container.

Целый набор классов помогает размещать компоненты, задавать цвет, шрифт, рисунки и изображения, реагировать на сигналы от мыши и клавиатуры.

На рисунке показаны и начальные классы иерархии библиотеки Swing - классы JComponent, JWindow, JFrame, JDialog, JApplet.



Как видите, библиотека графических классов AWT очень велика и детально проработана. Это многообразие классов только отражает многообразие задач построения ГИП. Стремление улучшить интерфейс безгранично. Оно приводит к созданию все новых библиотек классов и расширению существующих. Независимыми производителями создано уже много графических библиотек Java: KL Group, JBCL, и появляются все новые и новые библиотеки.

Графическая библиотека AWT предлагает более 20 готовых компонентов. Наиболее часто используются подклассы класса Component: классы Button, Canvas, Checkbox, Choice, Container, Label, List, Scrollbar, TextArea, TextField, Panel, ScrollPane, Window, Dialog, FileDialog, Frame.

Еще одна группа компонентов - это компоненты меню - классы Menuitem, MenuBar, Menu, PopupMenu, CheckboxMenuItem.

Начнем изучать эти компоненты от простых компонентов к сложным и от наиболее часто используемых к применяемым реже. Но сначала посмотрим на то общее, что есть во всех этих компонентах, на сам класс Component.

16.4.3. Класс Component

Класс Component - центр библиотеки AWT - очень велик и обладает большими возможностями. В нем 5 статических констант, определяющих размещение компонента внутри пространства, выделенного для компонента в содержащем его контейнере:

- BOTTOM_ALIGNMENT,
- CENTER_ALIGNMENT,
- LEFT_ALIGNMENT,
- RIGHT_ALIGNMENT,
- TOP_ALIGNMENT,

и около 100 методов.

Большинство методов - это методы доступа (Xxx – конкретизация):

- getXxx() - получить,
- isXxx() - проверить,
- setXxx() - установить.

Конструктор класса недоступен - он защищенный (protected), потому, что класс Component абстрактный, он не может использоваться сам по себе, применяются только его подклассы.

Компонент всегда занимает прямоугольную область со сторонами, параллельными сторонам экрана и в каждый момент времени имеет определенные размеры, измеряемые в пикселах, которые можно узнать методом getSize(), возвращающим объект класса Dimension, или целочисленными методами getHeight() и getWidth(), возвращающими высоту и ширину прямоугольника. Новый размер компонента можно установить из программы методами setSize(Dimension d) или setSize(int width, int height), если это допускает менеджер размещения контейнера, содержащего компонент.

У компонента есть предпочтительный размер, при котором компонент выглядит наиболее пропорционально. Его возвращает метод getPreferredSize() в виде объекта Dimension.

Компонент обладает минимальным и максимальным размерами. Их возвращают методы getMinimumSize() и getMaximumSize() в виде объекта Dimension.

В компоненте есть система координат. Ее начало - точка с координатами (0,0) - находится в левом верхнем углу компонента, ось Ох идет вправо, ось Оу - вниз, координатные точки расположены между пикселами.

В компоненте хранятся координаты его левого верхнего угла в системе координат объемлющего контейнера. Их можно узнать методами getLocation(), а изменить - методами setLocation(), переместив компонент в контейнере, если это позволит менеджер размещения компонентов.

Можно выяснить сразу и положение, и размер прямоугольной области компонента методом getBounds(), возвращающим объект класса Rectangle, и изменить разом и положение, и размер компонента методами setBounds(), если это позволит сделать менеджер размещения.

Компонент может быть недоступен для действий пользователя, тогда он выделяется на экране обычно светло-серым цветом. Доступность компонента можно проверить логическим методом isEnabied(), а изменить - методом setEnabled(Boolean enable).

Для многих компонентов определяется графический контекст - объект класса Graphics, который управляется методом paint(), который можно получить методом getGraphics().

В контексте есть текущий цвет и цвет фона - объекты класса Color. Цвет фона можно получить методом getBackground(), а изменить - методом setBackground(Color color).

В контексте есть шрифт - объект класса Font, возвращаемый методом getFont() и изменяемый методом setFont(Font font).

В компоненте определяется локаль - объект класса Locale. Его можно получить методом getLocale(), изменить - методом setLocale(Locale locale).

В компоненте существует курсор, показывающий положение мыши, - объект класса Cursor. Его можно получить методом getCursor(), изменяется форма курсора в "тяжелых" компонентах с помощью метода setCursor(Cursor cursor). Остановимся на этом классе подробнее.

Класс **Cursor**. Основа класса — статические константы, определяющие форму курсора:

- CROSSHAIR_CURSOR - курсор в виде креста, появляется обычно при поиске позиции для размещения какого-то элемента;
- DEFAULT_CURSOR - обычная форма курсора — стрелка влево вверх;
- HAND_CURSOR - "указующий перст", появляется обычно при выборе какого-то элемента списка;
- MOVE_CURSOR - крест со стрелками, возникает обычно при перемещении элемента;
- TEXT_CURSOR - вертикальная черта, появляется в текстовых полях;
- WAIT_CURSOR - изображение часов, появляется при ожидании.

Следующие курсоры появляются обычно при приближении к краю или углу компонента:

- E_RESIZE_CURSOR - стрелка вправо с упором;
- N_RESIZE_CURSOR - стрелка вверх с упором;
- NE_RESIZE_CURSOR - стрелка вправо вверх, упирающаяся в угол;
- NW_RESIZE_CURSOR - стрелка влево вверх, упирающаяся в угол;
- S_RESIZE_CURSOR - стрелка вниз с упором;
- SE_RESIZE_CURSOR - стрелка впрзво вниз, упирающаяся в угол;
- SW_RESIZE_CURSOR - стрелка влево вниз, упирающаяся в угол;
- W_RESIZE_CURSOR - стрелка влево с упором.

Перечисленные констзнты являются аргументом type в конструкторе класса Cursor(int type).

Вместо конструктора можно обратиться к статическому методу getPredefinedCursor(int type), создающему объект класса Cursor и возвращающему ссылку на него.

Получить курсор по умолчанию можно статическим методом getDefauitCursor(). Затем созданный курсор надо установить в компонент. Например, после выполнения:

```
Cursor curs = new Cursor(Cursor.WAIT_CURSOR);
someComp.setCursor(curs);
```

при появлении указателя мыши в компоненте someComp указатель примет вид часов.

Кроме этих предопределенных курсоров можно задать свою собственную форму курсора. Ее тип носит название CUSTOM_CURSOR. Сформировать свой курсор можно методом

```
createCustomCursor(Image cursor, Point hotspot, String name),
```

создающим объект класса cursor и возвращающим ссылку на него. Перед этим следует создать изображение курсора cursor — объект класса image. Аргумент name задает имя курсора, можно написать просто null. Аргумент hotspot задает точку фокуса курсора. Эта точка должна быть в пределах изображения курсора, точнее, в пределах, показываемых методом

getBestCursorSize(int desiredWidth, int desiredHeight),

возвращающим ссылку на объект класса Dimension. Аргументы метода означают желаемый размер курсора. Если графическая система не допускает создание курсоров, возвращается (0, 0). Этот метод показывает приблизительно размер того курсора, который создаст графическая система, например, (32, 32). Изображение cursor будет подогнано под этот размер, при этом возможны искажения.

В примере создается курсор в виде красного прямоугольного треугольника с катетами размером 32 пиксела и устанавливается в каком-то компоненте someComp.

```
Toolkit tk = Toolkit.getDefaultTooikit();
int colorMax = tk.getMaximumCursorColors();  // Наибольшее число цветов
Dimension d = tk.getBestCursorSize(50, 50);   // d — размер изображения
int w = d.width, h = d.height, k = 0;
Point p = new Point(0, 0);         // Фокус курсора будет в его верхнем левом углу
int[] pix = new int[w * h];           // Здесь будут пикселы изображения
for(int i = 0; i < w; i++)
for(int j = 0; j < h; j++)
if (j < i) pix[k++] = 0xFFFF0000;      // Левый нижний угол - красный
else pix[k++] = 0;                          // Правый верхний угол — прозрачный
    // Создается прямоугольное изображение размером (w, h),
    // заполненное массивом пикселов pix, с длиной строки w
Image im = createImage(new MemoryImageSource(w, h, pix, 0, w));
Cursor curs = tk.createCustomCursor(im, p, null);
someComp.setCursor(curs);
```

События.

- Событие ComponentEvent происходит при перемещении компонента, изменении его размера, удалении с экрана и появлении на экране.
- Событие FocusEvent возникает при получении или потере фокуса.
- Событие KeyEvent проявляется при каждом нажатии и отпускании клавиши, если компонент имеет фокус ввода.
- Событие MouseEvent происходит при манипуляциях мыши на компоненте.

Каждый компонент перед выводом на экран помещается в контейнер - подкласс класса Container. Класс Container - прямой подкласс класса Component, и наследует все его методы. Кроме них основу класса составляют методы добавления компонентов в контейнер:

- add(Component comp) - компонент comp добавляется в конец контейнера;
- add(Component comp, int index) - компонент comp добавляется в позицию index в контейнере, если index = -1, то компонент добавляется в конец контейнера;
- add(Component comp, Object constraints) - менеджеру размещения контейнера даются указания объектом constraints;
- add(String name. Component comp) — компонент получает имя name.

2 метода удаляют компоненты из контейнера:

- remove(Component comp) - удаляет компонент с именем comp;
- remove(int index) - удаляет компонент с индексом index в контейнере.

Один из компонентов в контейнере получает фокус ввода (input focus), на него направляется ввод с клавиатуры. Фокус можно переносить с одного компонента на другой клавишами <TaЬ> и <Shift>+<Tab>. Компонент может запросить фокус методом requestFocus() и передать фокус следующему компоненту методом transferFocus(). Компонент может проверить, имеет ли он фокус, своим логическим методом hasFocus(). Это методы класса Component.

Для облегчения размещения компонентов в контейнере определяется менеджер размещения (layout manager) — объект, реализующий интерфейс LayoutManager или его подинтерфейс LayoutManager2. Каждый менеджер размещает компоненты в каком-то своем порядке: один менеджер расставляет компоненты в таблицу, другой норовит растащить компоненты по сторонам, третий просто располагает их один за другим, как слова в тексте. Менеджер определяет смысл слов "добавить в конец контейнера" и "добавить в позицию index".

В контейнере в любой момент времени может быть установлен только один менеджер размещения. В каждом контейнере есть свой менеджер по умолчанию, установка другого менеджера производится методом

setLayout(LayoutManager manager)

Менеджеры размещения мы рассмотрим подробно в следующей главе. В данной главе мы будем размещать компоненты вручную, отключив менеджер по умолчанию методом setLayout (null).

События. Кроме событий Класса Component: ComponentEvent, FocusEvent, KeyEvent, MouseEvent при добавлении и удалении компонентов в контейнере происходит событие ContainerEvent.

16.4.4. Компонент Label

Компонент Label - это просто строка текста, оформленная как графический компонент для размещения в контейнере. Текст можно поменять только методом доступа setText(string text), но не вводом пользователя с клавиатуры или с помощью мыши.

Создается объект этого класса одним из трех конструкторов:

- Label() - пустой объект без текста;
- Label(String text) — объект с текстом text, который прижимается клевому краю компонента;
- Label(String text, int alignment) — объект с текстом text и определенным размещением в компоненте текста, задаваемого одной из трех констант: CENTER, LEFT, RIGHT.

Размещение можно изменить методом доступа setAlignment(int alignment).

Остальные методы, кроме методов, унаследованных от класса component, позволяют получить текст getText () и размещение getAlignment ().

В классе Label происходят события классов Component: ComponentEvent, FocusEvent, KeyEvent, MouseEvent.

16.4.5. Компонент Button

Компонент Button - это кнопка стандартного для данной графической системы вида с надписью, умеющая реагировать на щелчок кнопки мыши - при нажатии она "вдавливается" в плоскость контейнера, при отпускании - становится "выпуклой".

Два конструктора Button() и Button(String label) создают кнопку без надписи и с надписью label соответственно.

Методы доступа getLabel() и setLabel(String label) позволяют получить и изменить надпись на кнопке.

Главная функция кнопки - реагировать на щелчки мыши, и прочие методы класса обрабатывают эти действия.

Кроме событий класса Component: ComponentEvent, FocusEvent, KeyEvent, MouseEvent, при воздействии на кнопку происходит событие ActionEvent.

16.4.6. Компонент Checkbox

Компонент Checkbox — это надпись справа от небольшого квадратика, в котором в некоторых графических системах появляется галочка после щелчка кнопкой мыши — компонент переходит в состояние (state) on. После следующего щелчка галочка пропадает — это состояние off. В других графических системах состояние on отмечается "вдавливанием" квадратика. В компоненте checkbox состояния on/off отмечаются логическими значениями true/false соответственно.

3 конструктора Checkbox(), Checkbox(String label), Checkbox(String label,

Boolean state) создают компонент без надписи, с надписью label в состоянии off, и в заданном состоянии state.

Методы доступа getLabel(), setLabel(String label), getState(), setSate(Boolean state) возвращают и изменяют эти параметры компонента.

Компоненты Checkbox удобны для быстрого и наглядного выбора из списка, целиком расположенного на экране.

В классе Checkbox происходят события класса Component: ComponentEvent, FocusEvent, KeyEvent, MouseEvent, а при изменении состояния кнопки возникает событие ItemEvent.

16.4.7. Компонент CheckboxGroup

В библиотеке AWT радиокнопки не образуют отдельный компонент. Вместо этого несколько компонентов Checkbox объединяются в группу с помощью объекта класса CheckboxGroup.

Класс CheckboxGroup очень мал, поскольку его задача — просто дать общее имя всем объектам checkbox, образующим одну группу. В него входит один конструктор по умолчанию CheckboxGroup() и 2 метода доступа:

- getSelectedCheckbox(), возвращающий выбранный объект Checkbox;
- setSelectedCheckbox (Checkbox box), задающий выбор.

Чтобы организовать группу радиокнопок, надо сначала сформировать объект класса CheckboxGroup, а затем создавать в нем кнопки конструкторами

- Checkbox(String label, CheckboxGroup group, Boolean state)
- Checkbox(String label, Boolean state, CheckboxGroup group)

Эти конструкторы идентичны, просто при записи конструктора можно не думать о порядке следования его аргументов.

Только одна радиокнопка в группе может иметь состояние state = true.

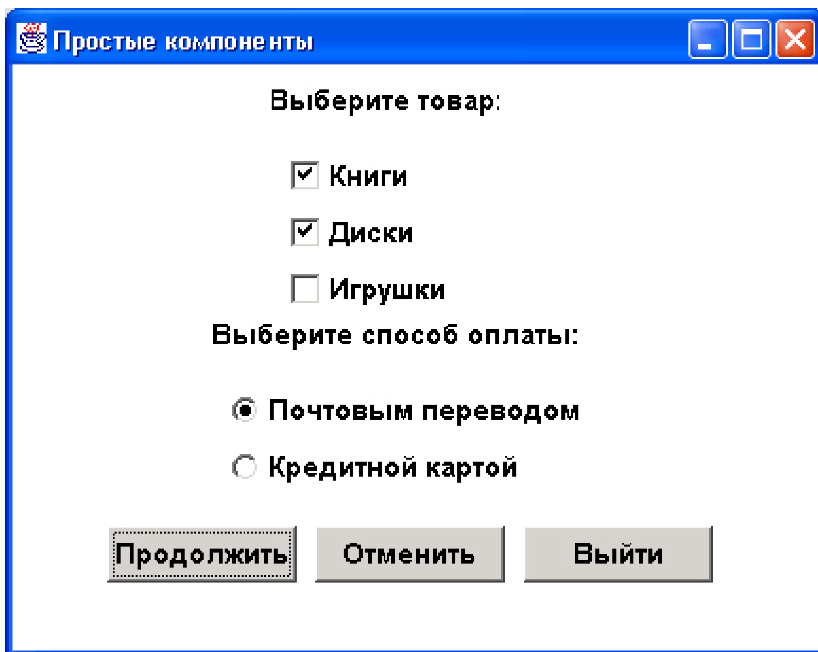
В листинге приведена программа, помещающая в контейнер Frame две метки Label сверху, под ними слева три объекта checkbox, справа — группу радиокнопок. Внизу — три кнопки Button.

Листинг. Размещение компонентов

```
import java.awt.*;
import j ava.awt.event.*;
Class SimpleComp extends Frame {
    SimpleComp(String s)  {
        super(s);
        setLayout(null);
        Font f = new Font("Serif", Font.BOLD, 15);
        setFont(f);
        Label L1 = new Label("Выберите товар:", Labe1.CENTER);
        L1.setBounds(0, 50, 120, 30); add(L1);
        Label L2 = new Label("Выберите способ оплаты:");
        L2.setBounds(160, 50, 200, 30); add(L2);
        Checkbox Ch1 = new Checkbox("Книги");
        Ch1.setBounds(20, 90, 100, 30); add(Ch1);
        Checkbox Ch2 = new Checkbox("Диски");
        Ch2.setBounds(20, 120, 100, 30); add(Ch2);
        Checkbox Ch3 = new Checkbox("Игрушки");
        Ch3.setBounds(20, 150, 100, 30); add(Ch3);
        CheckboxGroup grp = new CheckboxGroup();
        Checkbox Chg1 = new Checkbox("Почтовым переводом", grp, true);
        Chg1.setBounds{170, 90, 200, 30); add(Chg1);
        Checkbox chg2 = new Checkbox{"Кредитной картой", grp, false);
        chg2.setBounds(170, 120, 200, 30); add(chg2);
        Button B1 = new Button("Продолжить");
        B1.setBounds( 30, 220, 100, 30); add(B1));
        Button B2 = new Button("Отменить");
        B2.setBounds(140, 220, 100, 30); add(B2);
        Button B3 = new Button("Выйти");
        B3.setBounds(250, 220, 100, 30); add(B3);
        setSize(400, 300);
        setVisible(true);
    }
    public static void main(String[] args) {
        Frame f = new SimpleComp (" Простые компоненты");
        f.addWindowListener (
            new WindowAdapter()
```

```
            {
                public void windowClosing(WindowEvent ev) {
                    System.exit(0);
                }
            }
        );
    }
}
```

Это результат



Заметьте, что каждый создаваемый компонент следует заносить в контейнер, в данном случае Frame, методом add(). Левый верхний угол компонента помещается в точку контейнера с координатами, указанными первыми двумя аргументами метода setBounds(). Размер компонента задается последними двумя параметрами этого метода.

Если нет необходимости отображать весь список на экране, то вместо группы радиокнопок можно создать раскрывающийся список — объект класса Choice.

16.4.8. Компонент Choice

Компонент Choice - это раскрывающийся список, один, выбранный, пункт (item) которого виден в поле, а другие появляются при щелчке кнопкой мыши на небольшой кнопке справа от поля компонента.

Вначале конструктором Choice() создается пустой список.

Затем методом add(String text) в список добавляются новые пункты с текстом text. Они располагаются в порядке написания методов add() и нумеруются от нуля.

Вставить новый пункт в нужное место можно методом insert(String text, int position).

Выбор пункта можно произвести из программы методом select(String text) или select(int position).

Удалить один пункт из списка можно методом remove(String text) или remove (int position), а все пункты сразу - методом removeAll().

Число пунктов в списке можно узнать методом getItemCount().

Выяснить, какой пункт находится в позиции pos можно методом getItem(int pos), возвращающим строку.

Наконец, определение выбранного пункта производится методом getSelectedIndex(), возвращающим позицию этого пункта, или методом getSeiectedItem(), возвращающим выделенную строку.

В классе Choice происходят события класса Component: ComponentEvent, FocusEvent, KeyEvent, MouseEvent, а при выборе пункта возникает событие ItemEvent.

Если надо показать на экране несколько пунктов списка, то создайте объект класса List.

16.4.9. Компонент List

Компонент List - это список с полосой прокрутки, в котором можно выделить один или несколько пунктов. Количество видимых на экране пунктов определяется конструктором списка и размером компонента.

В классе 3 конструктора:

- List() - создает пустой список с 4 видимыми пунктами;
- List (int rows) - создает пустой список с rows видимыми пунктами;

- List (int rows, Boolean multiple) - создает пустой список, в котором можно отметить несколько пунктов, если multiple = true.

После создания объекта в список добавляются пункты с текстом item методами:

- метод add(String item) - добавляет новый пункт в конец списка;
- метод add(String item, int position) - добавляет новый пункт в позицию position.

Позиции нумеруются по порядку, начиная с нуля.

Удалить пункт можно методами remove(String item), remove(int position), removeAll().

Метод repiaceItem(String newitem, int pos) позволяет заменить текст пункта в позиции pos.

Количество пунктов в списке возвращает метод getItemCount().

Выделенный пункт можно получить методом getSelectedItem(), а его позицию - методом getSelectedIndex().

Если список позволяет осуществить множественный выбор, то выделенные пункты в виде массива типа string[] можно получить методом getSelectedItems(), позиции выделенных пунктов в виде массива типа int[] - методом getSelectedIndexes().

Кроме этих необходимых методов класс List содержит множество других, позволяющих манипулировать пунктами списка и получать его характеристики.

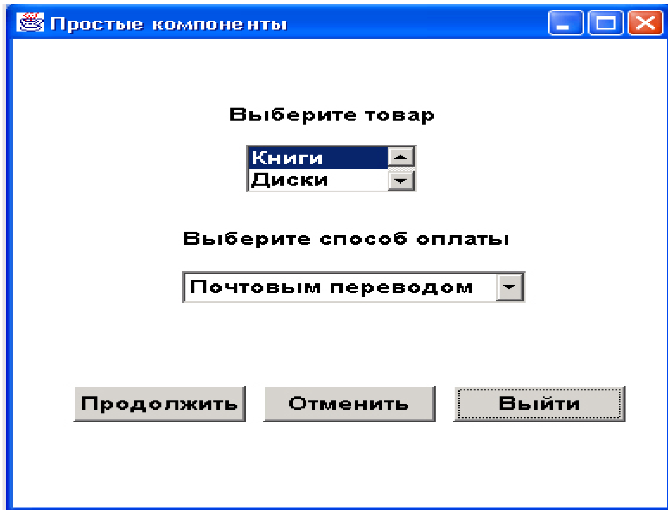
Кроме событий класса Component: ComponentEvent, FocusEvent, KeyEvent, MouseEvent, при двойном щелчке кнопкой мыши на выбранном пункте происходит событие ActionEvent.

Листинг. Использование списков

```
import java.awt.*;
import java.awt.event.*;
Class ListTest extends Frame {
    ListTest(String s) {
        super(s);
        setLayout(null);
        setFont(new Font("Serif", Font.BOLD, 15));
        Label L1 = new Label("Выберите товар:", Label.CENTER);
        L1.setBoundsdO, 50, 120, 30); add (L1);
        Label L2 = new Label("Выберите способ оплаты:");
```

```
        L2.setBounds(170, 50, 200, 30); add(L2);
        List L = new List(2, true);
        L.add("Книги");
        L.add("Диски");
        L.add("Игрушки");
        L.setBounds(20, 90, 100, 40); add(L);
        Choice Ch = new Choice();
        Ch.add("Почтовым переводом");
        Ch.add("Кредитной картой");
        Ch.setBounds(17O, 90, 200,30); add(Ch);
        Button B1 = new Button("Продолжить");
        B1.setBounds( 30, 150, 100, 30); add(B1);
        Button B2 = new Button("Отменить");
        B2.setBounds(140, 150, 100, 30); add(B2);
        Button B3 = new Button("Выйти");
        B3.setBounds(250, 150, 100, 30); add(B3);
        setSize(400, 200); setVisible(true);
    }
    public static void main(StringH args) {
        Frame f = new ListTest(" Простые компоненты");
        f.addWindowListener (
            new WindowAdapter()  {
                public void windowClosing(WindowEvent ev) {
                    System.exit(0);
                }
            }
        );
    }
}
```

Это результат



16.4.10. Класс TextComponent

В классе TextComponent нет конструктора, этот класс не используется самостоятельно.

Основной метод класса - метод getText() - возвращает текст, находящийся в поле ввода, в виде строки String.

Поле ввода может быть нередактируемым, в этом состоянии текст в поле нельзя изменить с клавиатуры или мышью. Узнать состояние поля можно логическим методом isEditable(), изменить значения в нем - методом setEditable(Boolean editable).

Текст, находящийся в поле, хранится как объект класса string, поэтому у каждого символа есть индекс (у первого — индекс 0). Индекс используется для определения позиции курсора (caret) методом getCaretPosition(), для установки позиции курсора методом setCaretPositionf(int ind) и для выделения текста.

Текст выделяется, как обычно, мышью или клавишами со стрелками при нажатой клавише <Shift>, но можно выделить его из программы методом select(int begin, int end). При этом помечается текст от символа с индексом begin включительно, до символа с индексом end исключительно.

Весь текст выделяет метод selectAll(). Можно отметить начало выделения методом setSeiectionStart(int ind) и конец выделения методом setSelectionEnd(int ind).

Важнее все-таки не задать, а получить выделенный текст. Его возвращает метод getSeiectedText(), а начальный и конечный индекс выделения возвращают методы getSelectionStart() и getSelectionEnd().

Кроме событий класса Component: ComponentEvent, FocusEvent, KeyEvent, MouseEvent, при изменении текста пользователем происходит событие TextEvent.

16.4.11. Компонент TextField

Компонент TextField - это поле для ввода одной строки текста. Ширина поля измеряется в колонках (column). Ширина колонки - это средняя ширина символа в шрифте, которым вводится текст. Нажатие клавиши <Enter> заканчивает ввод и служит сигналом к началу обработки введенного текста, т.е. при этом происходит событие ActionEvent.

В классе 4 конструктора:

- TextField() - создает пустое поле шириной в одну колонку;
- TextField(int columns) - создает пустое поле с числом колонок columns;
- TextField(String text) - создает поле с текстом text;
- TextField(String text, int columns) - создает поле с текстом text и числом колонок columns.

К методам, унаследованным от класса TextComponent, добавляются еще методы getColumns() и setColumns(int col).

Интересная разновидность поля ввода - поле для ввода пароля. В таком поле вместо вводимых символов появляется какой-нибудь особый эхо-символ, чаще всего звездочка, чтобы пароль никто не подсмотрел через плечо.

Данное поле ввода получается выполнением метода setEchoCnar(char echo). Аргумент echo - это символ, который будет появляться в поле. Проверить, установлен ли эхо-символ, можно логическим методом echoCharIsSet(), получить эхо-символ - методом getEchoChar().

Чтобы вернуть поле ввода в обычное состояние, достаточно выполнить метод setEchoChar(0).

Кроме событий класса Component: ComponentEvent, FocusEvent, KeyEvent, MouseEvent, при изменении текста пользователем происходит событие TextEvent, а при нажатии на клавишу <Enter> — событие ActionEvent.

16.4.12. Компонент TextArea

Компонент TextArea - это область ввода с произвольным числом строк. Нажатие клавиши <Enter> просто переводит курсор в начало следующей строки. В области ввода могут быть установлены линейки прокрутки, одна или обе.

Основной конструктор класса

TextArea(String text, int rows, int columns, int scrollbars)

создает область ввода с текстом text, числом видимых строк rows, числом колонок columns, и заданием полос.прокрутки scrollbars одной из 4 констант:

- SCROLLBARS_NONE,
- SCROLLBARS_HORIZONTAL_ONLY,
- SCROLLBARS_VERTICAL_ONLY,
- SCROLLBARS_BOTH.

Остальные конструкторы задают некоторые параметры по умолчанию:

- TextArea(String text, int rows, int columns) - присутствуют обе полосы прокрутки ;
- TextArea(int rows, int columns) - в поле пустая строка ;
- TextArea(string text) - размеры устанавливает контейнер;
- TextArea() - конструктор по умолчанию.

Среди методов класса TextArea наиболее важны методы:

- append (string text), добавляющий текст text в конец уже введенного текста;
- insert (string text, int pos), вставляющий текст в указанную позицию pos;
- replaceRange (String text, int begin, int end), удаляющий текст начиная с индекса begin включительно по end исключительно, и помещающий вместо него текст text.

Другие методы позволяют изменить и получить количество видимых строк.

Кроме Событий класса Component: ComponentEvent, FocusEvent, KeyEvent, MouseEvent, при изменении текста пользователем происходит событие TextEvent.

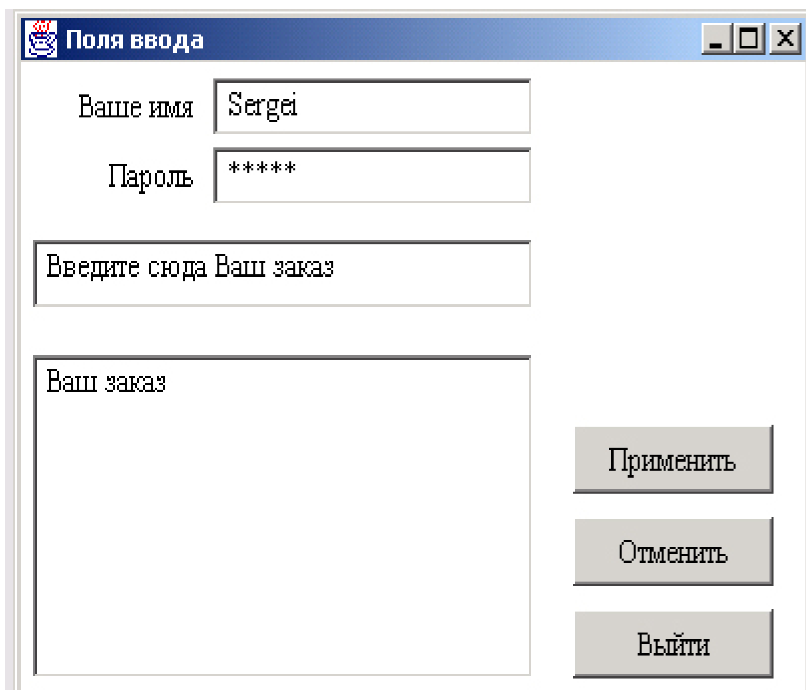
В листинге создаются три поля: TF1, TF2, TF3 для ввода имени пользователя, его пароля и заказа, и нередактируемая область ввода, в которой накапливается заказ. В поле ввода пароля TF2 появляется эхо-символ *.

Листинг. Поля ввода

```
import j ava.awt.*;
```

```
import j ava.awt.event.*;
Class TextTest extends Frame {
    TextTest(String s) {
        super(s);
        setLayout(null);
        setFont(new Font("Serif", Font.PLAIN, 14));
        Label L1 = new Label("Ваше имя:", Label.RIGHT);
        L1.setBounds(20, 30, 70, 25); add(L1);
        Label L2 = new Label("Пароль:", Label.RIGHT);
        L2.setBounds(20, 60, 70, 25); add(L2);
        TextField TF1 = new TextField(30) ;
        TF1.setBounds(100, 30, 160, 25); add(TF1);
        TextField TF2 = new TextField(30);
        TF2.setBounds(100, 60, 160, 25); add(TF2); TF2.setEchoChar('*');
        TextField TF3 = new TextField("Введите сюда Ваш заказ", 30);
        TF3.setBounds(10, 100, 250, 30); add(TF3);
        TextArea TA = new TextArea("Ваш заказ:", 5, 5);
        TextArea.SCROLLBARS_NONE);
        TA.setEditable(false);
        TA.setBounds(10, 150, 250, 140); add(TA);
        Button B1 = new Button("Применить");
        B1.setBounds(280, 180, 100, 30); add(B1);
        Button B2 = new Button("Отменить");
        B2.setBounds(280, 220, 100, 30); add(B2);
        Button B3 = new Button("Выйти");
        B3.setBounds(280, 260, 100, 30); add(B3);
        setSize(400, 300); setVisible(true);
        public static void main(String[] args) {
            Frame F= new TextTest(" Поля ввода");
            F.addWindowListener (
                new WindowAdapter()  {
                    public void windowClosing(WindowEvent ev) {
                        System.exit(0);
                    }
                }
            );
        }
    }
}
```

Это результат



16.4.13. Компонент Table

16.4.14. Компонент Scrollbar

Компонент Scrollbar - это полоса прокрутки, но в библиотеке AWT класс Scrollbar используется еще и для организации ползунка (slider). Объект может располагаться горизонтально или вертикально, обычно полосы прокрутки размещают внизу и справа.

Каждая полоса прокрутки охватывает некоторый диапазон значений и хранит текущее значение из этого диапазона. В линейке прокрутки есть пять элементов управления для перемещения по диапазону. Две стрелки на концах линейки вызывают перемещение на одну единицу (unit) в соответствующем направлении при щелчке на стрелке кнопкой мыши. Положение движка или бегунка (bubble, thumb) показывает текущее значение из диапазона и может его изменять при перемещении бегунка с помощью мыши. Два промежутка между движком и стрелками позволяют переместиться на один блок (block) щелчком кнопки мыши.

Смысл понятий "единица" и "блок" зависит от объекта, с которым работает полоса прокрутки. Например, для вертикальной полосы прокрутки при просмотре текста это может быть строка и страница или строка и абзац.

Методы работы с данным компонентом описаны в интерфейсе Adjustable, который реализован классом scroiibar.

В классе Scroiibar 3 конструктора:

- Scrollbar() — создает вертикальную полосу прокрутки с диапазоном 0—100, текущим значением 0 и блоком 10 единиц;
- Scrollbar(int orientation) — ориентация orientation задается одной из двух констант HORIZONTAL или VERTICAL ;
- Scrollbar(int orientation, int value, int visible, int min, int max) — задает, кроме ориентации, еще начальное значение value, размер блока visible, диапазон значений min—max.

Аргумент visible определяет еще и длину движка — она устанавливается пропорционально диапазону значений и длине полосы прокрутки. Например, конструктор по умолчанию задаст длину движка равной 0,1 длины полосы прокрутки.

Основной метод класса - getValue() - возвращает значение текущего положения движка на полосе прокрутки. Остальные методы доступа позволяют узнать и изменить характеристики объекта.

Кроме событий класса Component: ComponentEvent, FocusEvent, KeyEvent, MouseEvent, при изменении значения пользователем происходит событие Adj ustmentEvent.

В листинге создаются 3 вертикальные полосы прокрутки — красная, зеленая и синяя, позволяющие выбрать какое-нибудь значение соответствующего цвета в диапазоне 0—255, с начальным значением 127. Кроме них создается область, заполняемая получившимся цветом, и две кнопки. Линейки прокрутки, их заголовок и масштабные метки помещены в отдельный контейнер р типа Panel.

Листинг. Линейки прокрутки для выбора цвета

```
import j ava.awt.*;
import j ava.awt.event.*;
Class ScrollTest extends Frame {
    Scroiibar sbRed = new Scroiibar(Scroiibar.VERTICAL, 127, 10, 0, 255);
    Scroiibar sbGreen = new Scroiibar(Scroiibar.VERTICAL, 127, 10, 0, 255);
    Scroiibar sbBlue = new Scroiibar(Scroiibar.VERTICAL, 127, 10, 0, 255);
    Color mixedColor = new Color(127, 127, 127);
```

```
Button B1= new Button("Применить");
Button B2 = new Button("Отменить");
ScrollTest(String s) {
    super(s);
    setLayout(null);
    setFont(new Font("Serif", Font.BOLD, 15));
    Panel p = new Panel();
    p.setLayout(null);
    p.setBounds(10,50, 150, 260); add(p);
    Label Lc = new Label("Подберите цвет");
    Lc.setBounds(20, 0, 120, 30); p.add(Lc);
    Label Lmin = new Label("0", Label.RIGHT);
    Lmin.setBounds(0, 30, 30, 30); p.add(Lmin);
    Label Lmiddle = new Label("127", Label.RIGHT);
    Lmiddle.setBounds(0, 120, 30, 30); p.add(Lmiddle);
    Label Lmax = new Label("255", Label.RIGHT);
    Lmax.setBounds(0, 200, 30, 30); p.add(Lmax);
    sbRed.setBackground(Color.red);
    sbRed.setBounds(40, 30, 20, 200); p.add(sbRed);
    sbGreen.setBackground(Color.green);
    sbGreen.setBounds(70, 30, 20, 200); p.add(sbGreen);
    sbBlue.setBackground(Color.blue);
    sbBlue.setBounds(100, 30, 20, 200); p.add{sbBlue);
    Label Lp = new Label("Образец:");
    Lp.setBounds(250, 50, 120, 30); add Lp);
    Label Lm = new Label();
    Lm.setBackground(new Color(127, 127, 127));
    Lm.setBounds(220, 80, 120, 80); add(lm);
    B1.setBounds(240, 200, 100, 30); add(B1);
    B2.setBounds(240, 240, 100, 30); addB2);
    setSize(400, 300); setVisible(true);
}
public static void main(String[] args) {
    Frame f = new ScrollTestC' Выбор цвета");
    f.addWindowListener (
        new WindowAdapter()  {
            public void windowClosing(WindowEvent ev) {
                System.exit(0);
            }
        }
    );
}
```

}

В листинге использован контейнер Panel.

16.4.15. Контейнер Panel

Контейнер Panel - это невидимый компонент графического интерфейса, служащий для объединения нескольких других компонентов в один объект типа Panel.

Класс Panel очень прост, но важен. В нем всего 2 конструктора:

- Panel() - создает контейнер с менеджером размещения по умолчанию FlowLayoutJ
- Panel(LayoutManager layout) - создает контейнер с указанным менеджером размещения компонентов layout.

После создания контейнера в него добавляются компоненты унаследованным методом add():

```
Panel p = new Panel();
p.add(compl);
p.add(comp2);
```

и т.д. Размещает компоненты в контейнере его менеджер размещения.

Контейнер Panel используется очень часто. Он удобен для создания группы компонентов.

В листинге 3 полосы прокрутки вместе с заголовком "Подберите цвет" и масштабными метками 0, 127 и 255 образуют естественную группу. Если мы захотим переместить ее в другое место окна, нам придется переносить каждый из семи компонентов, входящих в указанную группу. При этом придется следить за тем, чтобы их взаимное положение не изменилось. Вместо этого мы создали панель р и разместили на ней все 7 элементов. Метод setBounds() каждого из рассматриваемых компонентов указывает в данном случае положение и размер компонента в системе координат панели р, а не окна Frame. В окно мы поместили сразу целую панель, а не ее отдельные компоненты.

Теперь для перемещения всей группы компонентов достаточно переместить панель, и находящиеся на ней объекты автоматически переместятся вместе с ней, не изменив своего взаимного положения.

16.4.16. Контейнер ScrollPane

Контейнер ScrollPane может содержать только 1 компонент, но зато такой, который не помещается целиком в окне. Контейнер обеспечивает средства прокрутки для просмотра большого компонента. В контейнере можно установить полосы прокрутки либо постоянно, константой SCROLLBARS_ALWAYS, либо так, чтобы они появлялись только при необходимости (если компонент действительно не помещается в окно) константой SCROLLBARS_AS_NEEDED.

Если полосы прокрутки не установлены (константа SCROLLBARS_NEVER), то перемещение компонента для просмотра нужно обеспечить из программы одним из методов setScrollPosition().

В классе 2 конструктора:

- ScrollPane() — создает контейнер, в котором полосы прокрутки появляются по необходимости;
- ScrollPane(int scroiibars) — создает контейнер, в котором появление линеек прокрутки задается одной из трех указанных выше констант.

Крнструкторы создают контейнер размером 100x100 пикселов, в дальнейшем можно изменить размер унаследованным методом-setsizet int width, int height).

Ограничение, заключающееся в том, что scroiiPane может содержать только один компонент, легко обходится. Всегда можно сделать этим единственным компонентом объект класса Panel, разместив на панели что угодно.

Среди методов класса интересны те, что позволяют прокручивать компонент В ScroiiPane:

- методы getHAdjustable() и getVAdjustabie о возвращают положение линеек прокрутки в виде интерфейса Adjustable;
- метод getScrollPosition() показывает кбординаты (x, y) точки компонента, находящейся в левом верхнем углу панели ScrollPane, в виде объекта класса Point;
- метод setScrollPosition(Point p) или setScrollPosition(int x, int y) прокручивает компонент в позицию (x, y).

16.4.17. Контейнер Window

Контейнер Window - это пустое окно, без внутренних элементов: рамки, строки заголовка, строки меню, полос прокрутки. Это просто прямоугольная область на экране. Окно типа Window самостоятельно, не содержится ни в каком контейнере, его не надо заносить в контейнер методом add(). Однако оно не связано с оконным менеджером графической системы. Следовательно, нельзя изменить его размеры, переместить в другое место экрана. Поэтому оно может быть создано только каким-нибудь уже существующим окном, владельцем (owner) или родителем (parent) окна window. Когда окно-владелец убирается с экрана, вместе с ним убирается и порожденное окно. Владелец окна указывается В конструкторе:

- window (Frame f) - создает окно, владелец которого — фрейм f;
- window (window owner) - создает окно, владелец которого - уже имеющееся окно или подкласс класса Window.

Созданное конструктором окно не выводится на экран автоматически. Его следует отобразить методом show(). Убрать окно с экрана можно методом hide(), а проверить, видно ли окно на экране - логическим методом isShowing().

Окно типа Window возможно использовать для создания всплывающих окон предупреждения, сообщения, подсказки. Для создания диалоговых окон есть подкласс Dialog, всплывающих меню - класс popupMenu.

Видимое на экране окно выводится на передний план методом toFront() или, наоборот, помещается на задний план методом toBack().

Уничтожить окно, освободив занимаемые им ресурсы, можно методом dispose().

Менеджер размещения компонентов в окне по умолчанию - BorderLayout. Окно создает свой экземпляр класса Toolkit, который возможно получить методом getToolkit().

Кроме событий класса Component: ComponentEvent, FocusEvent, KeyEvent, MouseEvent, при изменении размеров окна, его перемещении или удалении с экрана, а также показа на экране происходит событие windowEvent.

16.4.18. Контейнер Frame

Контейнер Frame - это полноценное готовое окно со строкой заголовка, в которую помещены кнопки контекстного меню, сворачивания окна в ярлык и разворачивания во весь экран и кнопка закрытия приложения. Заголовок окна записывается в конструкторе или методом setTitle(String title). Окно окружено рамкой. В него можно установить строку меню методом setMenuBar(MenuBar mb).

На кнопке контекстного меню в левой части строки заголовка изображена дымящаяся чашечка кофе - логотип Java. Вы можете установить там другое изображение методом seticonimage(Image icon), создав предварительно изображение icon в виде объекта класса image.

Все элементы окна Frame вычерчиваются графической оболочкой операционной системы по правилам этой оболочки. Окно Frame автоматически регистрируется в оконном менеджере графической оболочки и может перемещаться, менять размеры, сворачиваться в панель задач (task bar) с помощью мыши или клавиатуры, как "родное" окно операционной системы.

Создать окно типа Frame можно следующими конструкторами:

- Frame() - создает окно с пустой строкой заголовка;
- Frame(string title) - записывает аргумент title в строку заголовка.

Методы класса Frame осуществляют доступ к элементам окна, но не забывайте, что класс Frame наследует около 200 методов классов Component, Container и Window. В частности, наследуется менеджер размещения по умолчанию - BorderLayout.

Кроме событий класса Component: ComponentEvent, FocusEvent, KeyEvent, MouseEvent, при изменении размеров окна, его перемещении или удалении с экрана, а также показа на экране происходит событие windowEvent.

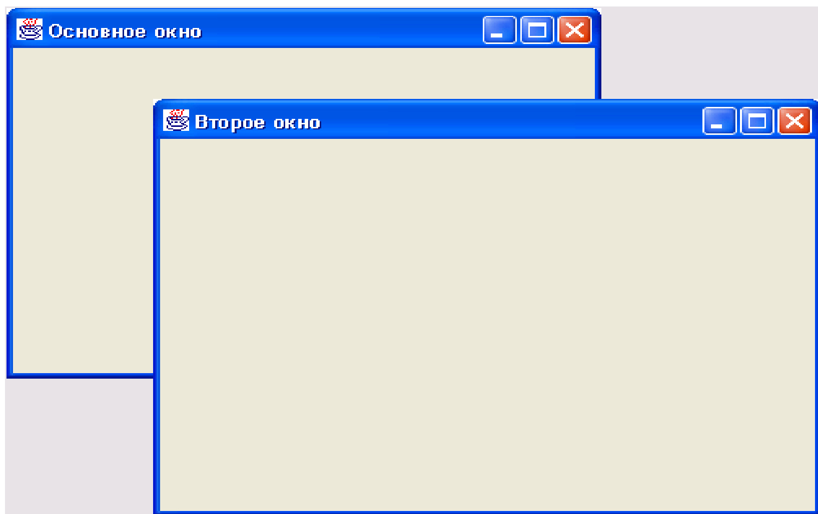
Программа листинга создает 2 окна типа Frame, в которые помещаются строки - метки Label. При закрытии основного окна щелчком по соответствующей кнопке в строке заголовка или комбинацией клавиш <Alt>+<F4> выполнение программы завершается обращением к методу system.exit(0), и закрываются оба окна. При закрытии второго окна происходит обращение к методу dispose(), и закрывается только это окно.

<u>Листинг. Создание двух окон</u>

```
import java.awt.* ;
import java.awt.event.*;
Class TwoFrames {
    public static void main(String[] args) {
        Fr1 f1 = new Frl(" Основное окно");
        Fr2 f2 = new Fr2(" Второе окно");
    }
}
Class Frl extends Frame {
    Fr1(String s) {
        super(s);
        setLayout(null);
        Font f = new Font("Serif", Font.BOLD, 15);
        setFont(f);
        Label L = new Label("Это главное окно", Labe1.CENTER);
        L.setBounds(10, 30, 180, 30);  add(L);
        setSize(200, 100);
        setvisible(true);
        addWindowListener (
            new WindowAdapter()  {
                public void windowClosing(WindowEvent ev) {
                    System.exit (0);
                }
            }
        );
    }
}
Class Fr2 extends Frame {
    Fr2(String s) {
        super(s);
        setLayout(null) ;
        Font f = new Font("Serif", Font.BOLD, 15);
        setFont(f);
        Label L= new Label("Это второе окно", Label.CENTER);
        L.setBounds(10, 30, 180, 30); add(L);
        setBounds(50, 50, 200, 100);
        setvisible(true);
        addWindowListener (
            new WindowAdapter()  {
                public void windowClosing(WindowEvent ev) {
                    dispose();
                }
```

```
            }
        );
    }
}
```

Это результат. Взаимное положение окон определяется оконным менеджером операционной системы и может быть не таким, какое показано на рисунке.



16.4.19. Контейнер Dialog

Контейнер Dialog - это окно обычно фиксированного размера, предназначенное для ответа на сообщения приложения. Оно автоматически регистрируется в оконном менеджере графической оболочки, следовательно, его можно перемещать по экрану, менять его размеры. Но окно типа Dialog, как и его суперкласс - окно типа window, - обязательно имеет владельца owner, который указывается в конструкторе. Окно типа Dialog может быть модальным (modal), в котором надо обязательно выполнить все предписанные действия, иначе из окна нельзя будет выйти.

В классе 7 конструкторов. Из них:

- Dialog(Dialog owner) — создает немодальное диалоговое окно с пустой строкой заголовка;
- Dialog(Dialog owner, string title) — создает немодальное диалоговое-окно со строкой заголовка title;

- Dialog(Dialog owner, String title, Boolean modal) — создает диалоговое окно, которое будет модальным, если modal = true.

4 других конструктора аналогичны, но создают диалоговые окна, принадлежащие окну типа Frame:

- Dialog(Frame owner)
- Dialog(Frame owner. String title)
- Dialog(Frame owner, Boolean modal)
- Dialog(Frame owner, String title, Boolean modal)

Среди методов класса интересны методы: isModal(), проверяющий состояние модальности, и setModal(Boolean modal), меняющий это состояние.

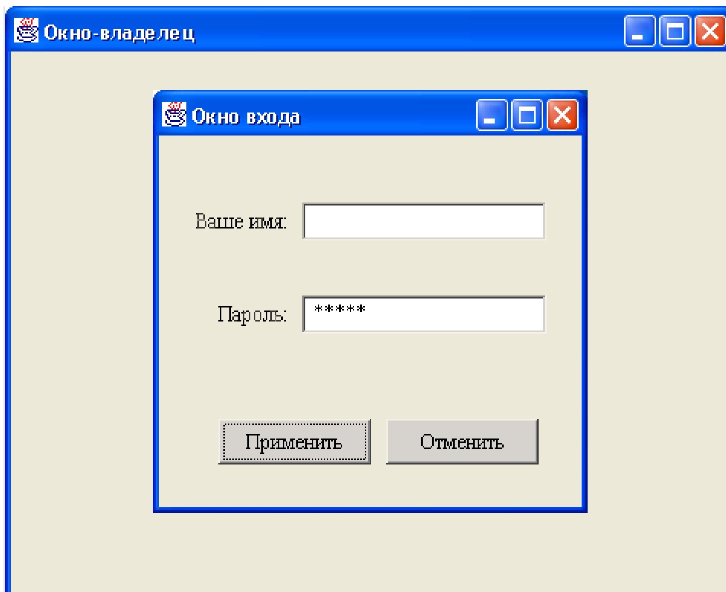
Кроме событий класса Component: ComponentEvent, FocusEvent, KeyEvent, MouseEvent, при изменении размеров окна, его перемещении или удалении с экрана, а также показа на экране происходит событие windowEvent.

В листинге создается модальное окно доступа, в которое вводится имя и пароль. Пока не будет сделан правильный ввод, другие действия невозможны.

Листинг. Модальное окно доступа

```
import j ava.awt.*;
import Java.awt.event.*;
Class LoginWin extends Dialog {
    LoginWin(Frame f, String s) {
        super(f, s, true);
        setLayout(null);
        setFont(new Font("Serif", Font.PLAIN, 14));
        Label L1 = new Label("Ваше имя:", Label.RIGHT);
        L1.setBounds(20, 30, 70, 25); add(L1);
        Label L2 = new Label("Пароль:", Label.RIGHT);
        L2.setBounds(20, 60, 70, 25); add(L2);
        TextField TF1 = new TextField(30);
        TF1.setBounds(100, 30, 160, 25); add(TF1);
        TextField TF2 = new TextField(30);
        TF2.setBounds(100, 60, 160, 25); add(TF2);  TF2.setEchoChar('*');
        Button B1 = new Button("Применить");
        B1.setBounds(50, 100, 100, 30); add(B1);
        Button B2 = new Button("Отменить");
        B2.setBounds(160, 100, 100, 30); add(B2);
        setBounds(50, 50, 300, 150);
    }
}
```

```
Class DialogTest extends Frame {
    DialogTest(String s) {
        super(s);
        setLayout(null); setSize(200, 100);
        setVisible(true);
        Dialog d = new LoginWin(this, " Окно входа"); d.setVisibl(true);
    }
    public static void main(String[] args) {
        Frame f = new DialogTest(" Окно-владелец");
        f.addWindowListener (
            new WindowAdapter()  {
                public void windowClosing(WindowEvent ev) {
                    System.exit(0);
                }
            }
        );
    }
}
```



16.4.20. Контейнер FileDialog

Контейнер FileDialog - это модальное окно с владельцем типа Frame, содержащее стандартное окно выбора файла операционной системы для открытия (константа LOAD) или сохранения (константа SAVE). Окна операционной системы создаются и помещаются в объект класса FileDialog автоматически.

В классе 3 конструктора:

- FileDialog(Frame owner) — создает окно с пустым заголовком для открытия файлоа;
- FileDialog(Frame owner, String title) — создает окно открытия файла с заголовком title;
- FileDialog(Frame owner, String title, int mode) — создает окно открытия или сохранения документа; аргумент mode имеет два значения: FileDialog.LOAD И FileDialog.SAVE.

Методы класса getDirectory() и getFiie() возвращают только выбранный каталог и имя файла в виде строки String. Загрузку или сохранение файла затем нужно производить методами классов ввода/вывода.

Можно установить начальный каталог для поиска файла и имя файла методами setDirectory(String dir) и setFile(String fileName).

Вместо конкретного имени файла fileName можно написать шаблон, например, *.java (первые символы — звездочка и точка), тогда в окне будут видны только имена файлов, заканчивающиеся точкой и словом java.

Метод setFilenameFilter(FilenameFilter filter) устанавливает шаблон filter для имени выбираемого файла. В окне будут видны только имена файлов, подходящие под шаблон. Этот метод не реализован в SUN JDK на платформе MS Windows.

Кроме событий класса Component: ComponentEvent, FocusEvent, KeyEvent, MouseEvent, при изменении размеров окна, его перемещении или удалении с экрана, а также показа на экране происходит событие windowEvent.

16.4.21. Класс MenuBar - меню

В контейнер типа Frame заложена возможность установки стандартной строки меню (menu bar), располагаемой ниже строки заголовка. Эта строка - объект класса MenuBar.

Все, что нужно сделать для установки строки меню в контейнере Frame - это создать объект класса MenuBar и обратиться к методу setMenuBar():

```
Frame f = new Frame("Пример меню");
```

```
MenuBar mb = new MenuBar();
f.setMenuBar(mb);
```

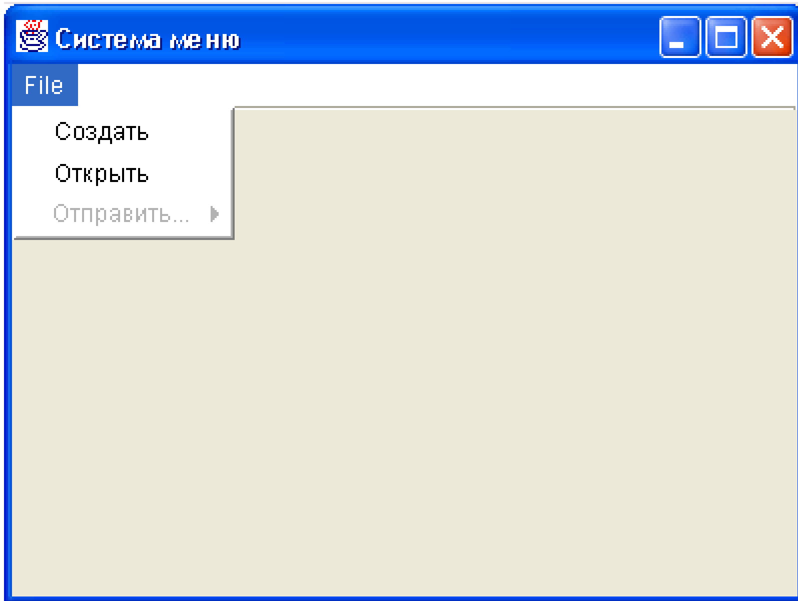
Если имя mb не понадобится, можно совместить 2 последних обращения к методам:

```
f.setMenuBar(new MenuBar());
```

Строка меню еще пуста и пункты меню не созданы. Каждый элемент строки меню - выпадающее меню (drop-down menu) - это объект класса Menu. Создать эти объекты и занести их в строку меню ничуть не сложнее, чем создать строку меню:

```
Menu mFile = new Menu("Файл");
mb.add(mFile);
Menu mEdit = new Menu("Правка");
mb.add(mEdit);
Menu mView = new Menu("Вид");
mb.add(mView);
Menu mHelp = new Menu("Справка");
mb.setHelpMenu(mHelp);
```

Элементы располагаются слева направо в порядке обращений к методам add(), как показано на рисунке. Во многих графических системах принято меню Справка (Help) прижимать к правому краю строки меню. Это достигается обращением к методу setHelpMenu(), но фактическое положение меню Справка определяется графической оболочкой.



Затем определяем каждое выпадающее меню, создавая его пункты. Каждый пункт меню - это объект класса MenuItem. Схема его создания и добавления к меню точно такая же, как и самого меню:

```
MenuItem create = new MenuItem("Создать");
mFile.add(create);
MenuItem open = new MenuItem("Открыть...");
mFile.add(open);
```

Пункты меню будут расположены сверху вниз в порядке обращения к методам add().

Часто пункты меню объединяются в группы. Одна группа от другой отделяется горизонтальной чертой. На рисунке черта проведена между командами Открыть и Отправить. Эта черта создается методом addSeparator() класса Menu или определяется как пункт меню с надписью специального вида - дефисом:

```
mFile.add(new Menuitem("-"));
```

Интересно, что класс Menu расширяет класс MenuItem, а не наоборот. Это означает, что меню само является пунктом меню, и позволяет задавать меню в качестве пункта другого меню, тем самым организуя вложенные подменю:

```
Menu send = new Menu("Отправить");
```

```
mFile.add(send);
```

Здесь меню send добавляется в меню mFile как один из его пунктов. Подменю send заполняется пунктами меню как обычное меню.

Часто команды меню создаются для выбора из них каких-то возможностей, подобно компонентам checkbox. Такие пункты можно выделить щелчком кнопки мыши или отменить выделение повторным щелчком. Эти команды - объекты класса CheckboxMenuItem:

```
CheckboxMenuItem disk = new CheckboxMenuItem("Диск А:", true);
send.add(disk);
send.add(new CheckboxMenuItem("Архив")) ;
```

Многие графические оболочки, но не MS Windows, позволяют создавать отсоединяемые (tear-off) меню, которые можно перемещать по экрану. Это указывается в конструкторе

```
Menu(String label, Boolean tearOff)
```

Если tearOff = true и графическая оболочка умеет создавать отсоединяемое меню, то оно будет создано. В противном случае этот аргумент просто игнорируется.

Наконец, надо назначить действия командам меню. Команды меню типа Menuitem порождают события типа ActionEvent, поэтому нужно присоединить к ним объект класса-слушателя как к обычным компонентам, записав что-то вроде

```
create.addActionListener(new SomeActionEventHandler())
open.addActionListener(new AnotherActionEventHandler())
```

Пункты типа CheckboxMenuItem порождают события типа ItemEvent, поэтому надо обращаться к объекту-слушателю этого события:

```
disk.addItemListener(new SomeItemEventHandler())
```

Очень часто действия, записанные в командах меню, вызываются не только щелчком кнопки мыши, но и "горячими" клавишами-акселераторами (shortcut), действующими чаще всего при нажатой клавише <Ctrl>. На экране в пунктах меню, которым назначены "горячие" клавиши, появляются подсказки вида Ctrl+N, Ctrl+O. "Горячая" клавиша определяется объектом класса MenuShortcut и указывается в его конструкторе константой класса KeyEvent, например:

```
MenuShortcut keyCreate = new MenuShortcut(KeyEvent.VK_N);
```

После этого "горячей" будет комбинация клавиш <Ctrl>+<N>. Затем полученный объект указывается в конструкторе класса MenuItem:

```
MenuItem create = new MenuItem("Создать", keyCreate);
```

Нажатие <Ctrl>+<N> будет вызывать окно создания.

Можно добавить еще нажатие клавиши <Shift>. Действие пункта меню будет вызываться нажатием комбинации клавиш <Shift>+<Ctrl>+<X>, если воспользоваться конструктором 2:

```
MenuShortcut(int key, Boolean useShift)
```

с аргументом useShift = true.

Программа рисования, созданная ранее, перегружена кнопками. Перенесем их действия в пункты меню. Добавим возможность манипуляции файлами и команду завершения работы. Это сделано в листинге ниже. Класс scribble не изменялся и в листинге не приведен.

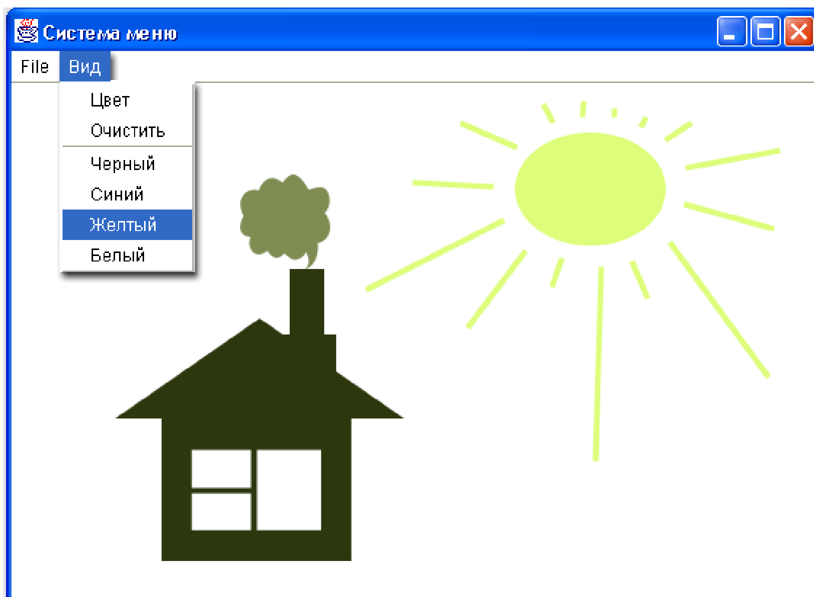
Листинг. Программа рисования с меню

```
import j ava.awt.*;
import j ava.awt.event.*;
public Class MenuScribble extends Frame {
    public MenuScribble(String s) {
        super(s);
        ScrollPane pane = new ScrollPane();
        pane.setSize(300, 300);
        add(pane, BorderLayout.CENTER);
        Scribble scr = new Scribble(this, 500, 500);
        pane.add(scr);
        MenuBar mb = new MenuBar();
        setMenuBar(mb);
        Menu f = new Menu("Файл");
        Menu v = new Menu("Вид");
        mb.add(f); mb.add(v);
        MenuItem open = new Menuitem("Открыть...",
            new MenuShortcut(KeyEvent.VK_0));
        MenuItem save = new Menuitem("Сохранить",
            new MenuShortcut(KeyEvent.VK_S));
        MenuItem saveAs = new Menultera("Сохранить как...");
        MenuItem exit = new Menuitem("Выход",
            new MenuShortcut(KeyEvent.VK_Q));
        f.add(open); f.add(save); f.add(saveAs);
        f.addSeparator(); f.add(exit);
        open.addActionListener (
            new ActionListener() {
```

```
                public void actionPerformed(ActionEvent e) {
                    FileDialog fd = new FileDialog(new Frame(),
                        " Загрузить", FileDialog.LOAD);
                    fd.setVisible(true);
                }
            }
        );
        saveAs.addActionListener  (
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    FileDialog fd = new FileDialog(new Frame(),
                        " Сохранить", FileDialog.SAVE);
                    fd.setVisible(true);
                }
            }
        );          exit.addActionListener  (
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    System.exit(0);
                }
            }
        );
        Menu c = new Menu("Цвет");
        MenuItem clear = new MenuItem("Очистить",
            new MenuShortcut(KeyEvent.VK_D));
        v.add(c); v.add(clear);
        MenuItem red = new MenuItem("Красный");
        MenuItem green = new MenuItem("Зеленый");
        MenuItem blue = new MenuItem("Синий");
        MenuItem black = new MenuItem("Черный");
        c.add(red); c.add(green); c.add(blue); c.add(black);
        red.addActionListener(scr);
        green.addActionListener(scr);
        blue.addActionListener(scr) ;
        black.addActionListener(scr) ;
        clear.addActionListener(scr) ;
        addWindowListener(new WinClose()); pack();
        setVisible(true) ;
    }
    Class WinClose extends WindowAdapter {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
```

```
        }
    }
    public static void main(String[] args) {
        new MenuScribble(" \"Рисовалка\" с меню");
    }
}
```

Это результат



16.4.22. Класс PopupMenu - всплывающее меню

Всплывающее меню (popup menu) появляется обычно при нажатии или отпускании правой или средней кнопки мыши и является контекстным (context) меню. Его команды зависят от компонента, на котором была нажата кнопка мыши. В языке Java всплывающее меню - объект класса PopupMenu. Этот класс расширяет класс Menu, следовательно, наследует все свойства меню и пункта меню Menultem. Всплывающее меню присоединяется не к строке меню типа MenuBar или к меню типа Menu в качестве подменю, а к определенному компоненту. Для этого в классе Component есть метод add(PopupMenu menu).

У некоторых компонентов, например TextFieid и TextArea, уже существует всплывающее меню. Подобные меню нельзя переопределить.

Присоединить всплывающее меню можно только к одному компоненту. Если надо использовать всплывающее меню с несколькими компонентами в контейнере, то его присоединяют к контейнеру, а нужный компонент определяют с помощью метода getComponent() класса MouseEvent.

Кроме унаследованных свойств и методов, в классе PopupMenu есть метод show(Component comp, int x, int y), показывающий всплывающее меню на экране так, что его левый верхний угол располагается в точке (x, y) в системе координат компонента comp. Чаще всего это компонент, на котором нажата кнопка мыши, возвращаемый методом getComponent(). Компонент comp должен быть внутри контейнера, к которому присоединено меню, иначе возникнет исключительная ситуация.

Всплывающее меню появляется в MS Windows при отпускании правой кнопки мыши, а в других графических системах могут быть иные правила. Чтобы учесть эту разницу, в класс MouseEvent введен логический метод isPopupTrigger(), показывающий, что возникшее событие мыши вызывает появление всплывающего меню. Его нужно вызывать при возникновении всякого события мыши, чтобы проверять, не является ли оно сигналом к появлению всплывающего меню, т.е. обращению к методу show(). Было бы слишком неудобно включать такую проверку во все 7 методов классов-слушателей событий мыши. Поэтому метод IsPopupTrigger() лучше вызывать в методе processMouseEvent().

Переделаем еще раз программу рисования, введя в класс scribble всплывающее меню для выбора цвета рисования и очистки окна и изменив обработку событий мыши. Для простоты уберем строку меню.

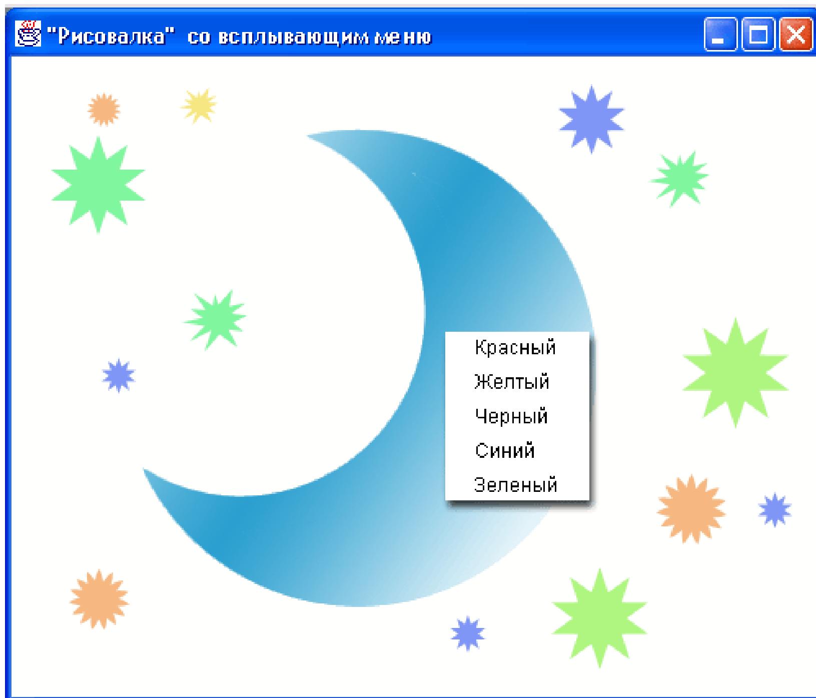
Листинг Программа рисования с всплывающим меню

```
import j ava.awt.* ;
import j ava.awt.event.*;
public Class PopupMenuScribble extends Frame {
    public PopupMenuScribble(String s) {
        super (s) ;
        ScrollPane pane = new ScrollPane();
        pane.setSize(300, 300);
        add(pane, BorderLayout.CENTER);
        Scribble scr = new Scribble(this, 500, 500);
        pane.add(scr);
        addWindowListener(new WinClose());
        pack ();
        setVisible(true);
    }
```

```
    Class WinClose extends WindowAdapter {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    }
    public static void main(String[] args) {
        new PopupMenuScribble(" \"Рисовалка\" с всплывающим меню");
    }
}
Class ScriBble extends Component implements ActionListener {
    protected int lastX, lastY, w, h;
    protected Color currColor = Color.black;
    protected Frame f;
    protected PopupMenu c;
    public Scribble(Frame frame, int width, int height) {
        f = frame; w = width; h = height;
        enableEvents(AWTEvent.MOUSE_EVENT_MASK |
             AWTEvent.MOUSEJtoTIONJEVENT_MASK);
        c = new PopupMenu ("Цвет") ;
        add(c);
        MenuItem clear = new MenuItem("Очистить",
            new MenuShortcut(KeyEvent.VK_D));
        MenuItem red = new MenuItem("Красный");
        MenuItem green = new MenuItem("Зеленый");
        MenuItem blue = new MenuItem("Синий");
        MenuItem black = new MenuItem("Черный");
        c.add(red); c.add(green); c.add(blue);
        c.add(black); c.addSeparator(); c.add(clear);
        red.addActionListener(this);
        green.addActionListener(this);
        blue.addActionListener(this);
        black.addActionListener(this);
        clear.addActionListener(this);
    }
    public Dimension getPreferredSize()  {
        return new Dimension(w, h);
    }
    public void actionPerformed(ActionEvent event) {
        String s = event.getActionCommand();
        if (s.equals("Очистить")) repaint();
        else if (s.equals("Красный")) currColor = Color.red;
        else if (s.equals("Зеленый")) currColor = Color.green;
```

```
        else if (s.equals("Синий")) currColor = Color.blue;
        else if (s.equals("Черный")) currColor = Color.black;
    }
    public void processMouseEvent(MouseEvent e) {
        if (e.isPopupTrigger())
            c.show(e.getComponent(), e.getXO, e.getY());
        else if (e.getld() == MouseEvent.MOUSE_PRESSED) {
                lastX = e.getX(); lastY = e.getY();
            }
        else super.processMouseEvent(e);
    }
    public void processMouseMotionEvent(MouseEvent e) {
        if (e.getld() = MouseEvent.MOUSE_DRAGGED) {
            Graphics g = getGraphics();
            g.setColor(currColor) ;
            g.drawLine(lastX, lastY, e.getX(), e.getY());
            lastX = e.getX(); lastY = e.getY();
        }
        else super.processMouseMotionEvent(e);
    }
}
```

Это результат



16.4.23. Создание собственных компонентов

Создать свой компонент, дополняющий свойства и методы уже существующих компонентов AWT, очень просто - надо лишь образовать свой класс как расширение существующего класса Button, TextFieid или другого класса-компонента.

Если надо скомбинировать несколько компонентов в один, новый, компонент, то достаточно расширить класс Panel, расположив компоненты на панели.

Если же требуется создать совершенно новый компонент, то AWT предлагает две возможности: создать "тяжелый" или "легкий" компонент. Для создания собственных "тяжелых" компонентов в библиотеке AWT есть класс canvas - пустой компонент, для которого создается свой peer-объект графической системы.

16.4.24. Компонент Canvas

Компонент Canvas - это пустой компонент. Класс Canvas очень прост - в нем только конструктор по умолчанию Canvas() и пустая реализация метода paint(Graphics g).

Чтобы создать свой "тяжелый" компонент, необходимо расширить класс Canvas, дополнив его нужными полями и методами, и при необходимости переопределить метод paint().

Например, как вы заметили, на стандартной кнопке Button можно написать только одну текстовую строку. Нельзя написать несколько строк или отобразить на кнопке рисунок. Создадим свой "тяжелый" компонент - кнопку с рисунком.

В листинге кнопка с рисунком - класс FiowerButton. Рисунок задается методом drawFiower(), а рисуется методом paint(). Метод paint() кроме того чертит по краям кнопки внизу и справа отрезки прямых, изображающих тень, отбрасываемую "выпуклой" кнопкой. При нажатии кнопки мыши на компоненте такие же отрезки чертятся вверху и слева - кнопка "вдавилась". При этом рисунок сдвигается на два пиксела вправо вниз - он "вдавливается" в плоскость окна.

Кроме этого, в классе FiowerButton задана реакция на нажатие и отпускание кнопки мыши. При каждом нажатии и отпускании кнопки меняется значение поля isDown и кнопка перечерчивается методом repaint(). Это достигается выполнением методов mousePressed() и mouseReleased().

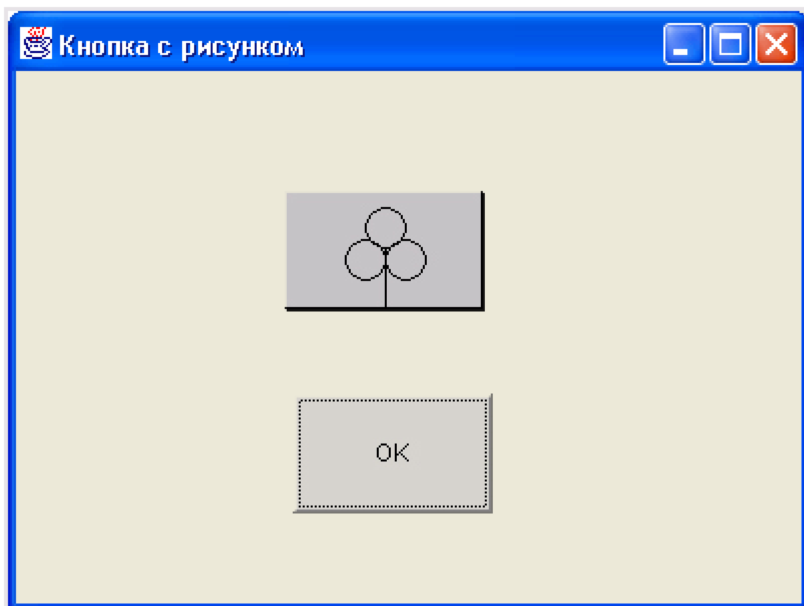
Для сравнения на рисунке вывода рядом помещена стандартная кнопка типа Button того же размера.

<u>Листинг. Кнопка с рисунком</u>

```
import j ava.awt.*;
import j ava.awt.event.*;
Class FiowerButton extends Canvas implements MouseListener {
    private Boolean isDown=false;
    public FiowerButton() {
        super();
        setBackground(Color.lightGray);
        addMouseListener(this);
    }
    public void drawFlower(Graphics g, int x, int y, int w, int h) {
        g.drawOvalfx + 2*w/5 - 6, y, w/5, w/5);
        g.drawLine(x + w/2 - 6, y + w/5, x + w/2 - 6, y + h - 4);
        g.drawOvalfx + 3*w/10 -6, y + h/3 - 4, w/5, w/5) ;
        g.drawOval(x + w/2 - б, y + h/3 - 4, w/5, w/5);
     }
    public void paint(Graphics g) {
        int w = getSizeO.width, h = getSize().height;
        if (isDown)  {
```

```
                g.drawLine(0, 0, w - 1, 0) ;
                g.drawLined, 1, w - I, I);
                g.drawLine(0, 0, 0, h - 1);
                g.drawUne (1, 1, 1, h - 1);
                drawFlower(g, 8, 10, w, h);
            }
            else {
                g.drawLine(0, h - 2, w - 2, h - 2);
                g.drawLined, h - 1, w - I, h - I);
                g.drawLinefw - 2, h - 2, w - 2, 0);
                g.drawLinefw - 1, h - 1, w - 1, 1);
                drawFlower (g, 6, 8, w, h) ;
            }
        }
        public void mousePressed(MouseEvent e) {
            isDown=true; repaint();
        }
        public void mouseReleased(MouseEvent e) {
            isDown=false; repaint();
        }
        public void mouseEntered(MouseEvent e){}
        public void mouseExited(MouseEvent e) {}
        public void mouseClicked(MouseEvent e){)
}
Class DrawButton extends Frame {
    DrawButton(String s) {
        super (s) ;
        setLayout(null);
        Button b = new Button("OK");
        b.setBounds(200, 50, 100, 60); add(b);
        FlowerButton d = new FlowerButton();
        d.setBounds(50, 50, 100, 60); add(d);
        setSize(400, 150);
        setVisible(true);
    }
    public static void main(String[] args) {
        Frame f= new DrawButton(" Кнопка с рисунком");
        f.addWindowListener (
            new WindowAdapter() {
                public void windowClosing(WindowEvent ev) {
                    System.exit(0);
                }
```

```
            }
        );
    }
}
```



16.4.25. Создание "легкого" компонента

"Легкий" компонент, не имеющий своего peer-объекта в графической системе, создается как прямое расширение класса Component или Container. При этом необходимо задать те действия, которые в "тяжелых" компонентах выполняет peer-объект.

Например, заменив в листинге заголовок класса FlowerButton строкой

```
Class FlowerButton extends Component implements MouseListener{
```

а затем перекомпилировав и выполнив программу, вы получите "легкую" кнопку, но увидите, что ее фон стал белым, потому что метод

setBackground(Color.lightGray) не сработал.

Это объясняется тем, что теперь всю черную работу по изображению кнопки на экране выполняет не peer-двойник кнопки, а "тяжелый" контейнер, в котором расположена кнопка, в нашем случае класс Frame. Контейнер же ничего не

знает о том, что надо обратиться к методу setBackground о, он рисует только то, что записано в методе paint(). Придется убрать метод setBackground() из конструктора и заливать фон серым цветом вручную в методе paint().

Легкий" контейнер не умеет рисовать находящиеся в нем "легкие" компоненты, поэтому в конце метода paint() "легкого" контейнера нужно обратиться к методу paint() суперкласса: super.paint(g);

Тогда рисованием займется "тяжелый" суперкласс-контейнер. Он нарисует и лежащий в нем "легкий" контейнер, и размещенные в контейнере "легкие" компоненты.

Совет. Завершайте метод paint() "легкого" контейнера обращением к методу paint() суперкласса.

Предпочтительный размер "тяжелого" компонента устанавливается peer-объектом, а для "легких" компонентов его надо задать явно, переопределив метод getPreferredSize(), иначе некоторые менеджеры размещения, например FiowLayout(), установят нулевой размер, и компонент не будет виден на экране.

Совет. Переопределяйте метод getPreferredSize().

Интересная особенность "легких" компонентов — они изначально рисуются прозрачными, не закрашенная часть прямоугольного объекта не будет видна. Это позволяет создать компонент любой видимой формы.

16.4.26. Размещение компонентов

До сих пор мы размешали компоненты "вручную", задавая их размеры и положение в контейнере абсолютными координатами в координатной системе контейнера. Для этого мы применяли метод setBounds().

Такой способ размешает компоненты с точностью до пиксела, но не позволяет перемещать их. При изменении размеров окна с помощью мыши компоненты останутся на своих местах, привязанными к левому верхнему углу контейнера. Кроме того, нет гарантии, что все мониторы отобразят компоненты так, как вы задумали.

Чтобы учесть изменение размеров окна, надо задать размеры и положение компонента относительно размеров контейнера, например, так:

```
int w = getSize().width;           // Получаем ширину
int h = getSizeO.height;           // и высоту контейнера
Button Ь = new Button("OK");   / Создаем кнопку
b.setBounds(9*w/20, 4*h/5, w/10, h/10);
```

и при всяком изменении размеров окна задавать расположение компонента заново.

Чтобы избавить программиста от этой кропотливой работы, в библиотеку AWT внесены два интерфейса: LayoutManager и порожденный от него интерфейс LayoutManager2, а также 5 реализаций этих интерфейсов: классы BorderLayout, CardLayout, FlowLayout, GridLayout и GridBagLayout. Эти классы названы менеджерами размещения (layout manager) компонентов.

Каждый программист может создать свои менеджеры размещения, реализовав интерфейсы LayoutManager или LayoutManager2.

Посмотрим, как размещают компоненты эти классы.

16.4.27. Менеджер FlowLayout

Наиболее просто поступает менеджер размещения FlowLayout. Он укладывает в контейнер один компонент за другим слева направо как кирпичи, переходя от верхних рядов к нижним. При изменении размера контейнера "кирпичи" перестраиваются. Компоненты поступают в том порядке, в каком они заданы в методах add().

В каждом ряду компоненты могут прижиматься к левому краю, если в конструкторе аргумент align равен FlowLayout.LEFT, к правому краю, если этот аргумент FlowLayout.RIGHT, или собираться в середине ряда, если FlowLayout.CENTER.

Между компонентами можно оставить промежутки (gap) по горизонтали hgap и вертикали vgap. Это задается в конструкторе: FlowLayout(int align, int hgap, int vgap).

Второй конструктор задает промежутки размером 5 пикселов: FlowLayout(int align).

Третий конструктор определяет выравнивание по центру и промежутки 5 пикселов: FlowLayout().

После формирования объекта эти параметры можно изменить методами:

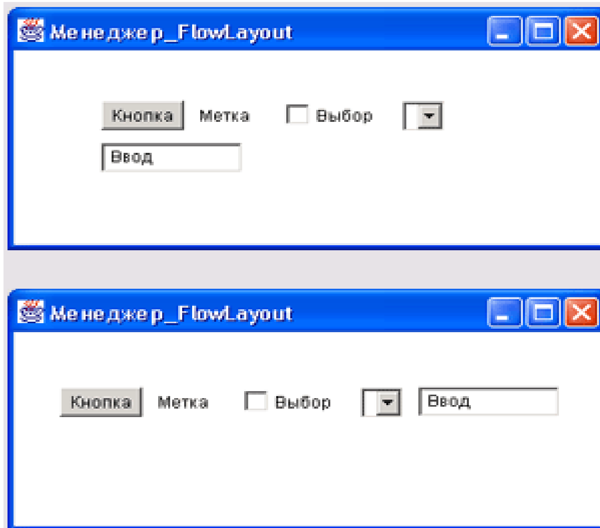
- setHgapfint hgap)
- setVgap(int vgap)
- setAlignment(int align)

В листинге создаются кнопка Button, метка Label, кнопка выбора checkbox, раскрывающийся список choice, поле ввода TextFieid и все это размещается в контейнере Frame.

Листинг. Менеджер размещения FlowLayout

```
import j ava.awt.*;
import j ava.awt.event.*;
Class FlowTest extends Frame {
    FlowTest(String s)  {
        super(s);
        setLayout (new FlowLayout (FlowLayout.LEFT, 10, 10));
        add(new Button("Кнопка"));
        add(new Label("Метка"));
        add(new Checkbox("Выбор"));
        add(new Choice());
        add(new TextFieldt"Справка", 10));
        setSize(300, 100); setVisible(true);
    }
    public static void main(String[] args) {
        Frame f = new FlowTest(" Менеджер FlowLayout");
        f.addWindowListener (
            new WindowAdapter()  {
                public void windowClosing(WindowEvent ev) {
                    System.exit(0);
                }
            }
        );
    }
}
```

Такой результат при разных размерах окна



16.4.28. Менеджер BorderLayout

Менеджер размещения BorderLayout делит контейнер на пять неравных областей, полностью заполняя каждую область одним компонентом. Области получили географические названия NORTH, SOUTH, WEST, EAST И CENTER.

Метод add() в случае применения BorderLayout имеет два аргумента: ссылку на компонент comp и область region, в которую помещается компонент — одну из перечисленных выше констант:

```
add(Component comp, String region)
```

Обычный метод add(Component comp) с одним аргументом помещает компонент в область CENTER.

В классе 2 конструктора:

- BorderLayout() — между областями нет промежутков;
- BorderLayout(int hgap int vgap) — между областями остаются горизонтальные hgap и вертикальные vgap промежутки, задаваемые в пикселах.

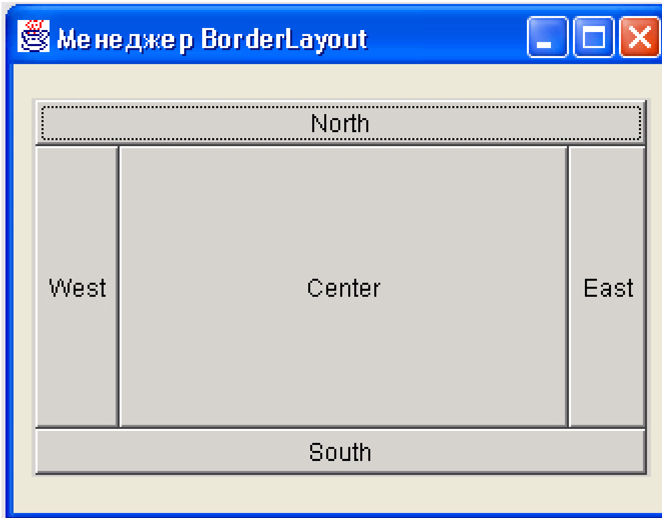
Если в контейнер помещается менее 5 компонентов, то некоторые области не используются и не занимают места в контейнере. Если не занята область CENTER, то компоненты прижимаются к границам контейнера.

В листинге создаются 5 кнопок, размещаемых в контейнере. Заметьте отсутствие установки менеджера в контейнере setLayout() - менеджер BorderLayout установлен в контейнере Frame по умолчанию.

Листинг. Менеджер размещения BorderLayout

```
import java.awt.*;
import ]ava.awt.event.* ;
Class BorderTest extends Frame {
    BorderTest(String s){ super(s);
    add(new Button("North"), BorderLayout.NORTH);
    add(new Button("South"), BorderLayout.SOUTH);
    add(new Button("West"), BorderLayout.WEST);
    add(new Button("East"), BorderLayout.EAST);
    add(new Button("Center"));
    setSize(300, 200);
    setVisible(true);
}
public static void main(String[] args) {
    Frame f= new BorderTest(" Менеджер BorderLayout");
    f.addWindowListener (
        new WindowAdapter()  {
            public void windowClosing(WindowEvent ev) {
                System.exit(0);
            }
        }
    );
}
```

Это результат



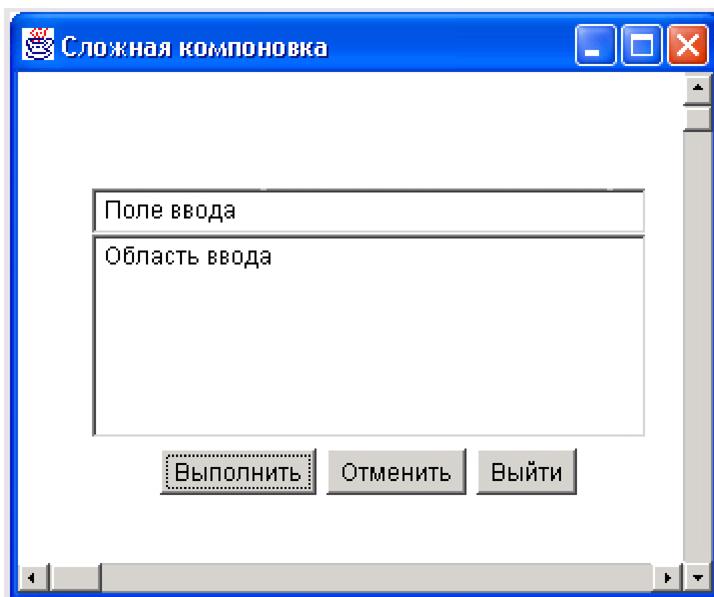
Менеджер размещения BorderLayout кажется неудобным: он располагает не больше 5 компонентов, последние растекаются по всей области, области имеют странный вид. Но дело в том, что в каждую область можно поместить не компонент, а панель, и размещать компоненты на ней. Напомним, что на панели Panel менеджер размещения по умолчанию FlowLayout.

Листинг. Сложная компоновка

```
import j ava.awt.*;
import java.awt.event.*;
Class BorderPanelTest extends Frame {
    BorderPanelTest(String s) {
        super(s);
        // Создаем панель p2 с тремя кнопками
        Panel p2 = new Panel();
        p2.add(new Button("Выполнить"));
        p2.add(new Button("Отменить"));
        p2.add(new Button("Выйти"));
        Panel p1 = new Panel();
        p1.setLayout(new BorderLayout());
        // Помещаем панель p2 с кнопками на "юге" панели p1
        p1.add(p2, BorderLayout.SOUTH);
        // Поле ввода помещаем на "севере"
        p1.add(new TextFieldt"Поле ввода", 20), BorderLayout.NORTH);
        // Область ввода помещается в центре
```

```
        p1.add(new TextArea("Область ввода", 5, 20,
            TextArea.SCROLLBARS_NONE), BorderLayout.CENTER);
        add(new Scrollbar(Scrollbar.HORIZONTAL), BorderLayout.SOUTH);
        addfnew Scrollbar(Scrollbar.VERTICAL), BorderLayout.EAST);
        // Панель p1  в "центре" контейнера add(p1, BorderLayout.CENTER);
        setSizePOO, 200);
        setVisible(true) ;
    }
    public static void main(String[] args) {
        Frame f= new BorderPanelTest(" Сложная компоновка");
        f.addWindowListener (
            new WindowAdapter()  {
                public void windowClosing(WindowEvent ev) {
                    System.exit(0);
                }
            }
        );
    }
}
```

Это результат



16.4.29. Менеджер GridLayout

Менеджер размещения GridLayout расставляет компоненты в таблицу с заданным в конструкторе числом строк rows и столбцов columns:

GridLayout(int rows, int columns)

Все компоненты получают одинаковый размер. Промежутков между компонентами нет.

Второй конструктор позволяет задать промежутки между компонентами в пикселах по горизонтали hgap и вертикали vgap:

GridLayout(int rows, int columns, int hgap, int vgap)

Конструктор по умолчанию GridLayout() задает таблицу размером 0x0 без промежутков между компонентами. Компоненты будут располагаться в одной строке.

Компоненты размещаются менеджером GridLayout слева направо по строкам созданной таблицы в том порядке, в котором они заданы в методах add().

Нулевое количество строк или столбцов означает, что менеджер сам создаст нужное их число.

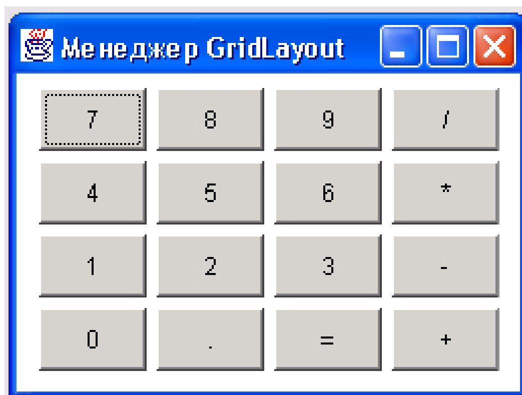
В листинге выстраиваются кнопки для калькулятора.

Листинг. Менеджер GridLayout

```
import Java.awt.*;
import j ava.awt.event.*;
import java.util.*;
Class GridTest extends Frame {
    GridTest(String s) {
        super(s);
        setLayout(new GridLayout(4, 4, 5, 5));
        StringTokenizer st = new StringTokenizer("7 89/456*123-0.=+");
        while(st.hasMoreTokens())
        add(new Button(st.nextToken()));
        setSize(200, 200); setvisible(true);
    }
    public static void main(String[] args) {
        Frame f= new GridTestt" Менеджер GridLayout");
        f.addWindowListener (
            new WindowAdapter()  {
                public void windowClosing(WindowEvent ev) {
                    System.exit(0);
```

```
                }
            }
        );
    }
}
```

Это результат



16.4.30. Менеджер CardLayout

Менеджер размещения cardLayout своеобразен - он показывает в контейнере только один, первый (first), компонент. Остальные компоненты лежат под первым в определенном порядке как игральные карты в колоде. Их расположение определяется порядком, в котором написаны методы add(). Следующий компонент можно показать методом next(Container с), предыдущий - методом previous(Container с), Последний - методом last(Container с), первый - методом first(Container с). Аргумент этих методов — ссылка на контейнер, в который помещены компоненты, обычно this.

В классе 2 конструктора:

- CardLayout() - не отделяет компонент от границ контейнера;
- CardLayout(int hgap, int vgap) - задает горизонтальные hgap и вертикальные vgap поля.

Менеджер CardLayout позволяет организовать и произвольный доступ к компонентам. Метод add() для менеджера CardLayout имеет своеобразный вид:

add(Component comp, Object constraints)

Здесь аргумент constraints должен иметь тип String и содержать имя компонента. Нужный компонент с именем name можно показать методом:

show(Container parent, String name)

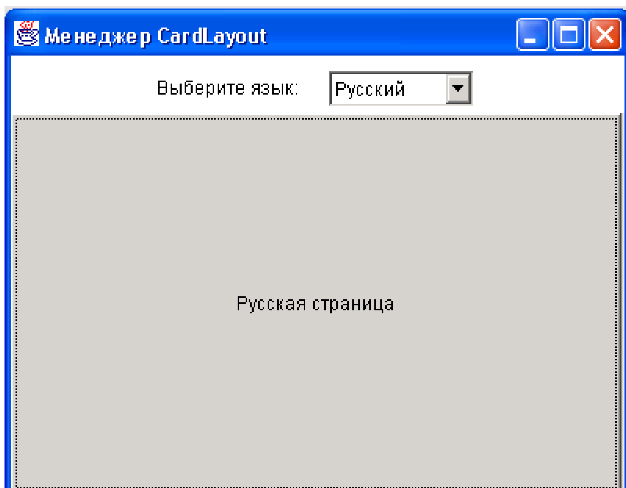
В листинге менеджер размещения c1 работает с панелью p, помещенной в "центр" контейнера Frame. Панель p указывается как аргумент parent в методах next() и show(). На "север" контейнера Frame отправлена панель p2 с меткой и раскрывающимся списком ch.

Листинг. Менеджер CardLayout

```
import j ava.awt.*;
import j ava.awt.event.*;
Class CardTest extends Frame {
    CardTest(String s)  {
        super(s);
        Panel p = new Panel();
        CardLayout cl = new CardLayout();
        p.setLayout(cl);
        p.add(new Button("Русская страница"),"pagel");
        p.add(new Button("English page"), "page2");
        p.add(new Button("Deutsche Seite"), "page3");
        add(p);
        cl.next(p);
        cl.show(p, "pagel");
        Panel p2 = new Panel();
        p2.add(new Label("Выберите язык:"));
        Choice ch = new Choice();
        ch.add("Русский");
        ch.add("Английский");
        ch.add("Немецкий");'
        p2.add(ch);
        add(p2, BorderLayout.NORTH);
        setSize(400, 300);
        setvisible(true); }
        public static void main(String[] args){
        Frame f= new CardTest{" Менеджер CardLayout");
        f.addWindowListener (
            new WindowAdapter()  {
                public void windowClosing(WindowEvent ev) {
                    System.exit(0);
                }
            }
        );
    }
```

}

Результат



16.4.31. Менеджер GridBagLayout

Менеджер размещения GridBagLayout расставляет компоненты наиболее гибко, позволяя задавать размеры и положение каждого компонента. Но он оказался очень сложным и применяется редко.

В классе GridBagLayout есть только 1 конструктор по умолчанию, без аргументов. Менеджер класса GridBagLayout, в отличие от других менеджеров размещения, не содержит правил размещения. Он играет только организующую роль. Ему передаются ссылка на компонент и правила расположения этого компонента, а сам он помещает данный компонент по указанным правилам в контейнер. Все правила размещения компонентов задаются в объекте другого класса, GridBagConstraints.

Менеджер размещает компоненты в таблице с неопределенным заранее числом строк и столбцов. Один компонент может занимать несколько ячеек этой таблицы, заполнять ячейку целиком, располагаться в ее центре, углу или прижиматься к краю ячейки.

Класс GridBagConstraints содержит 11 полей, определяющих размеры компонентов, их положение в контейнере и взаимное положение, и несколько констант - значений некоторых полей. Они перечислены в таблице. Эти параметры определяются конструктором, имеющим одиннадцать аргументов. Второй

конструктор - конструктор по умолчанию - присваивает параметрам значения, заданные по умолчанию.

Поле	Значение
anchor	Направление размещения компонента в контейнере. Константы: CENTER, NORTH, EAST, NORTHEAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST, и NORTHWEST; no умолчанию CENTER
fill	Растяжение компонента для заполнения ячейки. Константы: NONE, HORIZONTAL, VERTICAL, BOTH; ПО умолчанию NONE
gridheight	Количество ячеек в колонке, занимаемых компонентом. Целое типа int, по умолчанию 1. Константа REMAINDER означает, что компонент займет остаток колонки, RELATIVE — будет следующим по порядку в колонке
gridwidth	Количество ячеек в строке, занимаемых компонентом. Целое типа int, по умолчанию 1. Константа REMAINDER означает, что компонент займет остаток строки, RELATIVE — будет следующим в строке по порядку
gridx	Номер ячейки в строке. Самая левая ячейка имеет номер 0. По умолчанию константа RELATIVE, что означает: следующая по порядку
gridy	Номер ячейки в столбце. Самая верхняя ячейка имеет номер 0. По умолчанию константа RELATIVE, что означает: следующая по порядку
insets	Поля в контейнере. Объект класса insets; по умолчанию объект с нулями
ipadx, ipady	Горизонтальные и вертикальные поля вокруг компонентов; по умолчанию 0
weightx, weighty	Пропорциональное растяжение компонентов при изменении размера контейнера; по умолчанию 0,0

Как правило, объект класса GridBagConstraints создается конструктором по умолчанию, затем значения нужных полей меняются простым присваиванием новых значений, например:

```
GridBagConstraints gbc = new GridBagConstraints();
gbc.weightx = 1.0;
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.gridheight =2;
```

После создания объекта gbc класса GridBagConstraints менеджеру размещения указывается, что при помещении компонента comp в контейнер следует применять правила, занесенные в объект gbc. Для этого применяется метод

```
add(Component comp, GridBagConstraints gbc)
```

Итак, схема применения менеджера GridBagLayout такова:

```
GridBagLayout gbl = new GridBagLayout();      // Создаем менеджер
setLayout(gbl);                               // Устанавливаем его в контейнер
    // Задаем правила размещения по умолчанию
GridBagConstraints c = new GridBagConstraints();
Button b2 = new Button();             // Создаем компонент
C.gridwidth =2;                       // Меняем правила размещения
add(bl, C);   // Помещаем компонент b2 в контейнер по правилам размещения C
Button Ь2 = new Button();             // Создаем следующий компонент
C.gridwidth = 1;                      // Меняем правила для его размещения
add(b2, C);                           // Помещаем в контейнер
```

16.4.32. Обработка событий

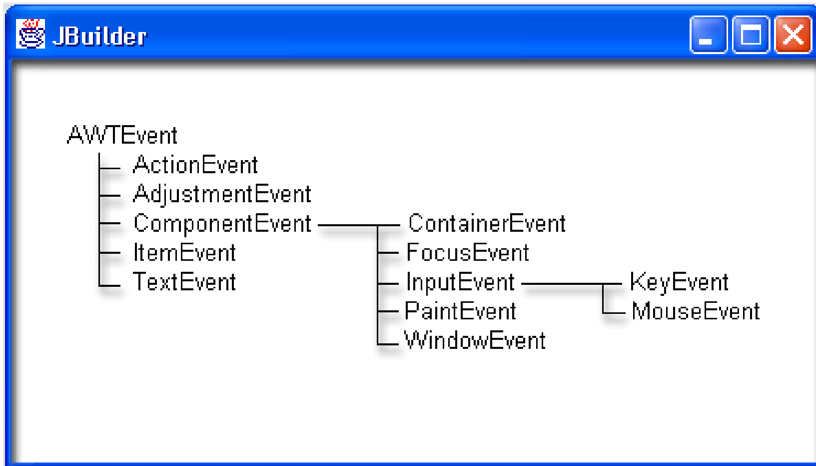
Мы написали много программ, создающих интерфейсы, но, собственно, интерфейса, т.е. взаимодействия с пользователем, эти программы не обеспечивают. Можно щелкать по кнопке на экране, она будет "вдавливаться" в плоскость экрана, но больше ничего не будет происходить. Можно ввести текст в поле ввода, но он не станет восприниматься и обрабатываться программой. Все это происходит из-за того, что мы не задали обработку действий пользователя, обработку событий.

Событие (Event) в библиотеке AWT возникает при воздействии на компонент какими-нибудь манипуляциями мышью, при вводе с клавиатуры, при перемещении окна, изменении его размеров. Объект, в котором произошло событие, называется источником (source) события.

Все события в AWT классифицированы. При возникновении события исполняющая система Java автоматически создает объект соответствующего событию класса. Этот объект не производит никаких действий, он только хранит все сведения о событии.

Во главе иерархии классов-событий стоит класс EventObject из пакета java.util - непосредственное расширение класса Object. Его расширяет абстрактный класс AWTEvent из пакета java.awt - глава классов, описывающих события библиотеки AWT. Дальнейшая иерархия классов-событий показана на рисунке. Все классы, отображенные на рисунке, кроме класса AWTEvent, собраны в пакет java.awt.event.

События типа ComponentEvent, FbeusEvent, KeyEvent, MouseEvent возникают во всех компонентах, а события типа ContainerEvent - только в контейнерах: Container, Dialog, FileDialog, Frame, Panel, ScrollPane, Window.



События типа WindowEvent возникают только в окнах: Frame, Dialog, FileDialog, Window.

События типа TextEvent генерируются только в контейнерах Textcomponent, TextArea, TextField.

События типа ActionEvent проявляются только в контейнерах Button, List, TextField.

События типа ItemEvent возникают только в контейнерах Checkbox, Choice, List.

Ссобытия типа AdjustmentEvent возникают только в контейнере Scrollbar.

Узнать, в каком объекте произошло событие, можно методом getSource() класса EventObject. Этот метод возвращает тип Object.

В каждом из этих классов-событий определен метод paramString(), возвращающий содержимое объекта данного класса в виде строки String. Кроме того, в каждом классе есть свои методы, предоставляющие те или иные сведения о событии. В частности, метод getId() возвращает идентификатор (identifier) события - целое число, обозначающее тип события. Идентификаторы события определены в каждом классе-событии как константы.

Методы обработки событий описаны в интерфейсах- слушателях (listener). Для каждого типа событий, кроме inputEvent (это событие редко используется самостоятельно), есть свой интерфейс. Имена интерфейсов составляются из имени события и слова Listener (слушатель), например, ActionListener, MouseListener. Методы интерфейса "слушают", что происходит в потенциальном источнике события. При возникновении события эти методы автоматически выполняются, получая в качестве аргумента объект-событие и используя при обработке сведения о событии, содержащиеся в этом объекте.

Чтобы задать обработку события определенного типа, надо реализовать соответствующий интерфейс. Классы, реализующие такой интерфейс, классы-обработчики (handlers) события,, называются слушателями (listeners): они "слушают", что происходит в объекте, чтобы отследить возникновение события и обработать его.

Чтобы связаться с обработчиком события, классы-источники события должны получить ссылку на экземпляр eventHandier класса-обработчика события одним из методов addXxxListener(XxxEvent eventHandier), где Xxx — имя события.

Такой способ регистрации, при котором слушатель оставляет "визитную карточку" источнику для своего вызова при наступлении события, называется обратный вызов (callback). Им часто пользуются студенты, которые, звоня родителям и не желая платить за телефонный разговор, говорят: "Перезвони мне по такому-то номеру". Обратное действие — отказ от обработчика, прекращение прослушивания - выполняется методом removeXxxListener().

Таким образом, компонент-источник, в котором произошло событие, не занимается его обработкой. Он обращается к экземпляру класса-слушателя, умеющего обрабатывать события, делегирует (delegate) ему полномочия по обработке.

Такая схема получила название схемы делегирования (delegation). Она удобна тем, что мы можем легко сменить класс-обработчик и обработать событие по-другому или назначить несколько обработчиков одного и того же события. С другой стороны, мы можем один обработчик назначить на прослушивание нескольких объектов-источников событий.

Замечание. В JDK 1.0 была принята другая модель обработки событий. Не удивляйтесь, читая старые книги и просматривая исходные тексты старых программ, но и не пользуйтесь старой моделью.

Приведем пример. Пусть в контейнер типа Frame помещено поле ввода TF типа TextField, не редактируемая область ввода TA типа TextArea и кнопка B типа Button. В поле TF вводится строка, после нажатия клавиши <Enter> или

щелчка кнопкой мыши по кнопке B строка переносится в область TA. После этого можно снова вводить строку в поле TF и т.д.

Здесь и при нажатии клавиши <Enter> и при щелчке кнопкой мыши возникает событие класса ActionEvent, причем оно может произойти в двух компонентах-источниках: поле TF или кнопке B. Обработка события в обоих случаях заключается в получении строки текста из поля TF (например, методом TF.getText() и помещений ее в область TA (скажем, методом TA.append()). Значит, можно написать один обработчик события ActionEvent, реализовав соответствующий интерфейс, который называется ActionListener. В этом интерфейсе всего один метод actionPerformed().

Листинг.

```
Class TextMove implements ActionListener {
    private TextField TF;
    private TextArea TA;
    TextMove(TextField TF, TextArea TA) {
        this.TF= TF; this.TA = TA;
    }
    public void actionPerformed(ActionEvent ae) {
        TA.append(tf.getText()+"\n");
    }
}
```

Обработчик событий готов. При наступлении события типа ActionEvent будет создан экземпляр класса-обработчика TextMove, конструктор получит ссылки на конкретные поля объекта-источника, метод actionPerformed(), автоматически включившись в работу, перенесет текст из одного поля в другое.

Теперь напишем класс-контейнер, в котором находятся источники TF и B события ActionEvent, и подключим к ним слушателя этого события TextMove, передав им ссылки на него методом addActionListener(), как показано в листинге.

Листинг. Обработка события ActionEvent

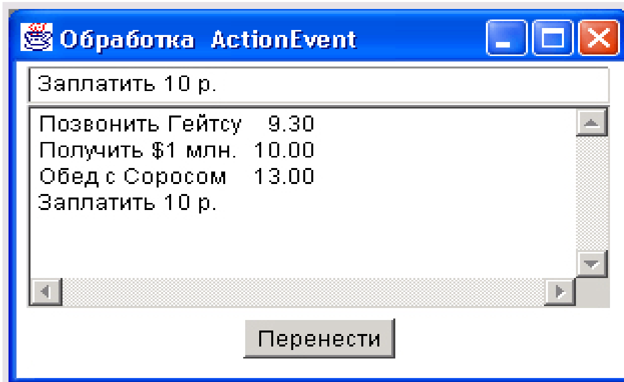
```
import j ava.awt.*;
impo rt j ava.awt.event.*;
Class MyNotebook extends Frame {
    MyNotebook(String title) {
        super(title);
        TextField TF = new TextField("Вводите текст", 50);
        add(TF, BorderLayout.NORTH);
        TextArea Tta = new TextArea();
        TA.setEditable(false);
```

```
        add(TA);
        Panel p = new Panel();
        add(p, BorderLayout.SOUTH);
        Button B = new Button("Перенести");
        p.add(B);
        TF.addActionListener(new TextMove(TF,TA));
        B.addActionListener(new TextMove(TF, TA));
        setSize(300, 200); setvisible(true);
    }
    public static void main(String[] args) {
        Frame f = new MyNotebook(" Обработка ActionEvent");
        f.addWindowListener (
            new WindowAdapter()  {
                public void windowClosing(WindowEvent ev) {
                    System.exit(0);
                }
            }
        );
    }
}
```

В листинге в методах addActionListener() создаются 2 экземпляра класса TextMove - для прослушивания поля TF и для прослушивания кнопки B. Можно создать один экземпляр класса TextMove, он будет прослушивать оба компонента:

```
    TextMove tml = new TextMove(tf, ta);
    TF.addActionListener(tml);
    B.addActionListener(tml);
```

Но в первом случае экземпляры создаются после наступления события в соответствующем компоненте, а во втором - независимо от того, наступило событие или нет, что приводит к расходу памяти, даже если событие не произошло. Решайте сами, что лучше.



16.4.33. Обработка действий мыши

Событие MouseEvent возникает в компоненте по любой из 7 причин:

- нажатие кнопки мыши - идентификатор MOUSE_PRESSED;
- отпускание кнопки мыши - идентификатор MOUSE_RELEASED;
- щелчок кнопкой мыши - идентификатор MOUSE_CLICKED (нажатие и отпускание не различаются);
- перемещение мыши - идентификатор MOUSE_MOVED;
- перемещение мыши с нажатой кнопкой - идентификатор MOUSE_DRAGGED;
- появление курсора мыши в компоненте - идентификатор MOUSE_ENTERED;
- выход курсора мыши из компонента - идентификатор MOUSE_EXITED.

Для их обработки есть 7 методов в двух интерфейсах:

```
public interface MouseListener extends EventListener {
    public void mouseClicked(MouseEvent e);
    public void mousePressed(MouseEvent e) ;
    public void mouseReleased(MouseEvent e);
    public void mouseEntered(MouseEvent e);
    public void mouseExited(MouseEvent e);
}
public interface MouseMotionListener extends EventListener {
    public void mouseDragged(MouseEvent e);
    public void mouseMoved(MouseEvent e);
}
```

Эти методы могут получить от аргумента e координаты курсора мыши в системе координат компонента методами e.getX(), e.getV(), или одним методом e.getPoint(), возвращающим экземпляр класса Point.

Двойной щелчок кнопкой мыши можно отследить методом e.getCiickCount(), возвращающим количество щелчков. При перемещении мыши возвращается 0.

Узнать, какая кнопка была нажата, можно с помощью метода e.getModifiers() класса inputEvent сравнением со следующими статическими константами класса inputEvent:

- BUTTON1_MASK - нажата первая кнопка, обычно левая;
- BUTTON2_MASK - нажата вторая кнопка, обычно средняя, или одновременно нажаты обе кнопки на двухкнопочной мыши;
- BUTTON3_MASK - нажата третья кнопка, обычно правая.

Чтобы облегчить задачу реализации интерфейсов, имеющих более одного метода, созданы классы-адаптеры.

16.4.34. Классы-адаптеры

Классы-адаптеры представляют собой пустую реализацию интерфейсов-слушателей, имеющих более одного метода. Их имена составляются из имени события и слова Adapter. Например, для действий с мышью есть два класса-адаптера. Выглядят они очень просто:

```
public abstract Class MouseAdapter implements MouseListener {
    public void mouseClicked(MouseEvent e){}
    public void mousePressed(MouseEvent e){}
    public void mouseReleased(MouseEvent e){}
    public void mouseEntered(MouseEvent e){}
    public void mouseExited(MouseEvent e){}
}
public abstract Class MouseMotionAdapter implements MouseMotionListener {
    public void mouseDragged(MouseEvent e){}
    public void mouseMoved(MouseEvent e){}
}
```

Классов-адаптеров всего 7. Кроме уже упомянутых 3 классов, это классы Component Adapter, ContainerAdapter, FocusAdapter и KeyAdapter.

16.4.35. Обработка действий клавиатуры

Событие KeyEvent происходит в компоненте по любой из 3 причин:

- нажата клавиша - идентификатор KEY_PRESSED;

- отпущена клавиша - идентификатор KEY_RELEASED;
- введен символ - идентификатор KEYJTYPED.

Последнее событие возникает из-за того, что некоторые символы вводятся нажатием нескольких клавиш, например, заглавные буквы вводятся комбинацией клавиш <Shift>+<буква>. Нажатие функциональных клавиш, например <F1>, не вызывает событие KEY_TYPED.

Обрабатываются эти события 3 методами, описанными в интерфейсе:

```
public interface KeyListener extends EventListener {
    public void keyTyped(KeyEvent e);
    public void keyPressed(KeyEvent e);
    public void keyReleased(KeyEvent e);
}
```

Аргумент (е) этих методов может дать следующие сведения.

Метод e.getKeyChar() возвращает символ Unicode типа char, связанный с клавишей. Если с клавишей не связан никакой символ, то возвращается константа CHAR_UNDEFINED.

Метод e.getKeyCode() возвращает код клавиши в виде целого числа типа int. В классе KeyEvent определены коды всех клавиш в виде констант, называемых виртуальными кодами клавиш (virtual key codes), например, VK_FI, VK_SHIFT, VK_A, VK_B, VK_PLUS. Они перечислены в документации к классу KeyEvent. Фактическое значение виртуального кода зависит от языка и раскладки клавиатуры. Чтобы узнать, какая клавиша была нажата, надо сравнить результат выполнения метода getKeyCode() с этими константами. Если кода клавиши нет, как происходит при наступлении события KEY_TYPED, то возвращается значение VK_UNDEFINED.

Чтобы узнать, не нажата ли одна или несколько клавиш-модификаторов <Alt>, <Ctrl>, <Meta>, <Shift>, надо воспользоваться унаследованным от класса inputEvent методом getModifiers() и сравнить его результат с константами ALT_MASK, CTRL_MASK, META_MASK, SHIFTJMASK. Другой способ - применить логические методы isAltDown(), isControlDown(), isMetaDown(), isShiftDown().

16.4.36. Событие TextEvent

Событие TextEvent происходит только по одной причине - изменению текста - и отмечается идентификатором TEXT_VALUE_CHANGED. Соответствующий интерфейс имеет только один метод:

```
public interface TextListener extends EventListener {
```

```
    public void textValueChanged(TextEvent e) ;
}
```

От аргумента e этого метода можно получить ссылку на объект-источник события методом getSource(), унаследованным от класса EventObject, например, так:

```
    TextComponent tc = (TextComponent)e.getSpurce();
    String s = tc.getText() ;
```

16.4.37. Обработка действий с окном

Событие windowEvent может произойти по 7 причинам:

- окно открылось - идентификатор WINDOW_OPENED;
- окно закрылось - идентификатор WINDOW_CLOSED;
- попытка закрытия окна - идентификатор WINDOW_CLOSING;
- окно получило фокус - идентификатор WINDOW_ACTIVATED;
- окно потеряло фокус - идентификатор WINDOW_DEACTIVATED;
- окно свернулось в ярлык - идентификатор WINDOW_ICONIFIED;
- окно развернулось - идентификатор WINDOW_DEICONIFIED.

Соответствующий интерфейс содержит 7 методов:

```
public interface WindowListener extends EventListener
{
    public void windowOpened(WindowEvent e);
    public void windowClosing(WindowEvent e);
    public void windowClosed(WindowEvent e);
    public void windowIconified(WindowEvent e);
    public void windowDeiconified(WindowEvent e);
    public void windowActivated(WindowEvent e);
    public void windowDeactivated(WindowEvent e);
}
```

Аргумент (e) этих методов дает ссылку типа window на окно-источник методом e.getWindow().

Чаще всего эти события используются для перерисовки окна методом repaint() при изменении его размеров и для остановки приложения при закрытии окна.

16.4.38. Событие ComponentEvent

Данное событие происходит в компоненте по 4 причинам:

- компонент перемещается - идентификатор COMPONENT_MOVED;

- компонент меняет размер - идентификатор COMPONENT_RESIZED;
- компонент убран с экрана - идентификатор COMPONENT_HIDDEN;
- компонент появился на экране - идентификатор COMPONENT_SHOWN.

Соответствующий интерфейс содержит описания 4 методов:

```
public interface ComponentListener extends EventListener
{
    public void componentResized(ComponentEvent e);
    public void componentMoved(Comp©nentEvent e);
    public void componentShown(ComponentEvent e);
    public void componentHidden(ComponentEvent e);
}
```

Аргумент (е) методов этого интерфейса предоставляет ссылку на компонент-источник события методом e.getComponent().

16.4.39. Событие ContainerEvent

Это событие происходит по 2 причинам:

- в контейнер добавлен компонент - идентификатор COMPONENT_ADDED;
- из контейнера удален компонент - идентификатор COMPONENT_REMOVED.

Этим причинам соответствуют методы интерфейса:

```
public interface ContainerListener extends EventListener
{
    public void componentAdded(ContainerEvent e) ;
    public void componentRemoved(ContainerEvent e);
}
```

Аргумент (е) предоставляет ссылку на компонент, чье добавление или удаление из контейнера вызвало событие, методом e.getChild(), и ссылку на контейнер - источник события методом e.getContainer(). Обычно при наступлении данного события контейнер перемещает свои компоненты.

16.4.40. Событие FocusEvent

Событие возникает в компоненте, когда он^

- получает фокус ввода - идентификатор FOCUS_GAINED,
- или теряет фокус - идентификатор FOCUS_LOST.

Соответствующий интерфейс:

```
public interface FocusListener extends EventListener {
    public void focusGainedtFocusEvent e) ;
    public void focusLost(FocusEvent e) ;
}
```

Обычно при потере фокуса компонент перечерчивается бледным цветом, для этого применяется метод brighter() класса Color, при получении фокуса становится ярче, что достигается применением метода darker(). Это приходится делать самостоятельно при создании своего компонента.

16.4.41. Событие ItemEvent

Это событие возникает при выборе или отказе от выбора элемента в списке List, Choice или флажка Checkbox и отмечается идентификатором ITEM_STATE_CHANGED.

Соответствующий интерфейс очень прост:

```
public interface ItemListener extends EventListener {
    void itemStateChanged(ItemEvent e);
}
```

Аргумент (е) предоставляет ссылку на источник методом e.getIemsElectable(), ссылку на выбранный пункт методом e.getItem() в виде Object.

Метод e.getStateChange() позволяет уточнить, что произошло:

- значение SELECTED указывает на то, что элемент был выбран,
- значение DESELECTED - произошел отказ от выбора.

16.4.42. Событие AdjustmentEvent

Это событие возникает для полосы прокрутки Scroiibar при всяком изменении ее бегунка и отмечается идентификатором ADJUSTMENT_VALUE_CHANGED.

Соответствующий интерфейс описывает один метод:

```
public interface AdjustmentListener extends EventListener {
    public void adjustmentValueChanged(AdjustmentEvent e);
}
```

Аргумент (е) этого метода предоставляет ссылку на источник события методом e.getAdjustable(), текущее значение положения движка полосы прокрутки методом e.getValue(), и способ изменения его значения методом e.getAdjustmentType(), возвращающим следующие значения:

- UNIT__INCREMENT - увеличение на одну единицу;
- UNIT_DECREMENT - уменьшение на одну единицу;

- BLOCK_INCREMENT - увеличение на один блок;
- BLOCK_DECREMENT - уменьшение на один блок;
- TRACK - процес передвижения бегунка полосы прокрутки.

В начале этой главы мы привели пример класса TextMove, слушающего сразу два компонента: поле ввода TF типа TextFieid и кнопку В типа Button.

Чаще встречается обратная ситуация - несколько слушателей следят за одним компонентом. К каждому компоненту можно присоединить сколько угодно слушателей одного и того же события или разных типов событий. Однако при этом не гарантируется какой-либо определенный порядок их вызова, хотя чаще всего слушатели вызываются в порядке написания методов addXxxListener().

Если нужно задать определенный порядок вызовов слушателей для обработки события, то придется обращаться к ним друг из друга или создавать объект, вызывающий слушателей в нужном порядке.

Ссылки на присоединенные методами addxxxbistener() слушатели можно было бы хранить в любом классе-коллекции, например, vector, но в пакет java.awt специально для этого введен класс AWTEventMuitiCaster. Он реализует все 11 интерфейсов xxxListener, значит, сам является слушателем любого события. Основу класса составляют своеобразные статические методы add(), написанные для каждого типа событий, например:

```
add(ActionListener a, ActionListener b)
```

Своеобразие этих методов двоякое: они возвращают ссылку на тот же интерфейс, в данном случае, ActionListener, и присоединяют объект (а) к объекту (b), создавая совокупность слушателей одного и того же типа. Это позволяет использовать их наподобие операций а += b. Заглянув в исходный текст класса Button, вы увидите, что метод addActionListener() очень прост:

```
public synchronized void addActionListener(ActionListener 1) {
    if (1 = null) { return; }
    actionListener = AWTEventMuiticaster.add(actionListener, 1);
    newEventsOnly = true;
}
```

Он добавляет к совокупности слушателей actionListener нового слушателя 1.

Для событий типа inputEvent, а именно, KeyEvent и MouseEvent, есть возможность прекратить дальнейшую обработку события методом consume(). Если записать вызов этого метода в класс-слушатель, то ни peer-объекты, ни следующие слушатели не будут обрабатывать событие. Этим способом обычно пользуются, чтобы отменить стандартные действия компонента, например, "вдавливание" кнопки.

16.4.43. Диспетчеризация событий

Если вам понадобится обработать просто действие мыши, не важно, нажатие это, перемещение или еще что-нибудь, то придется включать эту обработку во все 7 методов двух классов-слушателей событий мыши.

Эту работу можно облегчить, выполнив обработку не в слушателе, а на более ранней стадии. Дело в том, что прежде чем событие дойдет до слушателя, оно обрабатывается несколькими методами.

Чтобы в компоненте произошло событие AWT, должно быть выполнено хотя бы одно из двух условий: к компоненту присоединен слушатель или в конструкторе компонента определена возможность появления события методом enableEvents(). В аргументе этого метода через операцию побитового сложения перечисляются константы класса AWTEvent, задающие события, которые могут произойти в компоненте, например:

```
    enableEvents(AWTEvent.MOUSE_MOTION_EVENT_MASK |
        AWTEvent.MOUSE_EVENT_MASK I AWTEvent.KEY_EVENT_MASK)
```

При появлении события создается объект соответствующего класса xxxEvent. Метод dispatchEvent() определяет, где появилось событие — в компоненте или одном из его подкомпонентов, - и передает объект-событие методу processEvent() компонента-источника.

Метод processEvent() определяет тип события и передает его специализированному методу processxxxEvent(). Вот начало этого метода:

```
protected void processEvent(AWTEvent e) {
    if (e instanceof FocusEvent) {
        processFocusEvent((FocusEvent)e);
    }
    else if (e instanceof MouseEvent) {
            switch (e.getld() {
                case MouseEvent.MOUSE_PRESSED:
                case MouseEvent.MOUSE_RELEASED:
                case MouseEvent.MOUSE_CLICKED:
                case MouseEvent.MOUSE_ENTERED:
                case MouseEvent.MOUSE_EXITED:
                    processMouseEvent((MouseEvent)e);
                    break/case MouseEvent.MOUSE_MOVED:
                case MouseEvent.MOUSE_DRAGGED:
                    processMouseMotionEvent((MouseEvent)e);
                    break;
            }
```

```
        }
    else if (e instanceof KeyEvent) {
        processKeyEvent((KeyEvent)e);
    }
//...
```

Затем в дело вступает специализированный метод, например, processKeyEvent(). Он-то и передает объект-событие слушателю. Вот исходный текст этого метода:

```
protected void processKeyEvent(KeyEvent e) {
    KeyListener listener = keyListener;
    if (listener != null) {
        int id = e.getlDf);
        switch(id) {
            case KeyEvent.KEYJTYPED: listener.keyTyped(e);  break;
            case KeyEvent.KEY_PRESSED: listener.keyPressed(e);  break;
            case KeyEvent.KEY_RELEASED: listener.keyReleased(e);  break;
        }
    }
}
```

Из этого описания видно, что если вы хотите обработать любое событие типа AWTEvent, то вам надо переопределить метод processEvent(), а если более конкретное событие, например, событие клавиатуры, - переопределить более конкретный метод processKeyEvent(). Если вы не переопределяете весь метод целиком, то не забудьте в конце обратиться к методу суперкласса, например, super.processKeyEvent(e);

Замечание. Не забывайте обращаться к методу processXxxEvent() суперкласса.

16.4.44. Создание собственного события

Вы можете создать собственное событие и определить источник и условия его возникновения.

В листинге приведен пример создания события MyEvent. Событие MyEvent говорит о начале работы программы (START) и окончании ее работы (STOP).

Листинг. Создание собственного события

```
// 1. Создаем свой класс события:
public Class MyEvent extends java.util.EventObjectf protected int id;
public static final int START = 0, STOP = 1;
```

```
public MyEvent(Object source, int id) {
    super(source);
    this.id = id;
}
public int getID() { return id; }
// 2. Описываем Listener:
public interface MyListener extends java.util.EventListener {
    public void start{MyEvent e);
    public void stop(MyEvent e);
}
// 3. В теле нужного класса создаем метод fireEvent()
protected Vector listeners = new Vector();
public void fireEvent( MyEvent e) {
    Vector list = (Vector) listeners.clone();
    for (int i = 0; i < list.sizeO; i++) {
        MyListener listener = (MyListener) list.elementAt(i);
        switch(e.getId() {
            case MyEvent.START: listener.start(e); break;
            case MyEvent.STOP: listener.stop(e); break;
        }
    }
}
```

Все, теперь при запуске программы делаем

```
fireEvent(thi s, MyEvent.START);
```

а при окончании

```
fireEvent(this, MyEvent.STOP);
```

При этом все зарегистрированные слушатели получат экземпляры событий.

16.5. Библиотека Swing

Выше мы подробно рассмотрели возможности графической библиотеки AWT. Там же мы заметили, что в состав Java 2 SDK входит еще одна графическая библиотека, **Swing**, с более широкими возможностями, чем AWT. Фирма SUN настоятельно рекомендует использовать Swing, а не AWT, но, во-первых, Swing требует больше ресурсов, что существенно для российского разработчика, во-вторых, большинство браузеров не имеет в своем составе Swing. В-третьих, удобнее сначала познакомиться с библиотекой AWT, а уже потом изучать Swing.

Все примеры графических программ, приведенные для AWT, будут выполняться методами библиотеки Swing после небольшой переделки:

- Добавьте в заголовок строку import javax.swing.*;.
- Поменяйте Frame на JFrame, Applet на JApplet, Component на JComponent, Panel на JPanei. Не расширяйте свои классы от класса canvas, используйте jpanel или другие контейнеры Swing.
- Замените компоненты AWT на близкие к ним компоненты Swing. Чаще всего надо просто приписать букву J: JButton, JcheckBox, JDialog, jList, JMenu и т.д. Закомментируйте временно строку import java.awt.*; и попробуйте откомпилировать программу. Компилятор покажет, какие компоненты требуют замены.
- Включите в конструктор класса, расширяющего JFrame, строку Container c = getContentPane (); и располагайте все компоненты в контейнере с, Т.е. пишите c.add(), c.setLayout ().
- Класс jFrame содержит средства закрытия своего окна, надо только настроить их. Вы можете убрать addwindowListener(...) и включить в конструктор обращение К методу setDefaultCloseQperation(JFrame.EXITJB_CLOSE).
- В прямых подклассах класса jpanei замените метод paint о на paintcomponent () и удалите метод update о. Класс jpanei автоматически производит двойную буферизацию и надобности в методе update о больше нет. Уберите весь код двойной буферизации. В начало метода paintcomponent () включите обращение super.paintcomponent (g). Из подклассов классов JFrame, joialog, JAppiet метод paintcomponent () надо переместить в другие компоненты, например, JButton, JLabel, jpanei.
- Используйте вместо класса image класс imageicon. Конструкторы этого класса выполнят необходимое преобразование. Класс imageicon автоматически применяет методы класса MediaTracker для ожидания окончания загрузки.

- При создании апплетов расширением класса JAppiet не забывайте, что в классе Applet менеджером размещения по умолчанию служит класс FiowLayout, а в классе JAppiet менеджер размещения по умолчанию BorderLayout.

Откомпилировав и запустив измененную программу, вы увидите, что ее внешний вид изменился, чаще всего не в лучшую сторону. Теперь надо настроить компоненты Swing. Библиотека Swing предоставляет для этого широчайшие возможности.

16.6. Компоненты JavaBeans

Многие программисты предпочитают разрабатывать приложения с графическим интерфейсом пользователя с помощью визуальных средств разработки: JBuilder, Visual Age for Java, Visual Cafe и др. Эти средства позволяют помещать компоненты в контейнер графически, с помощью мыши.

Для того чтобы компонент можно было применять в таком визуальном средстве разработки он должен обладать дополнительными качествами. У него должен быть ярлык, помещаемый на панель компонентов. Среди полей компонента должны быть выделены свойства (properties), которые будут показаны в окне свойств Следует определить методы доступа getXxx(), setXxx() к каждому свойству

Компонент, снабженный этими и другими необходимыми качествами, в технологии Java называется компонентом **JavaBean**. В него может входить один или несколько классов. Как правило, файлы этих классов упаковываются в jar-архив и отмечаются в файле MANIFEST.MF как Java-Bean: True.

Все компоненты AWT и Swing являются компонентами JavaBeans. Если вы создаете свой графический компонент по правилам, изложенным выше, то вы тоже получаете свой JavaBean. Но для того чтобы не упустить каких-либо важных качеств JavaBean, лучше использовать для их разработки специальные средства.

Фирма SUN поставляет набор необходимых утилит и классов BDK (Beans Development Kit) для разработки JavaBeans. Так же, как и SDK, этот набор хранится на сайте фирмы в виде самораспаковывающегося архива. Его содержимое распаковывается в один каталогю После перехода в каталог и запуска файла run.bat (или run.sh в UNIX) открываются несколько окон утилиты BeanBox, работа с которой ведется по тому же принципу, что и в визуальных средствах разработки.

Визуальные средства разработки - это не основное применение JavaBeans. Главное достоинство компонентов, оформленных как JavaBeans, в том, что они без труда встраиваются в любое приложение. Более того, приложение можно

собрать из готовых JavaBeans как из строительных блоков, остается только настроить их свойства.

Специалисты пророчат большое будущее компонентному программированию. Они считают, что скоро будут созданы тысячи компонентов JavaBeans на все случаи жизни и программирование сведется к поиску в Internet нужных компонентов и сборке из них приложения.

17. Апплеты

17.1. Основы

До сих пор мы создавали приложения (applications), работающие самостоятельно (standalone) в JVM под управлением графической оболочки операционной системы. Эти приложения имели собственное окно верхнего уровня типа Frame, зарегистрированное в оконном менеджере (window manager) графической оболочки.

Кроме приложений, язык Java позволяет создавать апплеты (applets). Это программы, работающие в среде другой программы - браузера. Апплеты не нуждаются в окне верхнего уровня - им служит окно браузера. Они не запускаются JVM - их загружает браузер, который сам запускает JVM для выполнения апплета. Эти особенности отражаются на написании программы апплета.

С точки зрения языка Java, апплет - это всякое расширение класса Applet, который, в свою очередь, расширяет класс Panel. Таким образом, апплет — это панель специального вида, контейнер для размещения компонентов с дополнительными свойствами и методами. Менеджером размещения компонентов по умолчанию, как и в классе Panel, служит FiowLayout. Класс Applet находится в пакете java.applet, в котором кроме него есть только 3 интерфейса, реализованные в браузере. Надо заметить, что не все браузеры реализуют эти интерфейсы полностью.

Поскольку JVM не запускает апплет, отпадает необходимость в методе main(), его нет в апплетах.

В апплетах редко встречается конструктор. Дело в том, что при запуске первого создается его контекст. Во время выполнения конструктора контекст еще не сформирован, поэтому не все начальные значения удается определить в конструкторе.

Начальные действия, обычно выполняемые в конструкторе и методе main(), в апплете записываются в метод init() класса Applet. Этот метод автоматически запускается исполняющей системой Java браузера сразу же после загрузки апплета. Вот как он выглядит в исходном коде класса Applet:

```
public void init(){}
```

Метод init() не имеет аргументов, не возвращает значения и должен переопределяться в каждом апплете - подклассе класса Applet. Обратные действия - завершение работы, освобождение ресурсов - записываются при необходимости в метод destroy(), тоже выполняющийся автоматически при выгрузке апплета. В классе Applet есть пустая реализация этого метода.

Кроме методов init() и destroy() в классе Applet присутствуют еще 2 пустых метода, выполняющихся автоматически. Браузер должен обращаться к методу start() при каждом появлении апплета на экране и обращаться к методу stop(), когда апплет уходит с экрана. В методе stop() можно определить действия, приостанавливающие работу апплета, в методе start() - возобновляющие ее. Надо сразу же заметить, что не все браузеры обращаются к этим методам как должно.

Все эти методы в апплете необязательны. В листинге записан простейший апплет, выполняющий вечную программу HelloWorld.

Листинг. Апплет HelloWorld

```
import j ava.awt.*;
import j ava.applet.*;
public Class HeiioWorid extends Applet {
    public void paint(Graphics g) {
        g.drawstring("Hello, XXI century World 1 ", 10, 30);
    }
}
```

Эта программа записывается в файл HelloWorld.java и компилируется как обычно: javac HelloWorld.java. Компилятор создает файл HelloWorld.Class, но воспользоваться для его выполнения интерпретатором java теперь нельзя - нет метода main(). Вместо интерпретации надо дать указание браузеру для запуска апплета.

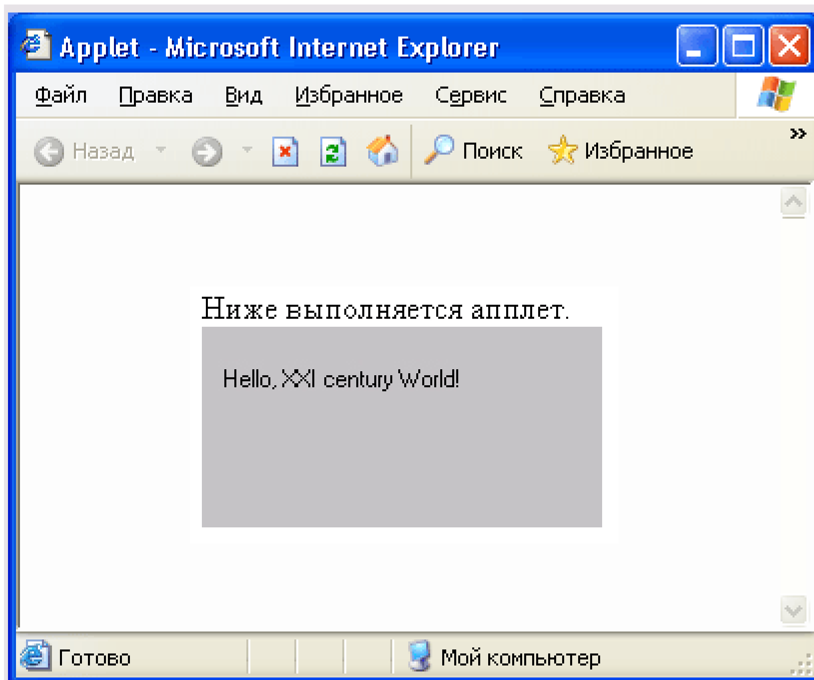
Все указания браузеру даются пометками, тегами (tags), на языке HTML (HyperText Markup Language). В частности, указание на запуск апплета дается в теге <applet>. В нем обязательно задается имя файла с классом апплета параметром code, ширина width и высота height панели апплета в пикселах. Полностью текст HTML для нашего апплета приведен в листинге.

Листинг. Файл HTML для загрузки апплета HelloWorid

```
<html>
<head><title> Applet</title></head>
<body>
    Ниже выполняется апплет.<br>
    <applet code = "HeiioWorid.Class" width = "200" height = "100">
    </applet>
</body>
</html>
```

Этот текст заносится в файл с расширением html или htm, например. HelloWorld.html. Имя файла произвольно, никак не связано с апплетом или классом апплета.

Оба файла - HelloWorld.html и HelloWorld.Class - помещаются в один каталог на сервере, и файл HelloWorld.html загружается в браузер, который может находиться в любом месте Internet. Браузер, просматривая HTML-файл, выполнит тег <appLet> и загрузит апплет. После загрузки апплет появится в окне браузера.



В этом простом примере можно заметить еще 2 особенности апплетов. Во-первых, размер апплета задается не в нем, а в теге <applet>. Это очень удобно, можно менять размер апплета, не компилируя его заново. Можно организовать апплет невидимым, сделав его размером в один пиксел. Кроме того, размер апплета разрешается задать в процентах по отношению к размеру окна браузера, например,

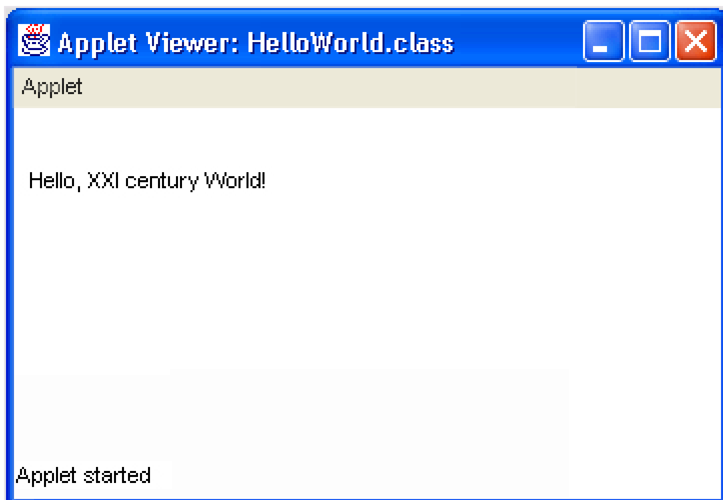
```
<applet code = "HelloWorld.Class" width = "100%" height = "100%">
```

Во-вторых, как видно на риунке, у апплета серый фон. Такой фон был в первых браузерах, и апплет не выделялся из текста в окне браузера. Теперь в браузе-

рах принят белый фон, его можно установить обычным для компонентов методом setBackground(Color.white), обратившись к нему в методе init().

В состав JDK любой версии входит программа appletViewer. Это простейший браузер, предназначенный для запуска апплетов в целях отладки. Если под рукой нет Internet-браузера, можно воспользоваться им. AppLetviewer запускается из командной строки:

```
appletViewer HelloWorld.html
```



Приведем более сложный пример. Апплет showWindow создает окно someWindow типа Frame, в котором расположено поле ввода типа TextFieid. В него вводится текст, и после нажатия клавиши <Enter> переносится в поле ввода апплета. В апплете присутствует кнопка. После щелчка кнопкой мыши по ней окно someWindow то скрывается с экрана, то вновь появляется на нем. То же самое должно происходить при уходе и появлении апплета в окне браузера в результате прокрутки, как записано в методах stop() и start()/

<u>Листинг. Апплет, создающий окно</u>

```
// Файл ShowWindow.java
import j ava.awt.*;
import j ava.awt.event.*;
import java.applet.*;
public Class ShowWindow extends Applet {
    private SomeWindow sw = new SomeWindow();
    private TextField tf = new TextField(30);
```

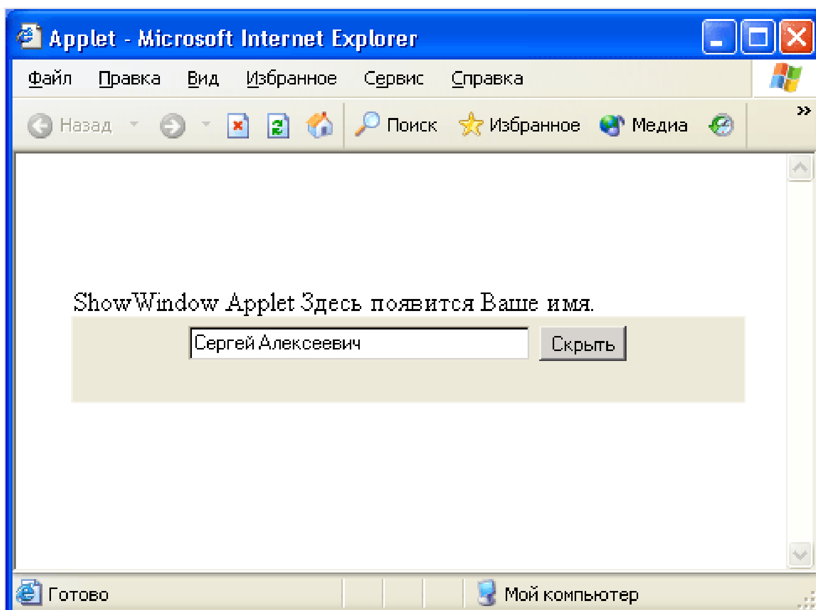
```
    private Button b = new Button("Скрыть");
    public void init() {
        add(tf); add(b); sw.pack();
        b.addActionListener(new ActShow());
        sw.tf.addActionListener(new ActShow());
    }
    public void start() {
        sw.setVisible(true);
    }
    public void stop() {
        sw.setVisible(false);
    }
public void destroy() {
        sw.disposeO ; sw = tf = b = null;
    }
public Class ActShow implements ActionListener {
        public void actionPerformed(ActionEvent ae) {
        if (ae.getSource() = sw.tf) tf.setText(sw.tf.getText() ) ;
        else if (b.getActionCoiranand() == "Показать") {
                sw.setVisible(true);
                b.setLabel("Скрыть") ;
            }
            else {
                sw.setVisible(false);
                b.setLabel("Показать");
            }
        }
    }
}
Class SomeWindow extends Frame {
    public TextField tf = new TextField(50);
    SomeWindow() {
        super(" Окно ввода");
        add(new Label("Введите, пожалуйста, свое имя"), "North");
        add(tf, "Center");
    }
}
```

Листинг. Файл ShowWindow.html

```
<html>
<headxtitle> ShowWindow Applet</title></head>
```

```
<body>
    Здесь появится Ваше имя.<br>
    <applet code = "ShowWindow.Class" width = "400" height = "50">
    </applet>
</body>
</html>
```

Результат



Следует еще сказать, что, начиная с версии HTML 4.0, есть тег <Object>, предназначенный для загрузки и апплетов, и других объектов, например, ActiveX. Кроме того, некоторые браузеры могут использовать для загрузки апплетов тег <embed>.

Сведения об окружении апплета.

Метод getCodeBase() возвращает URL-адрес каталога, в котором лежит файл класса апплета.

Метод getDocumentBase() возвращает URL-адрес каталога, в котором лежит HTML-файл, вызвавший апплет.

Браузер реализует интерфейс AppletContext, находящийся в пакете java.applet. Апплет может получить ссылку на этот интерфейс методом getAppletContext().

С помощью методов getApplet(String name) и getApplets() интерфейса AppletContext можно получить ссылку на указанный аргументом name апплет или на все апплеты, загруженные в браузер.

Метод showDocument(URL address) загружает в браузер HTML-файл с адреса address.

Метод showDocument (URL address, String target) загружает файл во фрейм, указанный вторым аргументом target. Этот аргумент может принимать следующие значения:

- _seif - то же окно и тот же фрейм, в котором работает апплет;
- _parent - родительский фрейм апплета;
- _top - фрейм верхнего уровня окна апплета;
- _biank - новое окно верхнего уровня;
- name -фрейм или окно с именем name, если оно не существует, то будет создано.

17.2. Изображение и звук в апплетах

Изображение в Java - это объект класса Image, представляющий прямоугольный массив пикселов. Его могут показать на экране логические методы drawImage() класса Graphics. Мы рассмотрим их подробно в следующей главе, а пока нам понадобятся 2 логических метода:

- drawImage(Image img, int x, int y, ImageObserver obs)
- drawImage(Image img, int x, int y, int width, int height, ImageObserver obs)

Методы начинают рисовать изображение, не дожидаясь окончания загрузки изображения img. Более того, загрузка не начнется, пока не вызван метод drawImage(). Методы возвращают false, пока загрузка не закончится.

Аргументы (x, y) задают координаты левого верхнего угла изображения img; width и height - ширину высоту изображения на экране; obs - ссылку на объект, реализующий интерфейс ImageObserver, следящий за процессом загрузки изображения. Последнему аргументу можно дать значение this.

Первый метод задает на экране такие же размеры изображения, как и у объекта класса Image, без изменений. Получить эти размеры можно методами getWidth(), getHeight{) класса Image.

Интерфейс ImageObserver, реализованный классом Component, а значит, и классом Applet, описывает только один логический метод imageUpdate (), выполняющийся при каждом изменении изображения. Именно этот метод побуждает перерисовывать компонент на экране при каждом его изменении. Посмотрим, как его можно использовать в процессе загрузки файлов из Internet.

Слежение за процессом загрузки.

Если вы хотя бы раз видели, как изображение загружается из Internet, то заметили, что оно появляется на экране по частям по мере загрузки. Это происходит в том случае, когда системное свойство awt.Image.IncrementalDraw имеет значение true.

При поступлении каждой порции изображения браузер вызывает логический метод imageUpdate() интерфейса ImageObserver. Аргументы этого метода содержат информацию о процессе загрузки изображения img. Рассмотрим их:

imageUpdate(Image img, int status, int x, int y, int width, int height);

Аргумент status содержит информацию о загрузке в виде одного целого числа, которое можно сравить со следующими константами интерфейса ImageObserver:

- WIDTH - ширина уже загруженной части изображения известна, и может быть получена из аргумента width;
- HEIGHT - высота уже загруженной части изображения известна, и может быть получена из аргумента height;'
- PROPERTIES - свойства изображения уже известны, их можно получить методом getProperties{) класса Image;
- SOMEBITS - получены пикселы, достаточные для рисования масштабированной версии изображения; аргументы x, y, width, height определены;
- FRAMEBITS - получен следующий кадр изображения, содержащего несколько кадров; аргументы x, y, width, height не определены;
- ALLBITS - все изображение получено, аргументы x, y, width, height не содержат информации;
- ERROR -загрузка прекращена, рисование прервано, определен бит ABORT ;
- ABORT - загрузка прервана, рисование приостановлено до прихода следующей порции изображения.

Вы можете переопределить этот метод в своем апплете и использовать его аргументы для слежения за процессом загрузки и определения момента полной загрузки.

Другой способ отследить окончание загрузки - воспользоваться методами класса MediaTracker. Они позволяют проверить, не окончена ли загрузка, или приостановить работу апплета до окончания загрузки. Один экземпляр класса MediaTracker может следить за загрузкой нескольких зарегистрированных в нем изображений.

17.3. Класс MediaTracker

Сначала конструктором MediaTracker(Component comp) создается объект класса для указанного аргументом компонента. Аргумент конструктора чаще всего this.

Затем методом addImage(Image img, int id) регистрируется изображение img под порядковым номером id. Несколько изображений можно зарегистрировать под одним номером.

После этого логическими методами checkId(int id), checkId(int id, Boolean load) и checkAll() проверяется, загружено ли изображение с порядковым номером id или все зарегистрированные изображения. Методы возвращают true, если изображение уже загружено, false - в противном случае. Если аргумент load равен true, то производится загрузка всех еще не загруженных изображений.

Методы statusId(int id), statusId(int id, Boolean load) и statusAll() возвращают целое число, которое можно сравнить со статическими константами COMPLETE, ABORTED, ERRORED.

Наконец, методы waitForId(int id) и waitForAll() ожидают окончания загрузки изображения.

Изображение, находящееся в объекте класса Image можно создать непосредственно по пикселам, а можно получить из графического файла, типа GIF или JPEG, одним из двух методов класса Applet:

- getImage(URL address) - задается URL-адрес графического файла;
- getImage(URL address, String fileName) - задается адрес каталог address и имя графического файла filename.

Аналогично, звуковой файл в апплетах представляется в виде объекта, реализующего интерфейс AudioCiip, и может быть получен из файла типа AU, AIFF, WAVE или MIDI одним из 3 методов класса Applet с такими же аргументами:

getAudioClip(URL address)
getAudioClip(URL address, String fileName)
newAudioClip(URL address)

Последний метод статический, его можно использовать не только в апплетах, но и в приложениях.

Интерфейс AudioClip из пакета java.applet очень прост. В нем всего 3 метода без аргументов. Метод play() проигрывает мелодию один раз. Метод loop() бесконечно повторяет мелодию. Метод stop() прекращает проигрывание.

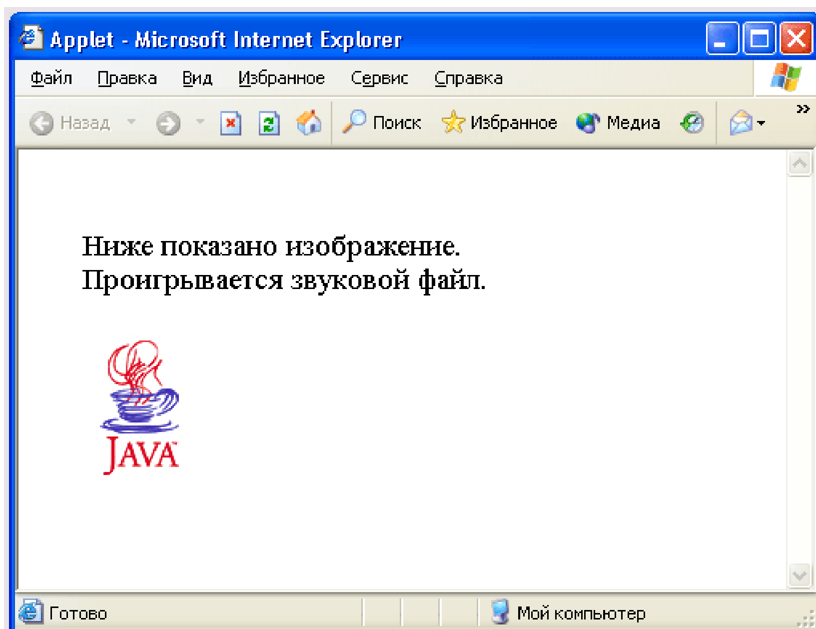
Этот интерфейс реализуется браузером. Конечно, перед проигрыванием звуковых файлов браузер должен быть связан со звуковой системой компьютера.

В листинге приведен простой пример загрузки изображения и звука из файлов, находящихся в том же каталоге, что и HTML-файл.

Листинг. Звук и изображение в апплете

```
import java.applet.*;
import j ava.awt.*;
import java.awt.image.*;
public Class SimpleAudioImage extends Applet {
    private Image img;
    private Audioclip ac;
    public void init()     {
        img = getImage(getDocumentBase(), "javalogo52x88.gif");
        ac = getAudioClip(getDocumentBase(), "yesterday.au"); }
        public void start (){ ac.loop();
    }
    public void paint(Graphics g) {
        int w = img.getWidthfthis), h = img.getHeight(this);
        g.drawImage(img, 0, 0, 2 * w, 2 * h, this); }
        public void stop() { ac.stop();
    }
}
```

Результат



Как видите, апплету в браузере позволено очень немного. Это не случайно. Апплет, появившийся в браузере откуда-то из Internet, может натворить много бед. Он может быть вызван из файла с увлекательным текстом, невидимо обыскать файловую систему и похитить секретные сведения, или, напротив, открыть окно, неотличимое от окна, в которое вы вводите пароль, и перехватить его.

Поэтому браузер сообщает при загрузке апплета: "Applet started", а в строке состояния окна, открытого апплетом, появляется надпись: "Warning: Applet Window". Но это не единственная защита от апплета. Рассмотрим данную проблему подробнее.

17.4. Защита от апплета

Браузер может вообще отказаться от загрузки апплетов. В Internet Explorer это делается с помощью команды Tools=>Internet Options=>Security.

Если браузер загружает апплет, то создает ему ограничения, так называемую "песочницу" (sandbox), в которой резвится апплет, но выйти из которой не может. Каждый браузер создает свои ограничения, но обычно они заключаются в том, что апплет:

- не может обращаться к файловой системе машины, на которой он выполняется, даже для чтения файлов или просмотра каталогов;
- может связаться по сети только с тем сайтом, с которого он был загружен;
- не может прочитать системные свойства;
- не может печатать на принтере, подключенном к тому компьютеру, на котором он выполняется;
- не может воспользоваться буфером обмена (clipboard); не может запустить приложение методом exec();
- не может использовать "родные" методы или загрузить библиотеку методом load();
- не может остановить JVM методом exit();
- не может создавать классы в пакетах java.*, а классы пакетов sun.* не может даже загружать.

Браузеры могут усилить или ослабить эти ограничения, например, разрешить локальным апплетам, загруженным с той же машины, где они выполняются, доступ к файловой системе. Наименьшие ограничения имеют доверенные (trusted) апплеты, снабженные электронной подписью с помощью классов из пакетов java.security.*.

При создании приложения, загружающего апплеты, необходимо обеспечить средства проверки апплета и задать ограничения. Их предоставляет класс securityManager. Экземпляр этого класса или его наследника устанавливается в JVM при запуске виртуальной машины статическим методом setSecurityManager(SecurityManager sm) класса System. Обычные приложения не могут использовать данный метод.

Каждый браузер расширяет класс SecurityManager по-своему, устанавливая те или иные ограничения. Единственный экземпляр этого класса создается при запуске JVM в браузере и не может быть изменен.

17.5. Заключение

Апплеты были первоначальным практическим применением Java. За первые 2 года существования Java были написаны тысячи очень интересных и красивых апплетов, ожививших WWW. Масса апплетов разбросана по Internet, хорошие примеры апплетов собраны в JDK в каталоге demo\applets.

В JDK вошел целый пакет java.applet, в который фирма SUN собиралась заносить классы, развивающие и улучшающие апплеты.

С увеличением скорости и улучшением качества компьютерных сетей значение апплетов сильно упало. Теперь вся обработка данных, прежде выполняе-

мая апплетами, переносится на сервер, браузер только загружает и показывает результаты этой обработки, становится "тонким клиентом".

С другой стороны, появилось много специализированных программ, в том числе написанных на Java, загружающих информацию из Internet. Такая возможность есть сейчас у всех музыкальных и видеопроигрывателей.

Фирма SUN больше не развивает пакет java.applet. В нем так и остался 1 класс и 3 интерфейса. В библиотеку Swing вошел класс JApplet, расширяющий класс Applet. В нем есть дополнительные возможности, например, можно установить систему меню. Он способен использовать все классы библиотеки Swing. Но большинство браузеров еще не имеют Swing в своем составе, поэтому приходится загружать классы Swing с сервера или включать их в jar-архив вместе с классами апплета.

18. Изображение и звук

18.1. Введение

Как: уже упоминалось в предыдущей главе, изображение в Java - это объект класса Image. Там же показано, как в апплетах применяются методы getImage() для создания этих объектов из графических файлов.

Приложения тоже могут применять аналогичные методы getImage() класса Toolkit из пакета java.awt с одним аргументом типа String или URL. Обращение к этим методам из компонента выполняется через метод getToolkit() класса Component и выглядит так:

```
Image img = getToolkit().getImage("C:\\images\\Ivanov.gif");
```

В общем случае обращение можно сделать через статический метод getDefaultToolkit() класса Toolkit:

```
Image img = Toolkit.getDefaultToolkit().getImage(" C:\\images\\Ivanov.gif ");
```

Но, кроме этих методов, класс Toolkit содержит 5 методов createImage(), возвращающих ссылку на объект типа image:

- createImage(String filsName) - создает изображение из содержимого графического файла filename ;
- createImage(URL address) - создает изображение из содержимого графического файла по адресу address ;
- createImage(byte [] imageData) - создает изображение из массива байтов imageData, данные в котором должны иметь формат GIF или JPEG;
- createImage(byte [] imageData, int offset, int length) - создает изображение из части массива imageData, начинающейся с индекса offset длиной length байтов;
- createImage(ImageProducer producer) - создает изображение, полученное от поставщика producer.

Последний метод есть и в классе Component. Он использует модель "поставщик-потребитель" и требует подробного объяснения.

18.2. Модель обработки "поставщик-потребитель"

Очень часто изображение перед выводом на экран подвергается обработке: меняются цвета отдельных пикселов или целых участков изображения, выделяются и преобразуются какие-то фрагменты изображения.

В библиотеке AWT применяются 2 модели обработки изображения. Одна модель реализует давно известную в программировании общую модель "постав-

щик-потребитель" (Producer-Consumer). Согласно этой модели один объект, "поставщик", генерирует сам или преобразует полученную из другого места продукцию, в данном случае, набор пикселов, и передает другим объектам. Эти объекты, "потребители", принимают продукцию и тоже преобразуют ее при необходимости. Только после этого создается объект класса Image и изображение выводится на экран. У одного поставщика может быть несколько потребителей, которые должны быть зарегистрированы поставщиком. Поставщик и потребитель активно взаимодействуют, обращаясь к методам друг друга.

В AWT эта модель описана в двух интерфейсах: ImageProducer bImageConsumer пакета j ava. awt. image.

Интерфейс ImageProducer описывает 5 методов:

- addConsumer(ImageConsumer ic) - регистрирует потребителя ic;
- removeConsumer(ImageConsumer ic) - отменяет регистрацию;
- isConsumer(ImageConsumer ic) - логический метод, проверяет, зарегистрирован ли потребитель ic;
- startProduction (ImageConsumer ic) - регистрирует потребителя ic И НЭ-чинает поставку изображения всем зарегистрированным потребителям;
- requestTopDownLeftRightResend (ImageConsumer ic) - используется потребителем для того, чтобы затребовать изображение еще раз в порядке "сверху-вниз, слева-направо" для методов обработки, применяющих именно такой порядок.

С каждым экземпляром класса Image связан объект, реализующий интерфейс ImageProducer. Его можно получить методом getSource() класса Image.

Самая простая реализация интерфейса ImageProducer - класс MemoryImageSource - создает пикселы в оперативной памяти по массиву байтов или целых чисел. Вначале создается массив pix, содержащий цвет каждой точки. Затем одним из 6 конструкторов создается объект класса MemoryImageSource. Он может быть обработан потребителем или прямо преобразован в тип Image методом createImage().

В листинге приведена простая программа, выводящая на экран квадрат размером 100x100 пикселов. Левый верхний угол квадрата синий, левый нижний — красный, правый верхний — зеленый, а к центру квадрата цвета перемешиваются.

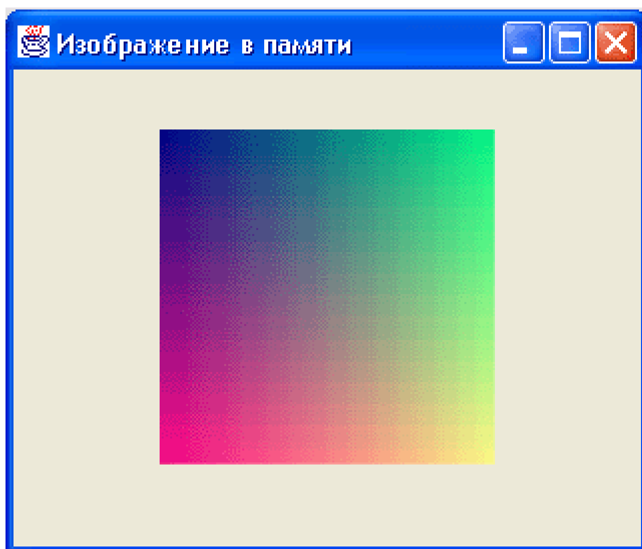
Листинг. Изображение, построенное по точкам

```
import java.awt.*;
import j ava.awt.event.*;
import java.awt.image.*;
```

```
Class InMemory extends Frame {
    private int w = 100, h = 100;
    private int[] pix = new int[w * h];
    private Image img;
    InMemory(String s)( super(s);
    int i = 0;
    for (int y = 0; y < h; y++) {
        int red = 255 * y / (h - 1);
        for (int x = 0; x < w; x++) {
            int green = 255 * x / (w — 1) ;
            pix[i++] = (255 << 24)|(red << 16)|(green << 8)| 128;
        }
    }
    setSize(250, 200);
    setVisible(true);
}
public vqid paint(Graphics gr) {
    if (img == null)
        img = createImage(new MemoryImageSource<w, h, pix> 0, w));
    gr.drawImage(img, 50, 50, this);
}
public static void main(String[] args) {
    Frame f = new InMemory(" Изображение в памяти");
    f.addWindowListener (
        new WindowAdapter()  {
            public void windowClosing(WindowEvent ev) {
                System.exit (0);
            }
        }
    );
}
```

В листинге в конструктор класса-поставщика MemoryImageSource (w, h, pix, о, w) заносится ширина w и высота h изображения, массив pix, смещение в этом массиве о и длина строки w. Потребителем служит изображение img, которое создается методом createImage() и выводится на экран методом drawImage(img, 50, 50, this). Левый верхний угол изображения img располагается в точке (50, 50) контейнера, а последний аргумент this показывает, что роль imageObserver играет сам класс InMemory. Это заставляет включить в метод paint() проверку if (img == null), иначе изображение будет постоянно перерисовываться.

Вывод этой программы.

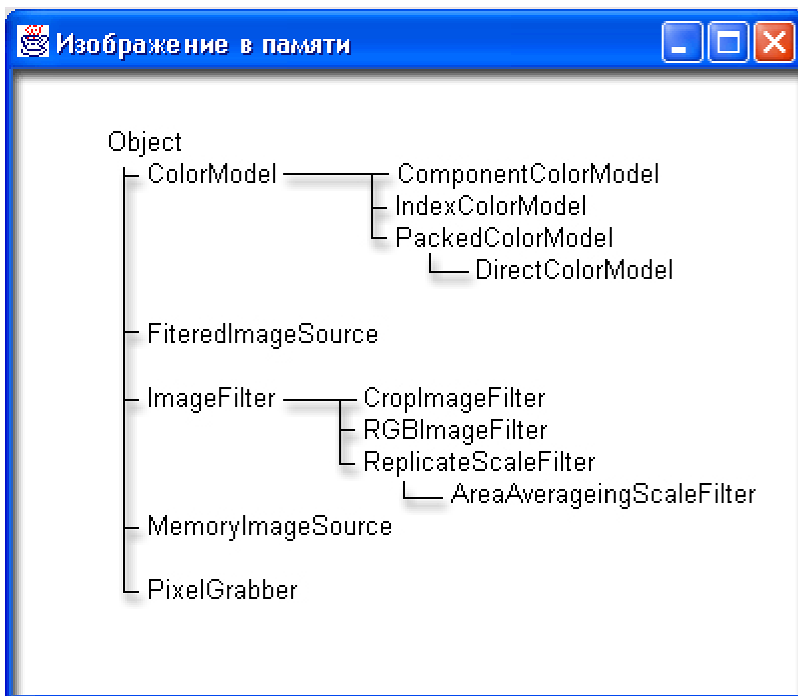


Интерфейс imageConsumer описывает 7методов, самыми важными из которых являются 2 метода setPixeis():

- setPixels(int x, int y, int width, int height, ColorModel model, byte[] pix, int offset, int scansize);
- Второй метод отличается только тем, что массив pix содержит элементы типа int.

К этим методам обращается поставщик для передачи пикселов потребителю. Передается прямоугольник шириной width и высотой height с заданным верхним левым углом (x, y), заполняемый пикселами из массива pix, начиная с индекса offset. Каждая строка занимает scansize элементов массива pix. Цвета пикселов определяются в цветовой модели model (обычно это модель RGB).

Классы модели "поставщик-потребитель"



Классы-фильтры.

Интерфейс imageConsumer нет нужды реализовывать, обычно используется его готовая реализация - класс imageFilter. Несмотря на название, этот класс не производит никакой фильтрации, он передает изображение без изменений. Для преобразования изображений данный класс следует расширить, переопределив метод setPixeils(). Результат преобразования следует передать потребителю, роль которого играет поле consumer этого класса.

В пакете java. awt. image есть 4 расширения класса ImageFilter:

- CropImageFilter (int x, int y, int w, int h) - выделяет фрагмент изображения, указанный в приведенном конструкторе;
- RGBimageFilter - позволяет изменять отдельные пикселы; это абстрактный класс, он требует расширения и переопределения своего метода filterRGBO ;
- ReplicateScaieFilter (int w, int h) - изменяет размеры изображения на указанные в приведенном конструкторе, дублируя строки и/или столбцы при увеличении размеров или убирая некоторые из них при уменьшении;

- AreaAveragingScaleFilter (int w, int h) - расширение предыдущего класса; использует более сложный алгоритм изменения размеров изображения, усредняющий значения соседних пикселов.

Применяются эти классы совместно со вторым классом-поставщиком, реализующим интерфейс ImageProducer - классом FilteredImageSource. Этот класс преобразует уже готовую продукцию, полученную от другого поставщика producer, используя для преобразования объект filter класса-фильтра imageFilter или его подкласса, Оба объекта задаются в конструкторе

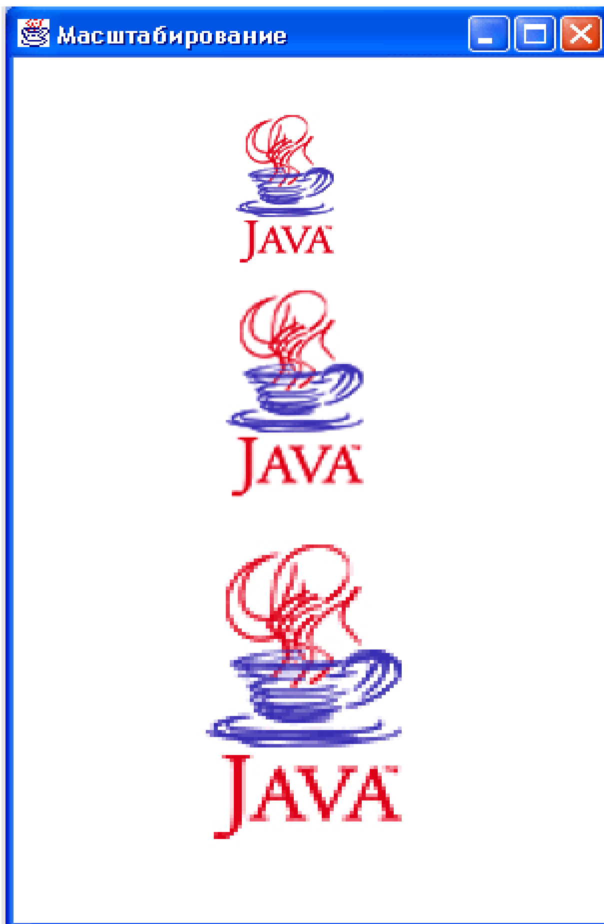
FilteredImageSource(ImageProducer producer, ImageFilter filter)

Листинг. Примеры масштабирования изображения

```
import j ava.awt.*;
import j ava.awt.event.*;
import j ava.awt.image.*;
Class CropTest extends Frame {
    private Image img, cropimg, replimg, averimg;
    CropTest(String s)  {
        super (s) ;
            // 1. Создаем изображение - объект класса Image
        img = getToolkit().getImage("javalogo52x88.gif");
            // 2. Создаем объекты-фильтры:
            // а) выделяем левый верхний угол размером 30x30
        CropImageFilter crp = new CropImageFilter(0, 0, 30, 30);
            // б) увеличиваем изображение в два раза простым методом
        RepiicateScaieFiiter rsf = new RepiicateScaieFiiter(104, 176);
            // в) увеличиваем изображение в два раза с усреднением
        AreaAveragingScaleFilter asf = new AreaAveragingScaleFilter(104, 176);
            // 3. Создаем измененные изображения
        cropimg = createImage(new FilteredImageSource(img.getSource(), crp));
        replimg = createImage(new FilteredImageSource(img.getSource(), rsf)};
        averimg = createImage(new FilteredImageSource(img.getSource(), asf));
        setSize(400, 350); setvisible(true);
    }
    public void paint(Graphics gS { g.drawImage(img, 10, 40, this);
    g.drawImage(cropimg, 150, 40, 100, 100, this);
    g.drawImage(replimg, 10, 150, this);
    g.drawImage(averimg, 150, 150, this);
}
public static void main(String[] args) {
    Frame f= new CropTest(" Масштабирование");
```

```
    f.addWindowListener (
        new WindowAdapter()  {
            public void windowClosing(WindowEvent ev) {
            System.exit(0);
            }
        }
    );
}
```

На рисунке сверху показано исходное изображение, ниже - увеличенный фрагмент, внизу - изображение, увеличенное двумя способами.



Как изменить цвет изображения.

В листинге ниже меняются цвета каждого пиксела изображения. Это достигается просто сдвигом rgb>>1 содержимого пиксела на один бит вправо в методе filterRGB(). При этом усиливается красная составляющая цвета. Метод fiterRGB() переопределен в расширении coiorFilter класса RGBImageFilter.

Листинг. Изменение цвета всех пикселов ;

```
import j ava.awt.*;
import java.awt.event.*;
import java.awt.image.*;
Class RGBTest extends Frame {
    private Image img, newimg;
    RGBTest(String s)  {
        super(s);
        img = getToolkit().getlmage("javalogo52x88.gif");
        RGBImageFilter rgb = new CoiorFilter();
        newimg = createlmage(new FilteredlmageSource(img.getSource(), rgb));
        setSize(400, 350);
        setVisible(true);
    }
    public void paint(Graphics g) {
        g.drawlmage(img, 10, 40, this);
        g.drawlmage(newimg, 150, 40, this);
    }
    public static void main(String[] args) {
        Frame f= new RGBTest(" Изменение цвета");
        f.addWindowListener (
            new WindowAdapter()  {
                public void wlndowClosing(WindowEvent ev) {
                    System.exit(0);
                }
            }
        );
    )
}
Class CoiorFilter extends RGBImageFilter {
    CoiorFilter() {
        canFilterlndexColorModel = true;
    }
    public int fiiterRGB(int x, int y, int rgb) {
        return rgb>>1;
```

```
    }
}
```

Как переставить пикселы изображения

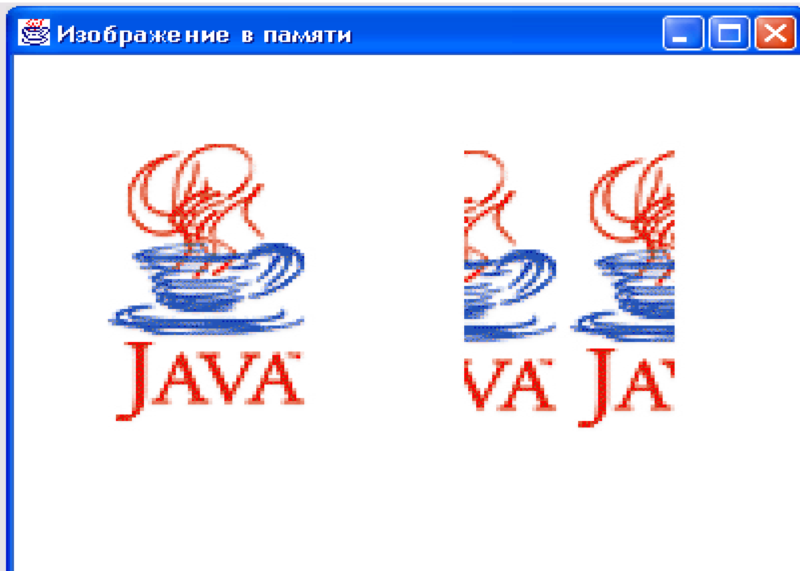
В листинге ниже определяется преобразование пикселов изображения. Создается новый фильтр - расширение shiftFiiter класса ImageFilter, сдвигающее изображение циклически вправо на указанное в конструкторе число пикселов. Все, что для этого нужно, - это переопределить метод setPixels().

Листинг. Циклический сдвиг изображения

```
import j ava.awt.*;
import j ava.awt.event.*;
import j ava.awt.image.*;
Class ShiftImage extends Frame {
    private Image img, newimg;
    ShiftImage(String s){ super(s);
        // 1. Получаем изображение из файла
    img = getToolkit().getImage("javalogo52x88.gif");
        // 2. Создаем экземпляр фильтра
    ImageFilter imf = new ShiftFiiter(26);
        // Сдвиг на 26 пикселов
        // 3. Получаем новые пикселы с помощью фильтра
    ImageProducer ip = new FilteredImageSource(img.getSource(), imf);
        // 4. Создаем новое изображение
    newimg = createImage(ip);
    setSize(300, 200);
    setVisible(true) ; }
    public void paint(Graphics gr) {
        gr.drawImage(img, 20, 40, this);
        gr.drawImage(newimg, 100, 40, this);
    }
    public static void main(StringU args) {
        Frame f= new ShiftImage(" Циклический сдвиг изображения");
        f.atidWindowListener (
            new WindowAdapter()  {
                public void windowClosing(WindowEvent ev) {
                    System.exit(0); )
                }
            }
        );
    }
```

```
// Класс-фильтр
Class ShiftFilter extends ImageFilter {
    private int sh;
        // Сдвиг на sh пикселов вправо.
    public ShiftFilter(int shift)!{ sh = shift;
}
public void setPixels(int x, int y, int w, int h,
    ColorModel m, byte[] pix, int off, int size) {
    for (int k = x; k < x+w; k++) {
        if (k+sh <= w)
            consumer.setPixels(k, y, 1, h, m, pix, off+sh+k, size);
        else
            consumer.setPixels(k, y, 1, h, m, pix, off+sh+k-w, size);
    }
}
}
```

Как видно из листинга, переопределение метода setPixels() заключается в том, чтобы изменить аргументы этого метода, переставив пикселы изображения, и передать их потребителю consumer - полю класса imageFiiter методом setPixels() потребителя. На рисунке показан результат выполнения этой программы.



18.3. Модель обработки прямым доступом

Вторая модель обработки изображения введена в Java 2D. Она названа моделью прямого доступа (immediate mode model).

Подобно тому, как вместо класса Graphics система Java 2D использует его расширение Graphics2D, вместо класса Image в Java 2D употребляется его расширение - класс BufferedImage. В конструкторе этого класса

BufferedImage(int width, int height, int imageType)

задаются размеры изображения и способ хранения точек - одна из констант:

TYPE_INT_RGB TYPE_4BYTE_ABRG TYPE_USHORT_565_RGB
TYPE_INT_ARGB TYPE_4BYTE_ABRG_PRE TYPE_USHORT_555_RGB
TYPE_INT_ARGB_PRE TYPE_BYTE_GRAY TYPE_USHORT_GRAY
TYPE_INT_BRG TYPE_BYTE_BINARY
TYPE_3BYTE_BRG TYPE_BYTE_INDEXED

Как видите, каждый пиксел может занимать 4 байта - INT, 4BYTE, или 2 байта - USHORT, или 1 байт -BYTE. Может использоваться цветовая модель RGB, или добавлена альфа-составляющая -ARGB, или задан другой порядок расположения цветовых составляющих - BRG, или заданы градации серого цвета - GRAY. Каждая составляющая цвета может занимать 1 байт, 5 или 6 битов.

Экземпляры класса BufferedImage редко создаются конструкторами. Для их создания чаще обращаются к методам createImage() класса Component с простым приведением типа:

BufferedImage bi = (BufferedImage) createimage(width, height)

При этом экземпляр bi получает характеристики компонента: цвет фона и цвет рисования, способ хранения точек.

Расположение точек в изображении регулируется классом Raster или его подклассом WritableRaster. Эти классы задают систему координат изображения, предоставляют доступ к отдельным пикселам методами getPixel(), позволяют выделять фрагменты изображения методами getPixels(). Класс WritableRaster дополнительно разрешает изменять отдельные пикселы методами setPixel() или целые фрагменты изображения методами setPixels() и setRect().

Начало системы координат изображения - левый верхний угол - имеет координаты (minX, minY), не обязательно равные нулю.

При создании экземпляра класса BufferedImage автоматически формируется связанный с ним экземпляр класса WritableRaster.

Точки изображения хранятся в скрытом буфере, содержащем одномерный или двумерный массив точек. Вся работа с буфером осуществляется методами одного из классов DataBufferByte, DataBufferInt, DataBufferShort, DataBufferUshort в зависимости от длины данных. Общие свойства этих классов собраны в их абстрактном суперклассе DataBuffer. В нем определены типы данных, хранящихся в буфере: TYPE_BYTE, TYPEJJSHORT, TYPE_INT, TYPEJJNDEFINED.

Методы класса DataBuffer предоставляют прямой доступ к данным буфера, но удобнее и безопаснее обращаться к ним методами классов Raster и WritableRaster.

При создании экземпляра класса Raster или класса WritableRaster создается экземпляр соответствующего подкласса класса DataBuffer.

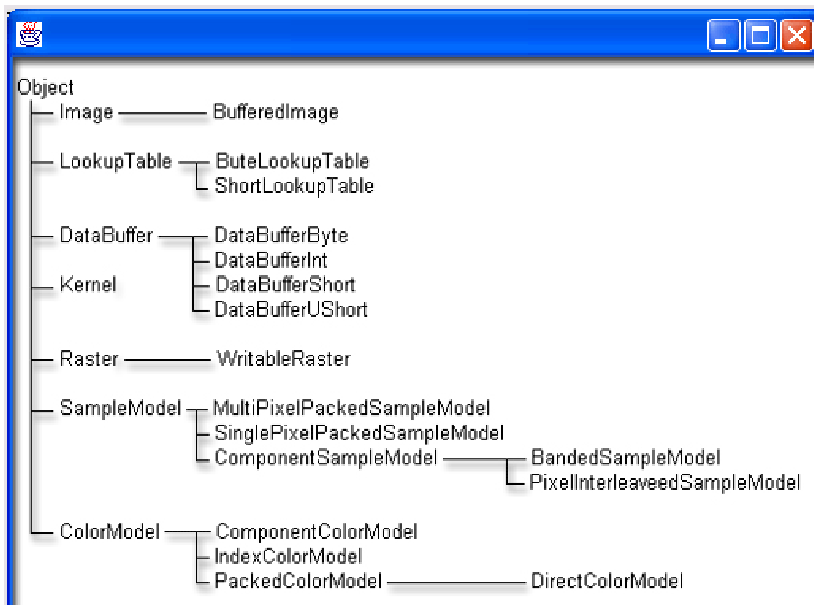
Чтобы отвлечься от способа хранения точек изображения, Raster может обращаться не к буферу DataBuffer, а к подклассам абстрактного класса SampleModel, рассматривающим не отдельные байты буфера, а составляющие (samples) цвета. В модели RGB - это красная, зеленая и синяя составляющие. В пакете java.awt.image есть 5 подклассов класса SampleModel:

- componentSampleModel - каждая составляющая цвета хранится в отдельном элементе массива DataBuffer;
- BandedSampleModel - данные хранятся по составляющим, составляющие одного цвета хранятся обычно в одном массиве, а DataBuffer содержит двумерный массив: по массиву для каждой составляющей; данный класс расширяет класс ComponentSampleModel ;
- PixelInterleavedSampieModel - все составляющие цвета одного пиксела хранятся в соседних элементах единственного массива DataBuffer ; данный класс расширяет класс ComponentSampleModel ;
- MultiPixeiPackedSampieModel - цвет каждого пиксела содержит только одну составляющую, которая может быть упакована в один элемент массива DataBuffer ;
- singlePixelPackedSampleModel - все составляющие цвета каждого пиксела хранятся в одном элементе массива DataBuffer.

На рисунке представлена иерархия классов Java 2D, реализующая модель прямого доступа.

Итак, Java 2D создает сложную и разветвленную трехслойную систему DataBuffer - SampieModei - Raster управления данными изображения BufferedImage. Вы можете манипулировать точками изображения, используя их координаты в методах классов Raster или спуститься на уровень ниже и обращаться к составляющим цвета пиксела методами классов SampleModei. Если

же вам надо работать с отдельными байтами, воспользуйтесь классами DataBuffer.



Применять эту систему приходится редко, только при создании своего способа преобразования изображения. Стандартные же преобразования выполняются очень просто.

18.4. Преобразование изображения в Java 2D

Преобразование изображения source, хранящегося в объекте класса BufferedImage, в новое изображение destination выполняется методом filter(BuffredImage source, BuffredImage destination) описанным в интерфейсе BuffredimageOp. Указанный метод возвращает ссылку на новый, измененный объект destination класса BuffredImage, что позволяет задать цепочку последовательных преобразований.

Можно преобразовать только координатную систему изображения методом filter(Raster source, WritableRaster destination) возвращающим ссылку на измененный объект класса WritableRaster. Данный метод описан в интерфейсе RasterOp.

Способ преобразования определяется классом, реализующим эти интерфейсы, а параметры преобразования задаются в конструкторе класса.

В пакете java.awt.image есть 6 классов, реализующих интерфейсы BuffredimageOp и RasterOp:

- AffineTransformOp - выполняет аффинное преобразование изображения: сдвиг, поворот, отражение, сжатие или растяжение по осям;
- RescaieOp - изменяет интенсивность изображения;
- LookupOp - изменяет отдельные составляющие цвета изображения;
- BandCombineOp - меняет составляющие цвета в Raster;
- ColorConvertdp - изменяет цветовую модель изображения;
- ConvolveOp - выполняет свертку, позволяющую изменить контраст и/или яркость изображения, создать эффект "размытости" и другие эффекты.

Рассмотрим, как можно применить эти классы для преобразования изображения.

18.4.1. Аффинное преобразование изображения

Класс AffineTransform и его использование подробно разобраны выше, здесь мы только применим его для преобразования изображения.

В конструкторе класса AffineTransformOp указывается предварительно созданное аффинное преобразование at и способ интерполяции interp и/или правила визуализации hints:

```
AffineTransformOp(AffineTransform at, int interp);
AffineTransformOp(AffineTransform at, RenderingHints hints);
```

Способ интерполяции — это одна из двух констант:

- TYPE_NEAREST_NEIGHBOR (по умолчанию во втором конструкторе)
- или TYPE_BILINEAR.

После создания объекта класса AffineTransformOp применяется метод filter(). При этом изображение преобразуется внутри новой области типа BufferedImage, как показано на риунке справа. Сама область выделена черным цветом.

Другой способ аффинного преобразования изображения - применить метод drawImage(BufferedImage img, BufferedImageOp op, int x, int y) класса Graphics2D. При этом преобразуется вся область img, как продемонстрировано на рисунке посередине.

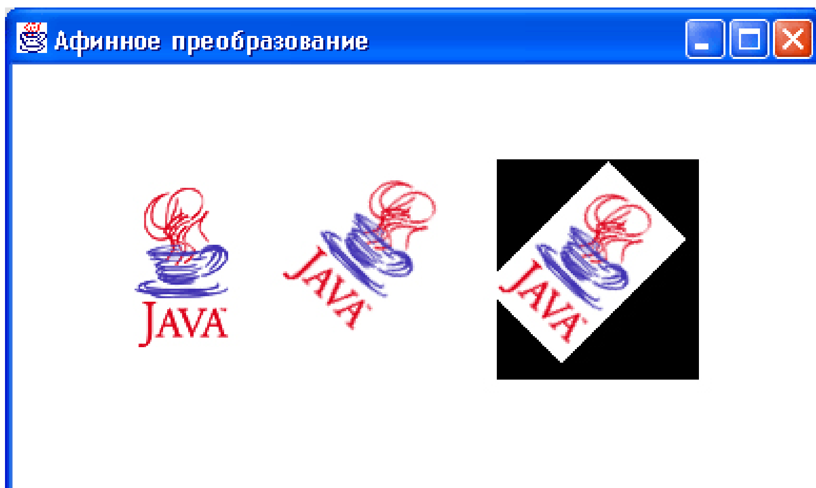
<u>Листинг. Аффинное преобразование изображения</u>

```
import j ava.awt.*;
import Java.awt.geom.*;
import Java.awt. image.*;
```

```
import java.awt.event.*;
public Class AffOp extends Frame {
    private BufferedImage bi;
    public AffOp(String s) {
        super (s) ;
            // Загружаем изображение
        img Image img = getToolkit().getImage("javalogo52x88.gif");
            // В этом блоке организовано ожидание загрузки
        try  {
            MediaTracker mt = new MediaTracker(this);
            mt.addImage(img, 0);
            mt.waitForID(O);                // Ждем окончания загрузки
        }
        catch(Exception e){ }
            // Размеры создаваемой области bi совпадают
            //с размерами изображения img
        bi = new BufferedImage(img.getWidth(this), img.getHeight(this),
        BufferedImage.TYPE_INT_RGB);
            // Создаем графический контекст big изображения bi
        Graphics2D big = bi.createGraphics();
            // Выводим изображение img в графический контекст
        big big.drawImage(img, 0, 0, this);
    }
    public void paint(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        int w = getSize().width;
        int h = getSize().height;
        int bw = bi.getWidth(this);
        int bh = bi.getHeight(this);
            // Создаем аффинное преобразование
         at AffineTransform at = new AffineTransform();
        at.rotate(Math.PI/4);     // Поворот на 45 градусов по часовой стрелке
            //Затем сдвигаем изображение вправо на величину bw
        at.preConcatenate(new AffineTransform(l, 0, O, 1, bw, 0));
        // Определяем область хранения bimg преобразованного изображения.
        Bufferedimage bimg =
            new BufferedImage(2*bw, 2*bw, BufferedImage.TYPE_INT_RGB);
            // Создаем объект biop,. содержащий преобразование at
        BufferedImageOp biop = new AffineTransformOp(at,
        AffineTransformOp.TYPE_NEAREST_NEIGHBOR);
            // Преобразуем изображение, результат заносим);
            // Выводим исходное изображение
```

```
            // Выводим измененную преобразованием область);
            // Выводим преобразованное внутри области bimg изображение
        g2.drawImage(bimg, null, w/2+3, 30);
    }
    public static void main(String[] args) {
        Frame f = new AffOpf" Аффинное преобразование");
        f.addWindowListener (
            new WindowAdapter()  {
                public void windowClosing(WindowEvent e) {
                    System.exit(0);
                }
            }
        );
        f.setSize(400, 200);
        f.setVisible(true) ;
    }
}
```

На рисунке показаны исходное изображение, преобразованная область и преобразованное внутри области изображение.



18.4.2. Изменение интенсивности изображения

Изменение интенсивности изображения выражается математически в умножении каждой составляющей цвета на число factor и прибавлении к результату умножения числа offset. Результат приводится к диапазону значений состав-

ляющей. После этого интенсивность каждой составляющей цвета линейно изменяется в одном и том же масштабе.

Числа factor и offset постоянны для каждого пиксела и задаются в конструкторе класса вместе с правилами визуализации hints:

RescaleOp(float factor, float^offset, RenderingHints hints)

После этого остается применить метод filter ().

Листинг. Изменение интенсивности изображения

```
import Java.awt.*;
import j ava.awt.image.*;
import j ava.awt.event.*;
public Class Rescale extends Frame {
    private BufferedImage bi;
    public Rescale(String s) {
        super (s) ;
        Image img = getToolkit().getImage("javalogo52x88.gif");
        try  {
            MediaTracker mt = new MediaTracker(this);
            mt.addImage(img, 0);
            mt.waitForlD(0);
        }
        catch(Exception e){ }
        bi = new BufferedImage(img.getWidth(this), img.getHeight(this),
        BufferedImage.TYPE_INT_RGB);
        Graphics2D big = bi.createGraphics();
        big.drawImage(img, 0, 0, this);
    }
    public void paint(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        int w = getSize().width;
        int bw = bi.getWidth(this);
        int bh = bi.getHeight(this);
        BufferedImage bimg =
        new BufferedImage(bw, bh, BufferedImage.TYPE_INT_RGB);
            //Начало определения преобразования
        RescaleOp rop = new RescaleOp(0.5f, 70.Of, null);
        rop.filter(bi, bimg);
            //Конец определения преобразования
        g2.drawImage(bi, null, 10, 30);
        g2.drawImage(bimg, null, w/2+3, 30);
```

```
    }
    public static void main(String[] args) {
        Frame f = new Rescale(" Изменение интенсивности");
        f.addWindowListener (
            new WindowAdapter()  {
                public void windowClosing(WindowEvent e) {
                    System.exit(0);
                }
            )
        );
        f.setSize(300, 200);
        f.setvisible(true);
    }
}
```

18.4.3. Изменение составляющих цвета

Чтобы изменить отдельные составляющие цвета, надо прежде всего посмотреть тип хранения элементов в BufferedImage, по умолчанию это TYPE_INT_RGB. Здесь 3 составляющие - красная, зеленая и синяя. Каждая составляющая цвета занимает один байт, все они хранятся в одном числе типа int. Затем надо составить таблицу новых значений составляющих. В листинге это двумерный массив samples. Потом заполняем данный массив нужными значениями составляющих каждого цвета. В листинге задается ярко-красный цвет рисования и белый цвет фона. По полученной таблице создаем экзем-

пляр класса ByteLookupTable, который свяжет эту таблицу с буфером данных. тот экземпляр используем для создания объекта класса LookupOp. Наконец, применяем метод filter() этого класса.

В листинге приведен только фрагмент программы.

Листинг. Изменение составляющих цвета

```
        //Вста вить в листинг полный
byte samples[][] = new byte[3][256];
for (int j = 0; j < 255; j++) {
    samples[0][j] = (byte)(255);    // Красная составляющая
    samples[1][j] = (byte)(0);      // Зеленая составляющая
    samples[2][j] = (byte)(0);      // Синяя составляющая
}
samples[0][255] = (byte) (255);  // Цвет фона — белый
samples[1][255] = (byte) (255) ;
samples [2] [255] = (bybej (255) ;
ByteLookupTabie blut=new ByteLookupTabie(0, samples);
LookupOp lop = new LookupOp(blut, null);
lop.filter(bi, bimg);
            // Конец вст авки
```

18.4.4. Создание различных эффектов

Операция свертки (convolution) задает значение цвета точки в зависимости от цветов окружающих точек следующим образом.

Пусть точка с координатами (x, y) имеет цвет, выражаемый числом A(x, y). Составляем массив из 9 вещественных чисел w(0), w(i),... w(8). Тогда новое значение цвета точки с координатами (x, y) будет равно:

w(0)*A(x-l, y-l)+w(l)*A(x, y-l)+w(2)*A(x+l, y-l)+
w(3)*A(x-l, y)+w(4)*A(x, y)+w(5)*A(x+l, y)+
w(6)*A(x-l, y+l)+w(7)*A(x, y+l)+w(8)*A(x+l, y+1)

Задавая различные значения весовым коэффициентам w(i), будем получать различные эффекты, усиливая или уменьшая влияние соседних точек.

Если сумма всех 9 чисел w(i) равна 1.0f, то интенсивность цвета останется прежней. Если при этом все веса равны между собой, т.е. равны 0.1111111f, то получим эффект размытости, тумана, дымки. Если вес w(4) значительно больше остальных при общей сумме их l.0f, то возрастет контрастность, возникнет эффект графики, штрихового рисунка.

Можно свернуть не только соседние точки, но и следующие ряды точек, взяв массив весовых коэффициентов из 15 элементов (3x5, 5x3), 25 элементов (5x5) и больше.

В Java 2D свертка делается так. Сначала определяем массив весов, например:

```
float[] w = (0, -1, O, -1, 5, -1, O, -1, 0};
```

Затем создаем экземпляр класса Kernel - ядра свертки:

```
Kernel kern = new Kernel(3, 3, w);
```

Потом объект класса ConvoiveOp с этим ядром:

```
ConvolveOp conv = new ConvoiveOp(kern);
```

Все готово, применяем метод filter(): conv.filter(bi, bimg);

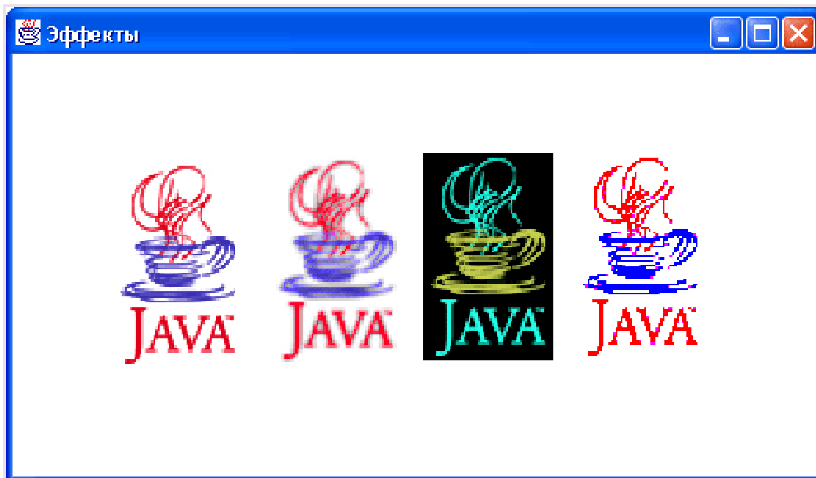
В листинге записаны действия, необходимые для создания эффекта "размытости".

<u>Листинг. Создание различных эффектов</u>

```
            // Вставить в листинг полный
float[] wl = { 0.11111111f, 0. 11111111f, 0. 11111111f,
          0. 11111111f, 0. 11111111f, 0. 11111111f,
          0. 11111111f, 0. 11111111f, 0. 11111111f };
Kernel kern = new Kernel(3, 3, wl);
ConvolveOp cop = new ConvolveOp(kern, ConvolveOp.EDGE_NO_OP, null);
copl.fliter(bi, bimg) ;
            // Конец вставки
```

На рисунке представлены слева направо исходное изображение и изображения, преобразованные весовыми матрицами wl, w2 и w3, где матрица w1 показана в листинге, а матрицы w2 и w3 выглядят так:

```
float[] w2 = { 0, -1, 0,-1, 4, -1, 0, -1, 0 } ;
float[] w3 = { -1, -1, -1,-1, 9, -1, -1, -1, -1 };
```



18.5. Анимация

18.5.1. Способы

Есть несколько способов создать анимацию. Самый простой из них — записать заранее все необходимые кадры в графические файлы, загрузить их в оперативную память в виде Объектов класса Image или BufferedImage и выводить по очереди на экран.

Это сделано в листинге. Заготовлено 10 кадров в файлах run1.gif, run2.gif, … run10.gif. Они загружаются в массив img[] и выводятся на экран циклически 100 раз, с задержкой в 0,1 сек.

<u>Листинг. Простая анимация</u>

```
import java.awt.*;
import ]ava.awt.event.*;
Class SimpleAnim extends Frame {
    private Image[] img = new Image[10];
    private int count;
    SimpleAnim(String s){ super(s);
    MediaTracker tr = new MediaTracker(this);
    for (int k = 0; k < 10; k++) {
        img[k] = getToolkit(}.getImage("run"+(k+D+".gif");
        tr.addImage(img[k], 0);
    }
    try  {
```

```
            tr.waitForAll(); // Ждем загрузки всех изображений
        }
        catch(InterruptedException e){}
        setSize(400, 300);
        setvisible(true);
    },
    public void paint(Graphics g) {
        g.drawImage(img[count % 10], 0, 0, this);
    }
    public void go() {
        while(count < 100) {
            repaint(); // Выводим следующий кадр
            try {       // Задержка в 0.1 сек
                Thread.sleep(100);
            }
            catch(InterruptedException e){}
            count++;
        }
    }
    public static void main(String[] args) {
        SimpleAnim f = new SimpleAnim(" Простая анимация");
        f.go();
        f.addWindowListener (
            new WindowAdapter() {
                public void windowClosing(WindowEvent ev) {
                    System.exit(0);
                }
            }
        );
    }
```

Обратите внимание на следующее важное обстоятельство. Мы не можем обратиться прямо к методу paint() для перерисовки окна компонента, потому что выполнение этого метода связано с операционной системой — метод paint() выполняется автоматически при каждом изменении содержимого окна, его перемещении и изменении размеров. Для запроса на перерисовку окна в классе Component есть метод repaint().

Метод repaint() ждет, когда представится возможность перерисовать окно, и потом обращается к методу update(Graphics g). При этом несколько обращений к repaint() могут быть произведены исполняющей системой Java за один раз.

Метод update() сначала обращается к методу g.ciearRect(), заполняющему окно цветом фона, а уж затем к методу paint (g). Полный исходный текст таков:

```
public void update(Graphics g) {
    if ((this instanceof java.awt.Canvas) || (this instanceof java.awt.Panel) ||
        (this instanceof java.awt.Frame)  || (this instanceof java.awt.Dialog)  ||
        (this instanceof java.awt.Window)) {
        g.clearRect(0, 0, width, height);
    }
    paint(g);
}
```

Если кадры анимации полностью перерисовывают окно, то его очистка методом clearRect() не нужна. Более того, она часто вызывает неприятное мерцание из-за появления на мгновение белого фона. В таком случае надо сделать следующее переопределение:

```
public void update(Graphics g) {
    paint(g);
}
```

Для "легких" компонентов дело обстоит сложнее. Метод repaint() последовательно обращается к методам repaint() объемлющих "легких" контейнеров, пока не встретится "тяжелый" контейнер, чаще всего это экземпляр класса Container. В нем вызывается метод update(), очищающий и перерисовывающий контейнер. После этого идет обращение к методам update() всех "легких" компонентов в контейнере.

Отсюда следует, что для устранения мерцания "легких" компонентов необходимо переопределять метод update() первого объемлющего "тяжелого" контейнера, обращаясь в нем к методам super.update(g) или super.paint(g).

Если кадры покрывают только часть окна, причем каждый раз новую, то очистка окна необходима, иначе старые кадры останутся в окне, появится "хвост". Чтобы устранить мерцание, используют прием, получивший название "двойная буферизация " (double buffering).

18.5.2. Двойная буферизация

Суть двойной буферизации в том, что в оперативной памяти создается буфер - объект класса Image или BufferedImage, и вызывается его графический контекст, в котором формируется изображение. Там же происходит очистка буфера, которая тоже не отражается на экране. Только после выполнения всех действий готовое изображение выводится на экран.

Все это происходит в методе update(), а метод paint() только обращается к update(). Листинги разъясняют данный прием.

Листинг. Двойная буферизация с помощью класса Image

```
public void update(Graphics g) {
    int w = getSize().width, h = getSize().height;
        // Создаем изображение-буфер в оперативной памяти
    Image offlmg = createlmage(w, h);
        // Получаем его графический контекст
    Graphics offGr = offImg.getGraphics();
        // Меняем текущий цвет буфера на цвет фона
    offGr.setColor(getBackground());
        //и заполняем им окно компонента, очищая буфер
    offGr.fillRect(0, 0, w, h);
        // Восстанавливаем текущий цвет буфера
    offGr.setColor(getForeground());
        // выводим в контекст изображение
    offGr.drawlmage(img[count % 10], 0, 0, this);
        // Рисуем в графическом контексте буфера
        // (необязательное действие)
    paint(offGr);
        // Выводим изображение-буфер на экран
        // (можно перенести в метод paint())
    g.drawlmage(offlmg, 0, 0, this);
        // Метод paint() необязателен
    public void paint(Graphics g)J update(g);
}
```

Листинг. Двойная буферизация с помощью класса BufferedImage

```
public void update(Graphics g) {
    Graphics2D g2 = (Graphics2D},g;
    int w = getSize().width, h = getSize().height;
        // Создаем изображение-буфер в оперативной памяти
    Bufferedimage bi = (Bufferedimage)createlmage(w, h);
        // Создаем графический контекст буфера
    Graphics2D big = bi.createGraphics();
        // Устанавливаем цвет фона
    big.setColor(getBackground());
        // Очищаем буфер цветом фона
    big.clearRect(0, 0, w, h);
        // Восстанавливаем текущий цвет
```

```
    big.setColor(getForeground());
        // Выводим что-нибудь в графический контекст big
        //...
        // Выводим буфер на экран
    g2.drawImage(bi, 0, 0, this);
}
```

Метод двойной буферизации стал фактическим стандартом вывода изменяющихся изображений, а в библиотеке Swing он применяется автоматически.

Данный метод удобен и при перерисовке отдельных частей изображения. В этом случае в изображении-буфере рисуется неизменяемая часть изображения, а в методе paint() - то, что меняется при каждой перерисовке.

В листинге показан второй способ анимации - кадры изображения рисуются непосредственно в программе, в методе update(), по заданному закону изменения изображения. В результате красный мячик прыгает на фоне изображения.

Листинг. Анимация рисованием

```
import Java.awt.*;
import j ava.awt.event.*;
import Java.awt.geom.*;
import java.awt.image.*;
Class DrawAniml extends Frame {
    private Image img;
    private int count;
    DrawAniml(String s) {
        super(s);
        MediaTracker tr = new MediaTracker(this);
        img = getToolkit().getImage("back2.jpg");
        tr.addImage(img, 0);
        try {
            tr.waitForID(0) ;
        }
        catch(InterruptedException e) { }
        SetSize(400, 400);
        setvisible(true);
    }
    public void update(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        int w = getSizeO.width, h = getSize().height;
        BufferedImage bi = (BufferedImage)createImage(w, h) ;
        Graphics2D big = bi.createGraphics();
```

```
            // Заполняем фон изображением img
        big.drawImage(img, 0, 0, this);
            // Устанавливаем цвет рисования
        big.setColor(Color.red);
            // Рисуем в графическом контексте буфера круг,
            // перемещающийся по синусоиде
        big.fill(new Arc2D.Double(4*count, 50+30*Math.sin(count),
            50, 50, 0, 360, Arc2D.OPEN));
            // Меняем цвет рисования
        big.setColor(getForeground());
            // Рисуем горизонтальную прямую
        big.draw(new Line2D.Double(0, 125, w, 125));
            // Выводим изображение-буфер на экран
        g2.drawImage(bi, 0, 0, this);
    }
    public void go() {
        while(count < 100) {
            repaint();
            try {
                Thread.sleep(10);
            }
            catch(InterruptedException e) { }
            count++;
        }
    }
    public static void main(String[] args) {
        DrawAniml f = new DrawAniml(" Анимация");
        f.go();
        f.addWindowListener (
            new WindowAdapter() {
                public void windowClosing(WindowEvent ev) {
                    System.exit(0);
                }
            }
        );
    }
}
```

Эффект мерцания, переливы цвета, затемнение и прочие эффекты, получающиеся заменой отдельных пикселов изображения, удобно создавать с помощью класса MemoryimageSource. Методы newPixels() этого класса вызывают немедленную перерисовку изображения даже без обращения к методу

repaint(), если перед этим выполнен метод setAnimated(true). Чаще всего применяются два метода:

- newPixels(int x, int y, int width, int height) - получателю посылается указанный аргументами прямоугольный фрагмент изображения;
- nevPixels() - получателю посылается все изображение.

В листинге показано применение этого способа. Квадрат, выведенный на экран, переливается разными цветами.

Листинг. Анимация с помощью MemoryImageSource

```
import Java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
Class InMemory extends Frame {
    private int w = 100, h = 100, count;
    private int[] pix = new int[w * h];
    private Image img;
    MemoryImageSource mis;
    InMemory(String s){ super(s);
    int i = 0;
    for(int y = 0; y < h; y++) {
        int red = 255 * y / (h - 1);
        for(int x = 0; x < w; x++)          {
            int green = 25$ * x / (w - 1);
            pix[i++] = (255 « 24}|(red << 16)|(green << 8) | 128;
        }
    }
    mis = new MemoryImageSource(w, h, pix, 0, w);
        // Задаем возможность анимации
    mis.setAnimated(true);
    img = createImage(mis);
    setSize(350, 300);
    setVisible(true);
}
public void paint(Graphics gr) {
    gr.drawImage(img, 10, 30, this);
}
public void update(Graphics g) {
    paint(g);
}
public void got) {
```

```
    while (count < 100) {
        int i = 0;
                // Изменяем массив пикселов по некоторому закону
        for(int y - 0; y < h;,y++)
            for (int x. = 0; x < w; x++)
                pix[i++J = (255 « 24)|(255 + 8 * count « 16)|
                    (8*count «8)| 255 + 8 * count;
                // Уведомляем потребителя об изменении
        mis.newPixels();
        try {
            Thread.sleep(100);
        }
        catch(InterruptedException e) { }
        count++;
    }
}
public static void main(String[] args) {
    InMemory f = new InMemory(" Изображение в памяти");
    f.go();
    f.addWindowListener (
        new WindowAdapter() {
            public void windowClosing(WindowEvent ev) {
                System.exit(0);
            }
        )
    );
}
```

Вот и все средства для анимации, остальное - умелое их применение. Комбинируя рассмотренные способы, можно добиться удивительных эффектов. В документации SUN J2SDK, в каталогах demo\applets и demo\jfc\Java2D \src, приведено много примеров апплетов и приложений с анимацией.

18.6. Звук

18.6.1. Введение

Как было указано в предыдущей главе, в апплетах реализуется интерфейс AudioClip. Экземпляр объекта, реализующего этот интерфейс можно получить методом getAudioClip(), который, кроме того, загружает звуковой файл, а затем пользоваться методами play(), loop() и stop() этого интерфейса для проигрывания музыки.

Для применения данного же приема в приложениях в класс Applet введен статический метод newAudioClip(URL address), загружающий звуковой файл, находящийся по адресу address, и возвращающий объект, реализующий интерфейс AudioClip. Его можно использовать для проигрывания звука в приложении, если конечно звуковая система компьютера уже настроена.

В листинге приведено простейшее консольное приложение, бесконечно проигрывающее звуковой файл doom.mid, находящийся в текущем каталоге. Для завершения приложения требуется применить средства операционной системы, например, комбинацию клавиш <Ctrl>+<C>.

Листинг. Простейшее аудиоприложение

```
import j ava.applet.* ;
import j ava.net.*;
Class SimpleAudio {
    SimpleAudio () {
        try  {
            AudioClip ac = Applet.newAudioClip(new URL("file:doom.mid"));
            ac.loop();
        }
        catch(Exception e) { }
    }
    public static void main(String[] args) {
        new SimpleAudio();
    }
}
```

Таким способом можно проигрывать звуковые файлы типов AU, WAVE, AIFF, MIDI без сжатия.

В состав виртуальной машины Java, входящей в SUN J2SDK начиная с версии 1.3, включено устройство, проигрывающее звук, записанный в одном из форматов AU, WAVE, AIFF, MIDI, преобразующее, микширующее и записывающее звук в тех же форматах.

Для работы с этим устройством созданы классы, собранные в пакеты javax.sound.sampled, javax.sound.midi, javax.sound.sampled.spi и javax.sound.midi.spi. Перечисленный набор классов для работы со звуком получил название Java Sound API.

18.6.2. Проигрывание звука в Java 2

Проигрыватель звука, встроенный в JVM, рассчитан на 2 способа записи звука: моно и стерео оцифровку (digital audio) с частотой дискретизации (sample rate)

от 8000 до 48000 Гц и аппроксимацией (quantization) 8 и 16 битов, и MIDI-последовательности (sequences) типа 0 и 1.

Оцифрованный звук должен храниться в файлах типа AU, WAVE и AIFF. Его можно проигрывать двумя способами.

Первый способ описан в интерфейсе Clip. Он рассчитан на воспроизведение небольших файлов или неоднократное проигрывание файла и заключается в том, что весь файл целиком загружается в оперативную память, а затем проигрывается.

Второй способ описан в интерфейсе SourceDataLine. Согласно этому способу файл загружается в оперативную память по частям в буфер, размер которого можно задать произвольно.

Перед загрузкой файла надо задать формат записи звука в объекте класса AudioFormat. Конструктор этого класса:

```
AudioFormat(float sampleRate, int sampleSize, int channels,
    Boolean signed, Boolean bigEndian)
```

требует знания частоты дискретизации sampleRate (по умолчанию 44100 Гц), аппроксимации sampleSize, заданной в битах (по умолчанию 16), числа каналов channels (1 - моно, по умолчанию 2 - стерео), запись чисел со знаком при, signed=true, или без знака, и порядка расположения байтов в числе bigEndian. Такие сведения обычно неизвестны, поэтому их получают косвенным образом из файла. Это осуществляется в два шага.

На первом шаге получаем формат файла статическим методом getAudioFiieFormat() класса AudioSystem, на втором - формат записи звука методом getFormat() класса AudioFiieFormat. Это описано в листинге. После того как формат записи определен и занесен в объект класса AudioFormat, в объекте класса DataLine.info собирается информация о входной линии (line) и способе проигрывания clip или SourceDataLine. Далее следует проверить, сможет ли проигрыватель обслуживать линию с таким форматом. Затем надо связать линию с проигрывателем статическим методом getLine() класса AudioSystem. Потом создаем поток данных из файла - объект класса AudioInputStream. Из этого потока тоже можно извлечь объект класса AudioFormat методом getFormat(). Данный вариант выбран в листинге. Открываем созданный поток методом ореп().

Все готово, теперь можно начать проигрывание методом start(), завершить методом stop(), "перемотать" в начало методом setFramePosition(0) или setMillisecondPosition(0).

Можно задать проигрывание n раз подряд методом loop(n) или бесконечное число раз методом loop(Clip.LOOP_CONTINUOUSLY). Перед этим необходимо установить начальную (n) и конечную (m) позиции повторения методом setLoopPoints(n, m).

По окончании проигрывания следует закрыть линию методом close().

Листинг. Проигрывание аудиоклипа

```
import javax.sound.sampled.*;
import java.io.*;
Class PlayAudio {
    PlayAudio(String s) {
        play(s);
    }
    public void play(String file) {
        Clip line = null;
        try {
                // Создаем объект, представляющий файл
            File f = new File (file);
                // Получаем информацию о способе записи файла
            AudioFileFormat aff = AudioSystem.getAudioFileFormat(f);
                // Получаем информацию о способе записи звука
            AudioFormat af = aff.getFormat();
                // Собираем всю информацию вместе,
                // добавляя сведения о классе
            Class DataLine.Info info = new DataLine.Info(Clip.Class, af) ;
                // Проверяем, можно ли проигрывать такой формат
            if (!AudioSystem.isLineSupported(info)) {
                System.err.printlnt"Line is not supported");
                System.exit(0);
            }
                // Получаем линию связи с файлом
            line = (Clip)AudioSystem.getLine(info);
                // Создаем поток байтов из файла
            AudioInputStream ais - AudioSystem.getAudioInputStream(f);
                // Открываем линию
            line.open(ais);
        }
        catch(Exception e) {
            System.err.println(e);
        }
                // Начинаем проигрывание
```

```
        line.start();
                // Здесь надо сделать задержку до окончания проигрывания
                // или остановить его следующим методом:
        line.stop();
                //По окончании проигрывания закрываем линию
        line.close();
    }
    public static void main(String[] args) {
        if (args.length != 1)
            System.out.printlnt"Usage: Java PlayAudio filename");
        new PlayAudio(args[0]);
    }
}
```

Как видите, методы Java Sound API выполняют элементарные действия, которые надо повторять из программы в программу. Как говорят, это методы "низкого уровня" (low level).

Второй способ, использующий методы интерфейса SourceDataLine, требует предварительного создания буфера произвольного размера.

Листинг. Проигрывание аудиофайла

```
import javax.sound.sampled.*;
import j ava.io.*;
Class PlayAudioLine {
    PlayAudioLine(String s) {
        play(s);
    }
    public void play(String file) {
        SourceDataLine line = null;
        AudioInputStream ais = null;
        byte[] b = new byte[2048]; // Буфер данных
        try {
            File f = new File(file);
                // Создаем входной поток байтов из файла f
            ais = AudioSystem.getAudioInputStream(f);
                // Извлекаем из потока информацию о способе записи звука
            AudioFormat af = ais.getFormat () ;
                // Заносим эту информацию в объект info
            DataLine.Info info = new DataLine.Info(SourceDataLine.Class, af);
                // Проверяем, приемлем ли такой способ записи звука
            if (!AudioSystem.isLineSupported(info))
```

```
            {
                System.err.println("Line is not supported");
                System.exit(0);
            }
                // Получаем входную линию
            line = (SourceDataLine)AudioSystem.getLine(info);
                // Открываем линию
            line.open(af);
                // Начинаем проигрывание
            line.start(); // Ждем появления данных в буфере int num = 0;
                // Раз за разом заполняем буфер
            while(( num = ais.read(b)) != -1) line.write(b, 0, num);
                    // "Сливаем" буфер, проигрывая остаток файла
            line.drain();
                    // Закрываем поток
            ais.close();
        }
        catch (Exception e){
            System.err.println (e);
        }
            // Останавливаем проигрывание
        line.stop();
            // Закрываем линию
        line.close();
    }
    public static void main(String[] args) {
        String s = "mrmba.aif";
        if (args.length > 0) s = args[0];
        new PlayAudioLine(s) ;
    }
}
```

18.6.3. События

Управлять проигрыванием файла можно с помощью событий. Событие класса LineEvent происходит при открытии (OPEN) и закрытии (CLOSE) потока, при начале (START) и окончании (STOP) проигрывания. Характер события отмечается указанными константами. Соответствующий интерфейс LineListener описывает только один метод update().

В MIDI-файлах хранится последовательность (sequence) команд для секвенсора (sequencer) - устройства для записи, проигрывания и редактирования MIDI-последовательности, которым может быть физическое устройство или

программа. Последовательность состоит из нескольких дорожек (tracks), на которых записаны MIDI-события (events). Каждая дорожка загружается в своем канале (channel). Обычно дорожка содержит звучание одного музыкального инструмента или запись голоса одного исполнителя или запись нескольких исполнителей, микшированную синтезатором (synthesizer).

Для проигрывания MIDI-последовательности в простейшем случае надо создать экземпляр секвенсора, открыть его и направить в него последовательность, извлеченную из файла, как показано в листинге. После этого следует начать проигрывание методом start(). Закончить проигрывание можно методом stop(), "перемотать" последовательность на начало записи или на указанное время проигрывания - методами setMicrosecondPosition(long mcs) или setTickPosition(long tick).

Листинг. Проигрывание MIDI-последовательности

```
import javax.sound.midi.*;
import j ava.io.*;
Class PlayMIDK
PlayMIDKString s) {
    play(s);
}
public void play(String file) {
    try {
        File f = new File(file);
            // Получаем секвенсор по умолчанию
        Sequencer sequencer = MidiSystem.getSequencerO;
            // Проверяем, получен ли секвенсор
        if (sequencer = null) {
            System.err.println("Sequencer is not supported");
            System.exit(0);
        }
            // Открываем секвенсор
        sequencer.open();
            // Получаем MIDI-последовательность из файла
        Sequence seq = MidiSystem.getSequence(f);
            // Направляем последовательность в секвенсор
        sequencer.setSequence(seq);
            // Начинаем проигрывание
        sequencer.start();
            // Здесь надо сделать задержку на время проигрывания,
            // а затем остановить:
        sequencer.stop();
```

```
    }
    catch(Exception e) {
        System.err.println(e);
    }
}
public static void main(String[] args) {
    String s = "doom.mid";
    if (args.length > 0) s = args[0];
    new PlayMIDI(s);
}
```

18.6.4. Синтез и запись звука в Java 2

Синтез звука заключается в создании MIDI-последовательности - объекта класса sequence - каким-либо способом: с микрофона, линейного входа, синтезатора, из файла, или просто создать в программе, как это делается в листинге.

Сначала создается пустая последовательность одним из двух конструкторов:

```
Sequence(float divisionType, int resolution)
Sequence(float divisionType, int resolution, int numTracks)
```

Первый аргумент divisionType определяет способ отсчета моментов (ticks) MIDI-событий - это одна из констант:

- PPQ (Pulses Per Quarter note) - отсчеты замеряются в долях от длительности звука в четверть;
- SMPTE_24, SMPTE_25, SMPTE_so, SMPTE_30DROP (Society of Motion Picture and Television Engineers) - отсчеты в долях одного кадра, при указанном числе кадров в секунду.

Второй аргумент resolution задает количество отсчетов в указанную единицу, например,

```
Sequence seq = new Sequence)Sequence.PPQ, 10);
```

задает 10 отсчетов в звуке длительностью в четверть.

Третий аргумент numTracks определяет количество дорожек в MIDI-последовательности.

Потом, если применялся первый конструктор, в последовательности создается одна или несколько дорожек:

```
Track tr = seq.createTrack() ;
```

Если применялся второй конструктор, то надо получить уже созданные конструктором дорожки:

```
Track[] trs = seq.getTracks();
```

Затем дорожки заполняются MIDI-событиями с помощью MIDI-сообщений. Есть несколько типов сообщений для разных типов событий. Наиболее часто встречаются сообщения типа shortMessage, которые создаются конструктором по умолчанию и потом заполняются методом setMessage():

```
ShortMessage msg = new ShortMessage();
rasg.setMessage(ShortMessage.NOTEJDN, 60, 93);
```

Первый аргумент указывает тип сообщения: NOTE_ON - начать звучание, NOTE_OFF - прекратить звучание и т.д. Второй аргумент для типа NOTE_ON показывает высоту звука, в стандарте MIDI это числа от 0 до 127, 60 - нота "до" первой октавы. Третий аргумент означает "скорость" нажатия клавиши MIDI-инструмента и по-разному понимается различными устройствами.

Далее создается MIDI-событие:

```
MidiEvent me = new MidiEvent{msg, ticks);
```

Первый аргумент конструктора msg - это сообщение, второй аргумент ticks - время наступления события (в нашем примере проигрывания ноты "до") в единицах последовательности seq (в нашем примере в десятых долях четверти). Время отсчитывается от начала проигрывания последовательности.

Наконец, событие заносится на дорожку:

```
tr.add(me);
```

Указанные действия продолжаются, пока все дорожки не будут заполнены всеми событиями. В листинге это делается в цикле, но обычно MIDI-события создаются в методах обработки нажатия клавиш на обычной или специальной MIDI-клавиатуре. Еще один способ - вывести на экран изображение клавиатуры и создавать MIDI-собьгшя в методах обработки нажатий кнопки мыши на этой клавиатуре.

После создания последовательности ее можно проиграть, как в листинге, или записать в файл или выходной поток. Для этого вместо метода start() надо применить метод startRecording(), который одновременно и проигрывает последовательность, и подготавливает ее к записи, которую осуществляют статические методы:

```
write(Sequence in, int type, File out)
write(Sequence in, int type, OutputStream out)
```

Второй аргумент type задает тип MIDI-файла, который лучше всего определить для заданной последовательности seq статическим методом getMidiFiieTypes(seq). Данный метод возвращает массив возможных типов. Надо воспользоваться нулевым элементом массива,

Листинг. Создание MIDI-последовательности нот звукоряда

```
import javax.sound.midi. *;
import java.io.*;
Class SynMIDI {
    SynMIDI()  {
        play(synth());
    }
    public Sequence synth() {
        Sequence seq = null;
        try  {
                // Последовательность будет отсчитывать по 10
                // MIDI-событий на Звук длительйостью в четверть
            seq = new Sequence(Sequence.PPQ, 10);
                // Создаем в последовательности одну дорожку
            Track tr = seq.createTrack();
            for (int k = 0; k < 100; k++) {
                ShortMessage msg = new ShortMessage()
                    // Пробегаем MIDI-ноты от номера 10 до 109
                msg.setMessage(ShortMessage.NOTE_ON, 10+k, 93);
                    // Будем проигрывать ноты через каждые 5 отсчетов
                tr.add(new MidiEvent(msg, 5*k));
                msg = null;
            }
        }
        catch (Exception e) {
            System, err.printing "From synth(): "+e);
            System.exit (0);
        }
        return seq;
    }
    public void play (Sequence seq) {
        try  {
            Sequencer sequencer = MidiSystem.getSequencer();
            if (sequencer = null) {
                System.err.println("Sequencer is not supported");
                System.exit(0);
```

```
                }
                sequencer.open();
                sequencer.setSequence(seq);
                sequencer.startRecording();
                int[] type = MidiSystem.getMidiFileTypes(seq);
                MidiSystem.write(seq, type[0], new File("gammas.mid"));
            }
            catch(Exception e) {
                System.err.println("From play(): " + e);
            }
        }
        public static void main(String[] args) {
            new SynMIDI();
        }
}
```

К сожалению, объем книги не позволяет коснуться темы о работе с синтезатором (synthesizer), микширования звука, работы с несколькими инструментами и прочих возможностей Java Sound API. В документации SUN J2SDK, в каталоге docs\guide\sound\prog_guide, есть подробное руководство программиста, а в каталоге demo\sound\src лежат исходные тексты синтезатора, использующего Java Sound API.

19. Thread – нити (подпроцессы)

19.1. Понятия

Основное понятие современных операционных систем - процесс (process). Как и все общие понятия, процесс трудно определить. Можно понимать под процессом выполняющуюся (runnable) программу, но надо помнить о том, что у процесса есть несколько состояний. Процесс может в любой момент перейти к выполнению машинного кода другой программы, а также "заснуть" (sleep) на некоторое время, приостановив выполнение программы. Он может быть выгружен на диск. Количество состояний процесса и их особенности зависят от операционной системы.

Все современные операционные системы многозадачные (multitasking), они запускают и выполняют сразу несколько процессов. Одновременно может работать браузер, текстовый редактор, музыкальный проигрыватель. На экране дисплея открываются несколько окон, каждое из которых связано со своим работающим процессом.

Если на компьютере только один процессор, то он переключается с одного процесса на другой, создавая видимость одновременной работы. Переключение происходит по истечении одного или нескольких "тиков" (ticks). Размер тика зависит от тактовой частоты процессора и обычно имеет порядок 0,01 секунды. Процессам назначаются разные приоритеты (priority). Процессы с низким приоритетом не могут прервать выполнение процесса с более высоким приоритетом, они меньше занимают процессор и поэтому выполняются медленно, как говорят, "на фоне". Самый высокий приоритет у системных процессов, например, у диспетчера (scheduler), который как раз и занимается переключением процессора с процесса на процесс. Такие процессы нельзя прерывать, пока они не закончат работу, иначе компьютер быстро придет в хаотическое состояние.

Каждому процессу выделяется определенная область оперативной памяти для размещения кода программы и ее данных — его адресное пространство.

В эту же область записывается часть сведений о процессе, составляющая его контекст (context). Очень важно разделить адресные пространства разных процессов, чтобы они не могли изменить код и данные друг друга. Операционные системы по-разному относятся к обеспечению защиты адресных пространств процессов. MS Windows NT/2000 тщательно разделяют адресные пространства, тратя на это много ресурсов и времени. Это повышает надежность выполнения программы, но затрудняет создание процесса. Такие операционные системы плохо справляются с управлением большого* 4 числа процессов.

Операционные системы семейства UNIX меньше заботятся о защите памяти, но легче создают процессы и способны управлять сотней одновременно работающих процессов.

Кроме управления работой процессов операционная система должна обеспечить средства их взаимодействия: обмен сигналами и сообщениями, создание разделяемых несколькими процессами областей памяти и разделяемого исполнимого кода программы. Эти средства тоже требуют ресурсов и замедляют работу компьютера.

Работу многозадачной системы можно упростить и ускорить, если разрешить взаимодействующим процессам работать в одном адресном пространстве. Такие процессы называются threads. В русской литературе предлагаются различные переводы этого слова. Буквальный перевод — "нить". Часто переводят thread как "поток", но в этой книге мы говорим о потоке ввода/вывода. Иногда просто говорят "тред". Встречается перевод "легковесный процесс", но в некоторых операционных системах, например, Solaris, есть и thread и lightweight process. Остановимся на слове "подпроцесс".

Подпроцессы создают новые трудности для операционной системы - надо очень внимательно следить за тем, чтобы они не мешали друг другу при записи в общие участки памяти, - но зато облегчают взаимодействие подпроцессов.

Создание подпроцессов и управление ими - это дело" операционной системы, но в язык Java введены средства для выполнения этих действий. Поскольку программы, написанные на Java, должны работать во всех операционных системах, эти средства позволяют выполнять только самые общие действия.

Когда операционная система запускает виртуальную машину Java для выполнения приложения, она создает один процесс с несколькими подпроцессами. Главный (main) подпроцесс выполняет байт-коды программы, а именно, он сразу же обращается к методу main() приложения. Этот подпроцесс может породить новые подпроцессы, которые, в свою очередь, способны породить подпроцессы и т.д. Главным подпроцессом апплета является один из подпроцессов браузера, в котором апплет выполняется. Главный подпроцесс не играет никакой особой роли, просто он создается первым.

Подпроцесс в Java создается и управляется методами класса Thread. После создания объекта этого класса одним из его конструкторов новый подпроцесс запускается методом start().

Получить ссылку на текущий подпроцесс можно статическим методом

```
Thread.currentThread();
```

Класс Thread реализует интерфейс Runnabie. Этот интерфейс описывает только один метод run(). Новый подпроцесс будет выполнять то, что записано в этом методе. Впрочем, класс Thread содержит только пустую реализацию метода run(), поэтому класс Thread не используется сам по себе, он всегда расширяется. При его расширении метод run() переопределяется.

Метод run() не содержит аргументов, т.к. некому передавать их значения в метод. Он не возвращает значения, его некуда передавать. К методу run() нельзя обратиться из программы, это всегда делается автоматически исполняющей системой Java при запуске нового подпроцесса методом start().

Итак, задать действия создаваемого подпроцесса можно двумя способами: расширить класс Thread или реализовать интерфейс Runnabie. Первый способ позволяет использовать методы класса Thread для управления подпроцессом. Второй способ применяется в тех случаях, когда надо только реализовать метод run(), или класс, создающий подпроцесс, уже расширяет какой-то другой класс.

19.2. Класс Thread содержимое

В классе Thread 7 конструкторов:

- Thread(ThreadGroup group, Runnabie target, String name) - создает подпроцесс с именем name, принадлежащий группе group и выполняющий метод run() объекта target. Это основной конструктор, все остальные обращаются к нему с тем или иным параметром, равным null;
- Thread() - создаваемый подпроцесс будет выполнять свой метод run ();
- Thread(Runnabie target);
- Thread(Runnabie target, String name);
- Thread(String name);
- Thread(ThreadGroup group, Runnabie target);
- Thread(ThreadGroup group, String name).

Имя подпроцесса name не имеет никакого значения, оно не используется виртуальной машиной Java и применяется только для различения подпроцессов в программе.

После создания подпроцесса его надо запустить методом start(). Виртуальная машина Java начнет выполнять метод run() этого объекта-подпроцесса.

Подпроцесс завершит работу после выполнения метода run(). Для уничтожения объекта-подпроцесса вслед за этим он должен присвоить значение null.

Выполняющийся подпроцесс можно приостановить статическим методом sleep (long ms) на ms миллисекунд. Этот метод мы уже использовали в предыдущих

главах. Если вычислительная система может отсчитывать наносекунды, то можно приостановить подпроцесс с точностью до наносекунд методом sleep(long ms, int nanosec).

В листинге приведен простейший пример. Главный подпроцесс создает два подпроцесса с именами Thread1 и Thread2, выполняющих один и тот же метод run(). Этот метод просто выводит 20 раз текст на экран, а затем сообщает о своем завершении.

Листинг. Два подпроцесса, запущенных из главного подпроцесса

```
Class OutThread extends Thread {
    private String msg;
    OutThread(String s, String name) {
        super(name); msg = s;
    }
    public void run() {
        for(int i = 0; i < 20; i++) {
        try  {
            Thread.sleep(100);
        }
        catch(InterruptedException ie) { }
        System.out.print(msg + " ");
    }
    System.out.println("End of " + getName());
}
Class TwoThreads {
    public static void main(String[] args) {
        new OutThread("HIP", "Thread 1").start();
        new OutThread("hop", "Thread 2").start();
        System.out.println();
    }
}
```

Как же добиться согласованности, как говорят, синхронизации (synchronization) подпроцессов?

Действия, входящие в синхронизированный блок или метод образуют критический участок (critical section) программы. Несколько подпроцессов, собирающихся выполнять критический участок, встают в очередь. Это замедляет работу программы, поэтому для быстроты ее выполнения критических участков должно быть как можно меньше, и они должны быть как можно короче. Многие методы Java 2 SDK синхронизированы.

19.3. Согласование работы подпроцессов

Возможность создания многопоточных программ заложена в язык Java с самого его создания. В каждом объекте есть 3 метода wait() и 1 метод notify(), позволяющие приостановить работу подпроцесса с этим объектом, позволить другому подпроцессу поработать с объектом, а затем уведомить (notify) первый подпроцесс о возможности продолжения работы. Эти методы определены прямо в классе Object и наследуются всеми классами.

С каждым объектом связано множество подпроцессов, ожидающих доступа к объекту (wait set). Вначале этот "зал ожидания" пуст.

Основной метод wait (long miiiisec) приостанавливает текущий подпроцесс this, работающий с объектом, на miiiisec миллисекунд и переводит его в "зал ожидания", в множество ожидающих подпроцессов. Обращение к этому методу допускается только в синхронизированном блоке или методе, чтобы быть уверенными в том, что с объектом работает только один подпроцесс. По истечении miiiisec или после того, как объект получит уведомление методом notify(), подпроцесс готов возобновить работу. Если аргумент miiiisec равен о, то время ожидания не определено и возобновление работы подпроцесса возможно только после того, как объект получит уведомление методом notify().

Отличие данного метода от метода sleep() в том, что метод wait() снимает блокировку с объекта. С объектом может работать один из подпроцессов из "зала ожидания", обычно тот, который ждал дольше всех, хотя это не гарантируется спецификацией JLS.

Второй метод wai () эквивалентен wait(0). Третий метод wait(long millisec, int nanosec) уточняет задержку на nanosec наносекунд, если их сумеет отсчитать операционная система.

Метод notify() выводит из "зала ожидания" только один, произвольно выбранный подпроцесс. Метод notifyAll() выводит из состояния ожидания все подпроцессы. Эти методы тоже должны выполняться в синхронизированном блоке или методе.

Как же применить все это для согласованного доступа к объекту? Как всегда, лучше всего объяснить это на примере.

19.4. Приоритеты подпроцессов

Планировщик подпроцессов виртуальной машины Java назначает каждому подпроцессу одинаковое время выполнения процессором, переключаясь с подпроцесса на подпроцесс по истечении этого времени. Иногда необходимо выделить какому-то подпроцессу больше или меньше времени по сравнению с

другим подпроцессом. В таком случае можно задать подпроцессу больший или меньший приоритет.

В классе Thread есть 3 целые статические константы, задающие приоритеты:

- NORM_PRIORITY - обычный приоритет, который получает каждый подпроцесс при запуске, его числовое значение 5;
- MIN_PRIORITY - наименьший приоритет, его значение 1;
- MAX_PRIORITY - наивысший приоритет, его значение 10.

Кроме этих значений можно задать любое промежуточное значение от 1 до 10, но надо помнить о том, что процессор будет переключаться между подпроцессами с одинаковым высшим приоритетом, а подпроцессы с меньшим приоритетом не станут выполняться, если только не приостановлены все подпроцессы с высшим приоритетом. Поэтому для повышения общей производительности следует приостанавливать время от времени методом sleep о подпроцессы с высоким приоритетом.

Установить тот или иной приоритет можно в любое время методом setPriority(int newPriority), если подпроцесс имеет право изменить свой приоритет.

Порожденные подпроцессы будут иметь тот же приоритет, что и подпроцесс-родитель.

Итак, подпроцессы, как правило, должны работать с приоритетом NORM_PRIORITY. Подпроцессы, большую часть времени ожидающие наступления какого-нибудь события, например, нажатия пользователем кнопки Выход, могут получить более высокий приоритет MAX_PRIORITY. Подпроцессы, выполняющие длительную работу, например, установку сетевого соединения или отрисовку изображения в памяти при двойной буферизации, могут работать с низшим приоритетом MIN_PRIORITY.

19.5. Подпроцессы-демоны

Работа программы начинается с выполнения метода main() главным подпроцессом. Этот подпроцесс может породить другие подпроцессы, они, в свою очередь, способны породить свои подпроцессы. После этого главный подпроцесс ничем не будет отличаться от остальных подпроцессов. Он не следит за порожденными им подпроцессами, не ждет от них никаких сигналов. Главный подпроцесс может завершиться, а программа будет продолжать работу, пока не закончит работу последний подпроцесс.

Это правило не всегда удобно. Например, какой-то из подпроцессов может приостановиться, ожидая сетевого соединения, которое никак не может насту-

пить. Пользователь, не дождавшись соединения, прекращает работу главного подпроцесса, но программа продолжает работать.

Такие случаи можно учесть, объявив некоторые подпроцессы демонами (daemons). Это понятие не совпадает с понятием демона в UNIX. Просто программа завершается по окончании работы последнего пользовательского (user) подпроцесса, не дожидаясь окончания работы демонов. Демоны будут принудительно завершены исполняющей системой Java.

Объявить подпроцесс демоном можно сразу после его создания, перед запуском. Это делается методом setoaemon(true). Данный метод обращается к методу checkAccess() и может выбросить SecurityException.

Изменить статус демона после запуска процесса уже нельзя.

Все подпроцессы, порожденные демоном, тоже будут демонами. Для изменения их статуса необходимо обратиться к методу setDaemon(false).

19.6. Группы подпроцессов

Подпроцессы объединяются в группы. В начале работы программы исполняющая система Java создает группу подпроцессов с именем main. Все подпроцессы по умолчанию попадают в эту группу.

В любое время программа может создать новые группы подпроцессов и подпроцессы, входящие в эти группы. Вначале создается группа - экземпляр класса ThreadGroup, конструктором

ThreadGroup(String name).

При этом группа получает имя, заданное аргументом name. Затем этот экземпляр указывается при создании подпроцессов в конструкторах класса Thread. Все подпроцессы попадут в группу с именем, заданным при создании группы.

Группы подпроцессов могут образовать иерархию. Одна группа порождается от другой конструктором

ThreadGroup(ThreadGroup parent, String name).

Группы подпроцессов используются главным образом для задания приоритетов подпроцессам внутри группы. Изменение приоритетов внутри группы не будет влиять на приоритеты подпроцессов вне иерархии этой группы. Каждая группа имеет максимальный приоритет, устанавливаемый методом setMaxPriority(int maxPri) класса ThreadGroup. Ни один подпроцесс из этой группы не может превысить значения maxPri, но приоритеты подпроцессов, заданные до установки maxPri, не меняются.

19.7. Заключение

Технология Java по своей сути - многозадачная технология, основанная на Threads. Поэтому, конструируя программу для Java, следует все время помнить, что она будет выполняться в многозадачной среде. Надо ясно представлять себе, что будет, если программа начнет выполняться одновременно несколькими подпроцессами, выделять критические участки и синхронизировать их.

С другой стороны, если программа осуществляет несколько действий, следует подумать, не сделать ли их выполнение одновременным, создав дополнительные подпроцессы и распределив их приоритеты.

20. Ввод-вывод

20.1. Потоки ввода/вывода

Программы, написанные нами в предыдущих главах, воспринимали информацию из параметров командной строки и графических компонентов, а результаты выводили на консоль или в графические компоненты. Однако во многих случаях требуется выводить результаты на принтер, в файл, базу данных или передавать по сети. Исходные данные тоже часто приходится загружать из файла, базы данных или из сети.

Для того чтобы отвлечься от особенностей конкретных устройств ввода/вывода, в Java употребляется понятие потока (stream). Считается, что в программу идет входной поток (input stream) символов Unicode или просто байтов, воспринимаемый в программе методами read(). Из программы методами write() или print(), println() выводится выходной поток (output stream) символов или байтов. При этом неважно, куда направлен поток: на консоль, на принтер, в файл или в сеть, методы write() и print() ничего об этом не знают.

Можно представить себе поток как трубу, по которой в одном направлении последовательно "текут" символы или байты, один за другим. Методы read(), write(), print(), println() взаимодействуют с одним концом трубы, другой конец соединяется с источником или приемником данных конструкторами классов, в которых реализованы эти методы.

Конечно, полное игнорирование особенностей устройств ввода/вывода сильно замедляет передачу информации. Поэтому в Java все-таки выделяется файловый ввод/вывод, вывод на печать, сетевой поток.

3 потока определены в классе System статическими полями in, out и err. Их можно использовать без всяких дополнительных определений, что мы и делали на протяжении всей книги. Они называются соответственно стандартным вводом (stdin), стандартным выводом (stdout) и стандартным выводом сообщений (stderr). Эти стандартные потоки могут быть соединены с разными конкретными устройствами ввода и вывода.

Потоки out и err - это экземпляры класса PrintStream, организующего выходной поток байтов. Эти экземпляры выводят информацию на консоль методами print(), println() и write(), которых в классе PrintStream имеется около 20 для разных типов аргументов.

Поток err предназначен для вывода системных сообщений программы: трассировки, сообщений об ошибках или, просто, о выполнении каких-то этапов программы. Такие сведения обычно заносятся в специальные журналы, log-

файлы, а не выводятся на консоль. В Java есть средства переназначения потока, например, с консоли в файл.

Поток in - это экземпляр класса InputStream. Он назначен на клавиатурный ввод с консоли методами read(). Класс InputStream абстрактный, поэтому реально используется какой-то из его подклассов.

Понятие потока оказалось настолько удобным и облегчающим программирование ввода/вывода, что в Java предусмотрена возможность создания потоков, направляющих символы или байты не на внешнее устройство, а в массив или из массива, т.е. связывающих программу с областью оперативной памяти. Более того, можно создать поток, связанный со строкой типа String, находящейся, опять-таки, в оперативной памяти. Кроме того, можно создать канал (pipe) обмена информацией между подпроцессами.

Еще один вид потока - поток байтов, составляющих объект Java. Его можно направить в файл или передать по сети, а потом восстановить в оперативной памяти. Эта операция называется сериализацией (serialization) объектов.

20.2. Классы пакета java.io

Методы организации потоков собраны в классы пакета java.io.

Кроме классов, организующих поток, в пакет java.io входят классы с методами преобразования потока. Например, можно преобразовать поток байтов, образующих целые числа, в поток этих чисел.

Еще одна возможность, предоставляемая классами пакета java.io, - слить несколько потоков в один поток.

В Java есть 4 ветви иерархии классов для создания, преобразования и слияния потоков. Во главе иерархии 4 класса, непосредственно расширяющих класс Object:

- Reader - абстрактный класс, в котором собраны самые общие методы символьного ввода;
- Writer - абстрактный класс, в котором собраны самые общие методы символьного вывода;
- InputStream - абстрактный класс с общими методами байтового ввода;
- OutputStream - абстрактный класс с общими методами байтового вывода.

Классы входных потоков Reader и InputStream определяют по 3 метода ввода:

- read() - возвращает один символ или байт, взятый из входного потока, в виде целого значения типа int; если поток уже закончился, возвращает -1;

- read(chart] buf) - заполняет заранее определенный массив buf символами из входного потока; в классе InputStream массив типа byte[] и заполняется он байтами; метод возвращает фактическое число взятых из потока элементов или -1, если поток уже закончился;
- read(char[] buf, int offset, int len) - заполняет часть символьного или байтового массива buf, начиная с индекса offset, число взятых из потока элементов равно len; метод возвращает фактическое число взятых из потока элементов или -1.

Эти методы выбрасывают исключение IOException, если произошла ошибка ввода/вывода.

Четвертый метод skip(long n) "проматывает" поток с текущей позиции на п символов или байтов вперед. Эти элементы потока не вводятся методами read(). Метод возвращает реальное число пропущенных элементов, которое может отличаться от п, например поток может закончиться.

Текущий элемент потока можно пометить методом mark(int n), а затем вернуться к помеченному элементу методом reset(), но не более чем через п элементов. Не все подклассы реализуют эти методы, поэтому перед расстановкой пометок следует обратиться к логическому методу markSupported(), который возвращает true, если реализованы методы расстановки и возврата к пометкам.

Классы выходных потоков Writer и OutputStream определяют по 3 почти одинаковых метода вывода:

- write(char[] buf) - выводит массив в выходной поток, в классе OutputStream массив имеет тип byte[];
- write(char[] buf, int offset, int len) - выводит len элементов массива buf, начиная с элемента с индексом offset;
- write(int elem) в классе Writer - выводит 16, а в классе OutputStream 8 младших битов аргумента elem в выходной поток,

В классе Writer есть еще 2 метода:

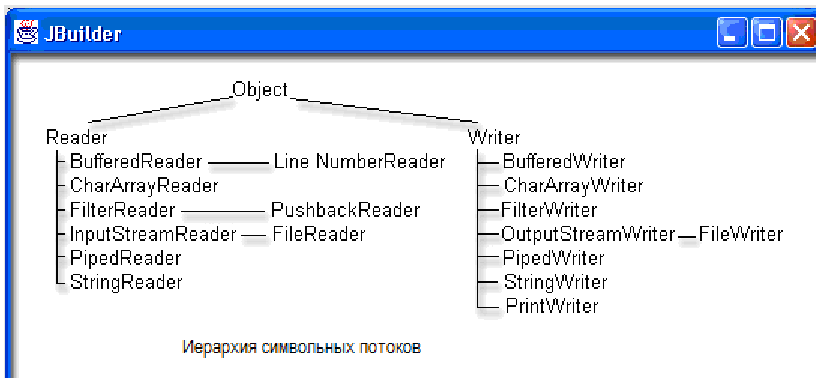
- write(string s) - выводит строку s в выходной поток;
- write(String s, int offset, int len) - выводит len символов строки s, начиная с символа с номером offset.

Многие подклассы классов Writer и OutputStream осуществляют буферизованный вывод. При этом элементы сначала накапливаются в буфере, в оперативной памяти, и выводятся в выходной поток только после того, как буфер заполнится. Это удобно для выравнивания скоростей вывода из программы и вывода потока, но часто надо вывести информацию в поток еще до заполне-

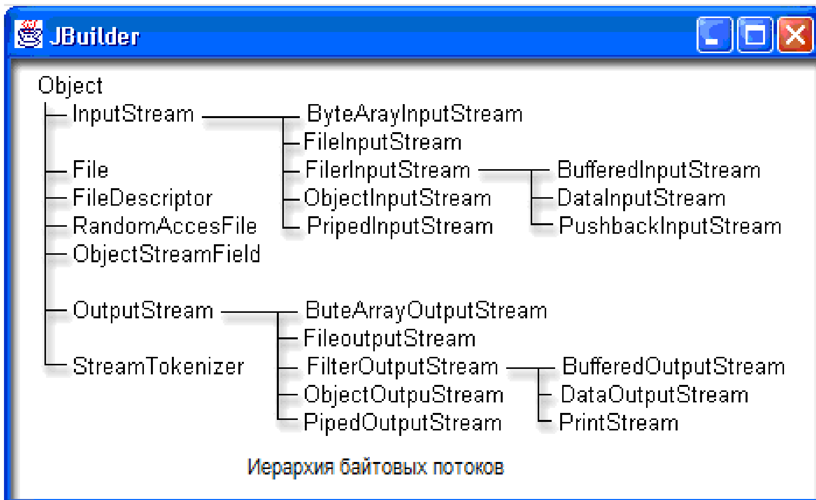
ния буфера. Для этого предусмотрен метод flush(). Данный метод сразу же выводит все содержимое буфера в поток.

Наконец, по окончании работы с потоком его необходимо закрыть методом closed.

Классы, входящие в иерархии потоков ввода/вывода:



Иерархия символьных потоков



Иерархия байтовых потоков

Все классы пакета java.io можно разделить на 2 группы: классы, создающие поток (data sink), и классы, управляющие потоком (data processing).

Классы, создающие потоки, в свою очередь, можно разделить на 5 групп:

- Классы потоков, связанных с файлами: FileReader, FileInputStream, FileWriterFile, Outputstream, RandomAccessFile.
- Классы потоков, связанных с массивами: CharArrayReader, ByteArrayInputStream, CharArrayWriter, ByteArrayOutputStream.
- Классы, создающие каналы обмена информацией между подпроцессами: PipedReader, PipedInputStream, PipedWriter, PipedOutputStream.
- Классы, создающие символьные потоки, связанные со строкой: StringReader, StringWriter
- Классы, создающие байтовые потоки из объектов Java: ObjectInputStream, ObjectOutputStream

Классы, управляющие потоком, получают в своих конструкторах уже имеющийся поток и создают новый, преобразованный поток. Можно представлять их себе как "переходное кольцо", после которого идет труба другого диаметра.

4 класса созданы специально для преобразования потоков:

- FilterReader.
- FilterInputStream.
- FilterWriter.
- FilterOutputStream

Сами по себе эти классы бесполезны - они выполняют тождественное преобразование. Их следует расширять, переопределяя методы ввода/вывода. Но для байтовых фильтров есть полезные расширения, которым соответствуют некоторые символьные классы. Перечислим их.

4 класса выполняют буферизованный ввод/вывод:

- BufferedReader.
- BufferedInputStream.
- BufferedWriter.
- BufferedOutputStream.

2 класса преобразуют поток байтов, образующих 8 простых типов Java, в эти самые типы:

- DataInputStream.
- DataOutputStream

2 класса содержат методы, позволяющие вернуть несколько символов или байтов во входной поток:

- PushbackReader.
- PushbackInputStream.

2 класса связаны с выводом на строчные устройства - экран дисплея, принтер:

- PrintWriter.
- PrintStream.

2 класса связывают байтовый и символьный потоки:

- InputStreamReader - преобразует входной байтовый поток в символьный поток;
- OutputStreamWriter - преобразует выходной символьный поток в байтовый поток.

Класс StreamTokenizer позволяет разобрать входной символьный поток на отдельные элементы (tokens) подобно тому, как класс StringTokenizer разбирал строку.

Из управляющих классов выделяется класс SequenceInputStream, сливающий несколько потоков, заданных в конструкторе, в один поток, и класс LineNumberReader, "умеющий" читать выходной символьный поток построчно. Строки в потоке разделяются символами '\n' и/или '\r'.

Этот обзор классов ввода/вывода немного проясняет положение, но не объясняет, как их использовать. Перейдем к рассмотрению реальных ситуаций.

20.3. Консольный ввод/вывод

Для вывода на консоль мы всегда использовали метод printin() класса PrintStream, никогда не определяя экземпляры этого класса. Мы просто использовали статическое поле out класса System, которое является объектом класса PrintStream. Исполняющая система Java связывает это поле с консолью.

Кстати говоря, если вам надоело писать System.out.printin(), то вы можете определить новую ссылку на system.out, например:

```
PrintStream pr = System.out;
```

и писать просто pr.println().

Консоль является байтовым устройством, и символы Unicode перед выводом на консоль должны быть преобразованы в байты. Для символов Latin 1 с кодами '\u0000' — '\u00FF' при этом просто откидывается нулевой старший байт и выводятся байты '0x00' —'0xFF'. Для кодов кириллицы, которые лежат в диапазоне '\u0400 1 —'\u04FF 1 кодировки Unicode, и других национальных алфа-

витов производится преобразование по кодовой таблице, соответствующей установленной на компьютере локали.

Конструктор класса PrintWriter, в котором задан байтовый поток, всегда неявно создает объект класса OutputStreamWriter с локальной кодировкой для преобразования байтового потока в символьный поток.

Ввод с консоли производится методами read() класса InputStream с помощью статического поля in класса System. С консоли идет поток байтов, полученных из scan-кодов клавиатуры. Эти байты должны быть преобразованы в символы Unicode такими же кодовыми таблицами, как и при выводе на консоль. Преобразование идет по той же схеме - для правильного ввода кириллицы удобнее всего определить экземпляр класса BufferedReader, используя в качестве "переходного кольца" объект класса InputStreamReader:

```
BufferedReader br = new BufferedReader(
    new InputstreamReader(System.an, "Cp866"));
```

Класс BufferedReader переопределяет 3 метода read() своего суперкласса Reader. Кроме того, он содержит метод readLine().

Метод readLine() возвращает строку типа String, содержащую символы входного потока, начиная с текущего, и заканчивая символом '\п' и/или '\r'. Эти символы-разделители не входят в возвращаемую строку. Если во входном потоке нет символов, то возвращается null.

20.4. Файловый ввод/вывод

Поскольку файлы в большинстве современных операционных систем понимаются как последовательность байтов, для файлового ввода/вывода создаются байтовые потоки с помощью классов FiieInputStream и FiieOutputStream. Это особенно удобно для бинарных файлов, хранящих байт-коды, архивы, изображения, звук.

Но очень много файлов содержат тексты, составленные из символов. Несмотря на то, что символы могут храниться в кодировке Unicode, эти тексты чаще всего записаны в байтовых кодировках. Поэтому и для текстовых файлов можно использовать байтовые потоки. В таком случае со стороны программы придется организовать преобразование байтов в символы и обратно.

Чтобы облегчить это преобразование, в пакет java.io введены классы FileReader и FileWriter. Они организуют преобразование потока: со стороны программы потоки символьные, со стороны файла - байтовые. Это происходит потому, что данные классы расширяют классы InputStreamReader и

OutputStreamWriter, соответственно, значит, содержат "переходное кольцо" внутри себя.

Несмотря на различие потоков, использование классов файлового ввода/вывода очень похоже.

В конструкторах всех 4 файловых потоков задается имя файла в виде строки типа String или ссылка на объект класса File. Конструкторы не только создают объект, но и отыскивают файл и открывают его. Например:

```
FileInputSream fis = new FileInputStream("PrWr.Java");
FileReader fr = new FileReader("D:\\jdkl.3\\src\\PrWr.Java");
```

При неудаче выбрасывается исключение класса FileNotFoundException, но конструктор класса FileWriter выбрасывает более общее исключение IOException.

После открытия выходного потока типа FileWriter или FileQutputStEeam содержимое файла, если он был не пуст, стирается. Для того чтобы можно было делать запись в конец файла, и в том и в другом классе предусмотрен конструктор с двумя аргументами. Если второй аргумент равен true, то происходит дозапись в конец файла, если false, то файл заполняется новой информацией. Например:

```
FileWriter fw = new FileWriter("ch!8.txt", true);
FileOutputStream fos = new FileOutputStream("D:\\samples\\newfile.txt");
```

Внимание. Содержимое файла, открытого на запись конструктором с одним аргументом, стирается.

Сразу после выполнения конструктора можно читать файл:

```
fis.read(); fr.read();
```

или записывать в него:

```
fos.write((char)c); fw.write((char)c);
```

По окончании работы с файлом поток следует закрыть методом close().

Преобразование потоков в классах FileReader и FileWriter выполняется по кодовым таблицам установленной на компьютере локали. Для правильного ввода кирилицы надо применять FileReader, а не FileInputStream. Если файл содержит текст в кодировке, отличной от локальной кодировки, то придется вставлять "переходное кольцо" вручную, как это делалось для консоли, например:

```
InputStreamReader isr = new InputStreamReader(fis, "KOI8_R"));
```

Байтовый поток fis определен выше.

20.5. Получение свойств файла

В конструкторах классов файлового ввода/вывода, описанных в предыдущем разделе, указывалось имя файла в виде строки. При этом оставалось неизвестным, существует ли файл, разрешен ли к, нему доступ, какова длина файла.

Получить такие сведения можно от предварительно созданного экземпляра класса File, содержащего сведения о файле. В конструкторе этого класса

```
File(String filename)
```

указывается путь к файлу или каталогу, записанный по правилам операционной системы. В UNIX имена каталогов разделяются наклонной чертой (/), в MS Windows - обратной наклонной чертой (\), в Apple Macintosh – двоеточием (:). Путь к файлу предваряется префиксом. В UNIX это наклонная черта, в MS Windows - буква раздела диска, двоеточие и обратная наклонная черта. Если префикса нет, то путь считается относительным и к нему прибавляется путь к текущему каталогу, который хранится в системном свойстве user.dir.

Конструктор не проверяет, существует ли файл с таким именем, поэтому после создания объекта следует это проверить логическим методом exists().

Класс File содержит около 40 методов, позволяющих узнать различные свойства файла или каталога.

Прежде всего, логическими методами isFile(), isDirectory() можно выяснить, является ли путь, указанный в конструкторе, путем к файлу или каталогу.

Для каталога можно получить его содержимое - список имен файлов и подкаталогов - методом list(), возвращающим массив строк string[]. Можно получить такой же список в виде массива объектов класса File[] методом listFiles(). Можно выбрать из списка только некоторые файлы, реализовав интерфейс FileNameFiiter и обратившись к методу

```
list(FileNameFilter filter).
```

Если каталог с указанным в конструкторе путем не существует, его можно создать логическим методом mkdir(). Этот метод возвращает true, если каталог удалось создать. Логический метод mkdirso создает еще и все несуществующие каталоги, указанные в пути.

Пустой каталог удаляется методом delete().

Для файла можно получить его длину в байтах методом length(), время последней модификации в секундах с 1 января 1970 г. методом lastModified(). Если файл не существует, эти методы возвращают нуль.

Логические методы canRead(), canWrite() показывают права доступа к файлу.

Файл можно переименовать логическим методом renameTo(Fiie newMame) или удалить логическим методом delete(). Эти методы возвращают true, если операция прошла удачно.

Если файл с указанным в конструкторе путем не существует, его можно создать логическим методом createNewFile(), возвращающим true, если файл не существовал и его удалось создать, и false, если файл уже существовал.

Статическими методами

```
    createTempFile(String prefix, String suffix, File tmpDir)
    createTempFile(String prefix, String suffix)
```

можно создать временный файл с именем prefix и расширением suffix в каталоге tmpDir или каталоге, указанном в системном свойстве java.io.tmpdir. Имя prefix должно содержать не менее 3 символов. Если suffix = null, то файл получит суффикс.tmp.

Перечисленные методы возвращают ссылку типа File на созданный файл. Если обратиться к методу deieteOnExit(), то по завершении работы JVM временный файл будет уничтожен.

Несколько методов getXxx() возвращают имя файла, имя каталога и другие сведения о пути к файлу. Эти методы полезны в тех случаях, когда ссылка на объект класса File возвращается другими методами и нужны сведения о файле. Наконец, метод toURL() возвращает путь к файлу в форме URL.

<u>Листинг. Определение свойств файла и каталога</u>

```
import java.io.*;
Class FileTest {
    public static void main(String[] args) throws IOException {
        PrintWriter pw = new PrintWriter(
            new OutputStreamWriter(System.out, "Cp866"), true);
        File f = new File("FileTest.Java");
        pw.println();
        pw.println("Файл \"" + f.getName() + "\" " +
            (f.exists()?"":"не ") + "существует");
        pw.println("Вы " + (f.canRead()?"":"не ") + "можете читать файл");
        pw.println("Вы " + (f.canWrite()?"":"не ") +
```

```
            "можете записывать в файл");
        pw.println("Длина файла " + f.length() + " б");
        pw.println() ;
        File d = new File("D:\\jdkl.3\\MyProgs");
        pw.println("Содержимое каталога:");
        if (d.exists() && d.isDirectory()) {
            String[] s = d.list();
            for (int i = 0; i < s.length; i++)
            pw.println(s[i]);
        }
    }
}
```

20.6. Буферизованный ввод/вывод

Операции ввода/вывода по сравнению с операциями в оперативной памяти выполняются очень медленно. Для компенсации в оперативной памяти выделяется некоторая промежуточная область — буфер, в которой постепенно накапливается информация. Когда буфер заполнен, его содержимое быстро переносится процессором, буфер очищается и снова заполняется информацией.

Классы файлового ввода/вывода не занимаются буферизацией. Для этой цели есть 4 специальных класса BufferedXxx, перечисленных выше. Они присоединяются к потокам ввода/вывода как "переходное кольцо", например:

```
    BufferedReader br = new BufferedReader(isr);
    BufferedWriter bw = new BufferedWriter(fw);
```

Потоки isr и fw определены выше.

Листинг. Буферизованный файловый ввод/вывод

```
import java.io.*;
Class DOStoUNIX {
    public static void main(String[] args) throws IOException {
        if (args.length != 2) {
            System.err.println("Usage: DOStoUNIX Cp866file KOI8_Rfile");
            System.exit(0);
        }
        BufferedReader br = new BufferedReader(
            new InputStreamReader(new FileInputStream(args[0]), "Cp866"));
        BufferedWriter bw = new BufferedWriter(
            new OutputStreamWriter(new FileOutputStreamtargs[1]), "KOI8_R"));
        int c = 0;
```

```
            while ((c = br.readO) != -1)
            bw.write((char)c);
            br.closeO; bw.close();
            System.out.println("The job's finished.");
        }
}
```

20.7. Поток простых типов Java

Класс DataOutputStream позволяет записать данные простых типов Java в выходной поток байтов методами writeBoolean(Boolean b), writeBytefint b), writeShort(int h), writeChar(int c), writelnt"(int n), writeLong(long 1), writeFloat(float f), writeDouble(double d).

Метод writeBytes(string s) записывает каждый символ строки s в один байт, отбрасывая старший байт кодировки каждого символа Unicode.

Класс DatalnputStream преобразует входной поток байтов типа InputStream, составляющих данные простых типов Java, в данные этого типа. Такой поток, как правило, создается методами класса DataOutputstream. Данные из этого потока можно прочитать методами readBoolean(), readByte(), readShort(), readChar(), readlnt(), readLong(), readFloat(), readDouble(), возвращающими данные соответствующего типа.

Методы readUnsignedByte() и readUnsignedShort() возвращают целое типа int, в котором старшие три или два байта нулевые, а младшие один или два байта заполнены байтами из входного потока.

Метод readUTF(), двойственный методу writeUTF(), возвращает строку типа String, полученную из потока, записанного методом writeUTF().

Еще один, статический, метод readUTF(Datainput in) делает то же самое со входным потоком in, записанным в кодировке UTF-8. Этот метод можно применять, не создавая объект класса DatalnputStream.

20.8. Прямой доступ к файлу

Если необходимо интенсивно работать с файлом, записывая в него данные разных типов Java, изменяя их, отыскивая и читая нужную информацию, то лучше всего воспользоваться методами класса RandomAccessFile.

В конструкторах этого класса

- RandomAccessFile(File file, String mode)
- RandomAccessFile(String fileName, String mode)

вторым аргументом mode задается режим открытия файла. Это может быть строка "r" — открытие файла только для чтения, или "rw" — открытие файла для чтения и записи.

Этот класс собрал все полезные методы работы с файлом. Он содержит все методы классов DataInputStream и DataOutputStream, кроме того, позволяет прочитать сразу целую строку методом readLine() и отыскать нужные данные в файле.

Байты файла нумеруются, начиная с 0, подобно элементам массива. Файл снабжен неявным указателем (file pointer) текущей позиции. Чтение и запись производится, начиная с текущей позиции файла. При открытии файла конструктором указатель стоит на начале файла, в позиции 0. Текущую позицию можно узнать методом getFiiePointer(). Каждое чтение или запись перемещает указатель на длину прочитанного или записанного данного. Всегда можно переместить указатель в новую позицию pos методом seek(long pos). Метод seek(0) перемещает указатель на начало файла.

В классе нет методов преобразования символов в байты и обратно по кодовым таблицам, поэтому он не приспособлен для работы с кириллицей.

20.9. Каналы обмена информацией

В предыдущей главе мы видели, каких трудов стоит организовать правильный обмен информацией между подпроцессами. В пакете java.io есть 4 класса PipedXxx, облегчающие эту задачу.

В одном подпроцессе - источнике информации - создается объект класса PipedWriter или PipedOutputStream, в который записывается информация методами write() этих классов.

В другом.подпроцессе - приемнике информации - формируется объект класса PipedReader или PipedInputStream. Он связывается с объектом-источником с помощью конструктора или специальным методом connect(), и читает информацию методами read().

Источник и приемник можно создать и связать в обратном порядке.

Так создается однонаправленный канал (pipe) информации. На самом деле это некоторая область оперативной памяти, к которой организован совместный доступ двух или более подпроцессов. Доступ синхронизируется, записывающие процессы не могут помешать чтению.

Если надо организовать двусторонний обмен информацией, то создаются два канала.

20.10. Сериализация объектов

Методы классов ObjectInputStream и ObjectOutputStream позволяют прочитать из входного байтового потока или записать в выходной байтовый поток данные сложных типов - объекты, массивы, строки - подобно тому, как методы классов DataInputStream и DataOutputStream читают и записывают данные простых типов.

Сходство усиливается тем, что классы ObjectXxx содержат методы как для чтения, так и записи простых типов. Впрочем, эти методы предназначены не для использования в программах, а для записи/чтения полей объектов и элементов массивов.

Процесс записи объекта в выходной поток получил название сериализации (serialization), а чтения объекта из входного потока и восстановления его в оперативной памяти - десериализации (deserialization).

Сериализация объекта нарушает его безопасность, поскольку зловредный процесс может сериализовать объект в массив, переписать некоторые элементы массива, представляющие private-поля объекта, обеспечив себе, например, доступ к секретному файлу, а затем десериализовать объект с измененными полями и совершить с ним недопустимые действия.

Поэтому сериализации можно подвергнуть не каждый объект, а только тот, который реализует интерфейс Seriaiizabie. Этот интерфейс не содержит ни полей, ни методов. Реализовать в нем нечего. По сути дела запись

```
    Class A implements Seriaiizabie{...}
```

это только пометка, разрешающая сериализацию класса А.

Как всегда в Java, процесс сериализации максимально автоматизирован. Достаточно создать объект класса ObjectOutputStream, связав его с выходным потоком, и выводить в этот поток объекты методом writeObject():

```
MyClass me = new MyClass("abc", -12, 5.67e-5);
int[] arr = {10, 20, 30};
ObjectOutputStream oos = new ObjectOutputStream(
    new FileOutputStream("myObjects.ser")) ;
oos.writeObject(me);
oos.writeObject(arr);
oos.writeObject("Some string");
oos.writeObject (new Date());
oos.flush();
```

В выходной поток выводятся все нестатические поля объекта, независимо от прав доступа к ним, а также сведения о классе этого объекта, необходимые для его правильного восстановления при десериализации. Байт-коды методов класса не сериализуются.

Если в объекте присутствуют ссылки на другие объекты, то они тоже сериализуются, а в них могут быть ссылки на другие объекты, которые опять-таки сериализуются, и получается целое множество причудливо связанных между собой сериализуемых объектов. Метод writeObject() распознает две ссылки на один объект и выводит его в выходной поток только один раз. К тому же, он распознает ссылки, замкнутые в кольцо, и избегает зацикливания.

Все классы объектов, входящих в такое сериализуемое множество, а также все их внутренние классы, должны реализовать интерфейс seriaiizabie, в противном случае будет выброшено исключение класса NotseriaiizabieException и процесс сериализации прервется. Многие классы J2SDK реализуют этот интерфейс. Учтите также, что все потомки таких классов наследуют реализацию. Например, класс java.awt.Component реализует интерфейс Serializable, значит, все графические компоненты можно сериализовать. Не реализуют этот интерфейс обычно классы, тесно связанные с выполнением программ, например, java.awt.Toolkit. Состояние экземпляров таких классов нет смысла сохранять или передавать по сети. Не реализуют интерфейс Serializable и классы, содержащие внутренние сведения Java "для служебного пользования".

Десериализация происходит так же просто, как и сериализация:

```
ObjectInputStream ois = new ObjectInputStream(
    new FileInputStream("myObjects.ser"));
MyClass mcl = (MyClass)ois.readObject();
int[] a = (int[])ois.readObject();
String s = (String)ois.readObject();
Date d = (Date)ois.readObject() ;
```

Нужно только соблюдать порядок чтения элементов потока.

Если не нужно сериализовать какое-то поле, то достаточно пометить его служебным словом transient, например:

```
    transient MyClass me = new MyClass("abc", -12, 5.67e-5);
```

Метод writeObject() не записывает в выходной поток поля, помеченные static и transient. Впрочем, это положение можно изменить, переопределив метод writeObject() или задав список сериализуемых полей.

Вообще процесс сериализации можно полностью настроить под свои нужды, переопределив методы ввода/вывода и воспользовавшись вспомогательными

классами. Можно даже взять весь процесс на себя, реализовав не интерфейс Serializable, а интерфейс Externaiizabie, но тогда придется реали-зовать методы readExternal() и writeExternal(), выполняющие ввод/вывод.

20.11. Печать в Java

Поскольку принтер - устройство графическое, вывод на печать очень похож на вывод графических объектов на экран. Поэтому в Java средства печати входят в графическую библиотеку AWT и в систему Java 2D.

В графическом компоненте кроме графического контекста - объекта класса Graphics, создается еще "печатный контекст". Это тоже объект класса Graphics, но реализующий интерфейс printGraphics и полученный из другого источника - объекта класса PrintJob, входящего в пакет java.awt. Сам же этот объект создается с помощью класса Toolkit пакета java.awt. На практике это выглядит так:

```
PrintJob pj = getToolkitO.get,Print Job (this, "Job Title", null);
Graphics pg = pj.getGraphics();
```

Метод getPrintJob() сначала выводит на экран стандартное окно Печать (Print) операционной системы. Когда пользователь выберет в этом окне параметры печати и начнет печать кнопкой ОК, создается объект pj. Если пользователь отказывается от печати при помощи кнопки Отмена (Cancel), то метод возвращает null.

В классе Toolkit 2 метода getPrint Job():

- getPrintJob(Frame frame, String jobTitle, JobAttributes jobAttr, PageAttributes pageAttr)
- getPrintJob(Frame frame, String jobTitle, Properties prop)

Аргумент frame указывает на окно верхнего уровня, управляющее печатью. Этот аргумент не может быть null. Строка jobTitle задает заголовок задания, который не печатается, и может быть равна null. Аргумент prop зависит от реализации системы печати, часто это просто null, в данном случае задаются стандартные параметры печати.

Аргумент jobAttr задает параметры печати. Класс JobAttributes, экземпляром которого является этот аргумент, устроен сложно. В нем 5 подклассов, содержащих статические константы - параметры печати, которые используются в конструкторе класса. Впрочем, есть конструктор по умолчанию, задающий стандартные параметры печати.

Аргумент pageAttr задает параметры страницы. Класс pageProperties тоже содержит 5 подклассов со статическими константами, которые и задают параметры страницы и используются в конструкторе класса. Если для печати достаточно стандартных параметров, то можно воспользоваться конструктором по умолчанию.

После того как "печатный контекст" - объект pg класса Graphics - определен, можно вызывать метод print(pg) или printAll(pg) класса Component. Этот метод устанавливает связь с принтером по умолчанию и вызывает метод paint(pg). На печать выводится все то, что задано этим методом.

Например, чтобы распечатать текстовый файл, надо в процессе ввода разбить его текст на строки и в методе paint(pg) вывести строки методом pg.drawString(). При этом следует учесть, что в "печатном контексте" нет шрифта по умолчанию, всегда надо устанавливать шрифт методом pg.setFont().

После выполнения всех методов print о применяется метод pg.dispose(), вызывающий прогон страницы, и метод pj.end(), заканчивающий печать.

В листинге 18.7 приведен простой пример печати текста и окружности, заданных в методе paint (>. Этот метод работает два раза: первый раз вычерчивая текст и окружность на экране, второй раз, точно так же, на листе бумаги, вставленной в принтер. Все методы печати собраны в один метод simplePrint().

20.12. Печать средствами Java 2D

Расширенная графическая система Java 2D предлагает новые интерфейсы и классы для печати, собранные в пакет java.awt.print. Эти классы полностью перекрывают все стандартные возможности печати библиотеки AWT. Более того, они удобнее в работе и предлагают дополнительные возможности. Если этот пакет установлен в вашей вычислительной системе, то, безусловно, нужно применять его, а не стандартные средства печати AWT.

Как и стандартные средства AWT, методы классов Java 2D выводят на печать содержимое графического контекста, заполненного методами класса Graphics или класса Graphics2D.

Всякий класс Java 2D, собирающийся печатать хотя бы одну страницу текста, графики или изображения называется классом, рисующим страницы (page painter). Такой класс должен реализовать интерфейс Printable. В этом интерфейсе описаны 2 константы и только 1 метод print(). Класс, рисующий страницы, должен реализовать этот метод. Метод print() возвращает целое типа int и имеет 3 аргумента:

print(Graphics g, PageFormat pf, int ind);

Первый аргумент g - это графический контекст, выводимый на лист бумаги, второй аргумент pf - экземпляр класса PageFormat, определяющий размер и ориентацию страницы, третий аргумент ind - порядковый номер страницы, начинающийся с нуля.

Метод print() класса, рисующего страницы, заменяет собой метод paint(), использовавшийся стандартными средствами печати AWT. Класс, рисующий страницы, не обязан расширять класс Frame и переопределять метод paint(). Все заполнение графического контекста методами класса Graphics или Graphics2D теперь выполняется в методе print().

Когда печать страницы будет закончена, метод print() должен возвратить целое значение, заданное константой PAGE_EXISTS. Будет сделано повторное обращение к методу print() для печати следующей страницы. Аргумент ind при этом возрастет на 1. Когда ind превысит количество страниц, метод print() должен возвратить значение NO_SUCH_PAGE, что служит сигналом окончания печати.

Следует помнить, что система печати может несколько раз обратиться к методу paint() для печати одной и той же страницы. При этом аргумент ind не меняется, а метод print() должен создать тот же графический контекст.

Класс PageFormat определяет параметры страницы. На странице вводится система координат с единицей длины 1/72 дюйма, начало которой и направление осей определяется одной из 3 констант:

- PORTRAIT — начало координат расположено в левом верхнем углу страницы, ось Ox направлена вправо, ось Oy — вниз;
- LANDSCAPE — начало координат в левом нижнем углу, ось Ox идет вверх, ось Oy — вправо;
- REVERSE_LANDSCAPE — начало координат в правом верхнем углу, ось Ox идет вниз, ось Oy — влево.

Большинство принтеров не может печатать без полей, на всей странице, а осуществляет вывод только в некоторой области печати (imageable area), координаты левого верхнего угла которой возвращаются методами getImageableX() и getImageableY(), а ширина и высота - методами getImageableWidth() и getImageableHeight().

Эти значения надо учитывать при расположении элементов в графическом контексте, например, при размещении строк текста методом drawString(),

В классе только 1 конструктор по умолчанию PageFormat(), задающий стандартные параметры страницы, определенные для принтера по умолчанию вычислительной системы.

Читатель, добравшийся до этого места книги, уже настолько поднаторел в Java, что у него возникает вопрос: "Как же тогда задать параметры страницы?" Ответ простой: "С помощью стандартного окна операционной системы".

Метод pageDiaiog(PageDiaiog pd) открывает на экране стандартное окно Параметры страницы (Page Setup) операционной системы, в котором уже заданы параметры, определенные в объекте pd. Если пользователь выбрал в этом окне кнопку Отмена, то возвращается ссылка на объект pd, если кнопку ОК, то создается и возвращается ссылка на новый объект. Объект pd в любом случае не меняется. Он обычно создается конструктором.

Итак, параметры страницы определены, метод print() тоже. Теперь надо дать задание на печать (print job) — указать количество страниц, их номера, порядок печати страниц, количество копий. Все эти сведения собираются в классе PrinterJob.

Система печати Java 2D различает 2 вида заданий. В более простых заданиях - PrintableJob - есть только один класс, рисующий страницы, поэтому у всех страниц одни и те же параметры, страницы печатаются последовательно с первой по последнюю или с последней страницы по первую, это зависит от системы печати.

Второй, более сложный вид заданий - PageableJob - определяет для печати каждой страницы свой класс, рисующий страницы, поэтому у каждой страницы могут быть собственные параметры. Кроме того, можно печатать не все, а только выбранные страницы, выводить их в обратном порядке, печатать на обеих сторонах листа. Для осуществления этих возможностей определяется экземпляр класса Book или создается класс, реализующий интерфейс Pageable.

В классе Book, опять-таки, 1 конструктор, создающий пустой объект:

```
Book b = new Book()
```

После создания в данный объект добавляются классы, рисующие страницы. Для этого в классе Book есть 2 метода:

- append (Printable p, PageFormat pf) - добавляет объект p в конец;
- append(Printable p, PageFormat pf, int numPages) - добавляет numPages экземпляров p в конец; если число страниц заранее не известно, то задается константа UNKNOWN_NUMBER_OF_PAGES.

При составлении задания на печать, т.е. после создания экземпляра класса PrinterJob, надо указать вид задания одним и только одним из 3 методов этого класса setPrintable(Printable pr), setPrintable(Printable pr, PageFormat pf) или setPageble (Pageable pg). Заодно задаются один или несколько классов рг, рисующих страницы в этом задании.

Остальные параметры задания можно задать в стандартном диалоговом окне Печать (Print) операционной системы, которое открывается на экране при выполнении логического метода printoiaiog (). Указанный метод не имеет аргументов. Он возвратит true, когда пользователь щелкнет по кнопке ОК, и false после нажатия кнопки Отмена.

Остается задать число копий, если оно больше 1, методом setCopies(int n) и задание сформировано.

Еще один полезный метод defaultPage() класса PrinterJob возвращает объект класса PageFormat по умолчанию. Этот метод можно использовать вместо конструктора класса PageFormat.

Осталось сказать, как создается экземпляр класса PrinterJob. Поскольку этот класс тесно связан с системой печати компьютера, его объекты создаются не конструктором, а статическим методом getPrinterJob(), Имеющимся в том же самом классе Printer Job.

Начало печати задается методом print() класса PrinterJob. Этот метод не имеет аргументов. Он.последбватель но вызывает методы print (g, pf, ind) классов, рисующих страницы, для каждой страницы.

21. Сетевые средства

21.1. Введение

Когда число компьютеров в учреждении переваливает за десяток и сотрудникам надоедает бегать с дискетами друг к другу для обмена файлами, тогда в компьютеры вставляются сетевые карты, протягиваются кабели и компьютеры объединяются в сеть. Сначала все компьютеры в сети равноправны, они делают одно и то же — это одноранговая (peer-to-peer) сеть. Потом покупается компьютер с большими и быстрыми жесткими дисками, и все файлы учреждения начинают храниться на данных дисках — этот компьютер становится файл-сервером, предоставляющим услуги хранения, поиска, архивирования файлов. Затем покупается дорогой и быстрый принтер. Компьютер, связанный с ним, становится принт-сервером, предоставляющим услуги печати. Потом появляются графический сервер, вычислительный сервер, сервер базы данных. Остальные компьютеры становятся клиентами этих серверов. Такая архитектура сети называется архитектурой клиент-сервер (client-server).

Сервер постоянно находится в состоянии ожидания, он прослушивает (listen) сеть, ожидая запросов от клиентов. Клиент связывается с сервером и посылает ему запрос (request) с описанием услуги, например, имя нужного файла. Сервер обрабатывает запрос и отправляет ответ (response), в нашем примере, файл, или сообщение о невозможности оказать услугу. После этого связь может быть разорвана или продолжиться, организуя сеанс связи, называемый сессией (session).

Запросы клиента и ответы сервера формируются по строгим правилам, совокупность которых образует протокол (protocol) связи. Всякий протокол должен, прежде всего, содержать правила соединения компьютеров. Клиент перед посылкой запроса должен удостовериться, что сервер в рабочем состоянии, прослушивает сеть, и услышал клиента. Послав запрос, клиент должен быть уверен, что запрос дошел до сервера, сервер понял запрос и готов ответить на него. Сервер обязан убедиться, что ответ дошел до клиента. Окончание сессии должно быть четко зафиксировано, чтобы сервер мог освободить ресурсы, занятые обработкой запросов клиента.

Все правила, образующие протокол, должны быть понятными, однозначными и короткими, чтобы не загружать сеть. Поэтому сообщения, пересылаемые по сети, напоминают шифровки, в них имеет значение каждый бит.

Итак, все сетевые соединения основаны на трех основных понятиях: клиент, сервер и протокол. Клиент и сервер — понятия относительные. В одной сессии компьютер может быть сервером, а в другой — клиентом. Например, файл-сервер может послать принт-серверу файл на печать, становясь его клиентом.

Для обслуживания протокола: формирования запросов и ответов, проверок их соответствия протоколу, расшифровки сообщений, связи с сетевыми устройствами создается программа, состоящая из двух частей. Одна часть программы работает на сервере, другая — на клиенте. Эти части так и называются серверной частью программы и клиентской частью программы, или, короче, сервером и клиентом.

Очень часто клиентская и серверная части программы пишутся отдельно, разными фирмами, поскольку от этих программ требуется только, чтобы они соблюдали протокол. Более того, по каждому протоколу работают десятки клиентов и серверов, отличающихся разными удобствами.

Обычно на одном компьютере-сервере работают несколько программ-серверов. Одна программа занимается электронной почтой, другая — пересылкой файлов, третья предоставляет Web-страницы. Для того чтобы их различать, каждой программе-серверу придается номер порта (port). Это просто целое положительное число, которое указывает клиент, обращаясь к определенной программе-серверу. Число, вообще говоря, может быть любым, но наиболее распространенным протоколам даются стандартные номера, чтобы клиенты были твердо уверены, что обращаются к нужному серверу. Так, стандартный номер порта электронной почты 25, пересылки файлов - 21, Web-сервера - 80. Стандартные номера простираются от 0 до 1023. Числа, начиная с 1024 до 65 535, можно использовать для своих собственных номеров портов.

Все это похоже на телевизионные каналы. Клиент-телевизор обращается посредством антенны к серверу-телецентру и выбирает номер канала. Он уверен, что на первом канале ОРТ, на втором — РТР и т.д.

Чтобы равномерно распределить нагрузку на сервер, часто несколько портов прослушиваются программами-серверами одного типа. Web-сервер, кроме порта с номером 80, может прослушивать порт 8080, 8001 и еще какой-нибудь другой.

В процессе передачи сообщения используется несколько протоколов. Даже когда мы отправляем письмо, мы сначала пишем сообщение, начиная его: "Глубокоуважаемый Иван Петрович!" и заканчивая: "Искренне преданный Вам". Это один протокол. Можно начать письмо словами: "Вася, привет!" и закончить: "Ну, пока". Это другой протокол. Потом мы помещаем письмо в конверт и пишем на нем адрес по протоколу, предложенному Министерством связи. Затем письмо попадает на почту, упаковывается в мешок, на котором пишется адрес по протоколу почтовой связи. Мешок загружается в самолет, который перемещается по своему протоколу. Заметьте, что каждый протокол только добавляет к сообщению свою информацию, не меняя его, ничего не зная о том, что сделано по предыдущему протоколу и что будет сделано по правилам следующе-

го протокола. Это очень удобно — можно программировать один протокол, ничего не зная о других протоколах.

Прежде чем дойти до адресата, письмо проходит обратный путь: вынимается из самолета, затем из мешка, потом из конверта. Поэтому говорят о стеке (stack) протоколов: "Первым пришел, последним ушел".

В современных глобальных сетях принят стек из четырех протоколов, называемый стеком протоколов TCP/IP.

Сначала мы пишем сообщение, пользуясь программой, реализующей прикладной (application) протокол: HTTP (80), SMTP (25), TELNET (23), FTP (21), POP3 (100) или другой протокол. В скобках записан стандартный номер порта.

Затем сообщение обрабатывается по транспортному (transport) протоколу. К нему добавляются, в частности, номера портов отправителя и получателя, контрольная сумма и длина сообщения. Наиболее распространены транспортные протоколы TCP (Transmission Control Protocol) и UDP (User Datagram Protocol). В результате работы протокола TCP получается TCP-пакет (packet), а протокола UDP — дейтаграмма (datagram).

Дейтаграмма невелика — всего около килобайта, поэтому сообщение делится на прикладном уровне на части, из которых создаются отдельные дейтаграммы. Дейтаграммы посылаются одна за другой. Они могут идти к получателю разными маршрутами, прийти совсем в другом порядке, некоторые дейтаграммы могут потеряться. Прикладная программа получателя должна сама позаботиться о том, чтобы собрать из полученных дейтаграмм исходное сообщение. Для этого обычно перед посылкой части сообщения нумеруются, как страницы в книге. Таким образом, протокол UDP работает как почтовая служба. Посылая книгу, мы разрезаем ее на страницы, каждую страницу отправляем в своем конверте, и никогда не уверены, что все письма дойдут до адресата.

TCP-пакет тоже невелик, и пересылка также идет отдельными пакетами, но протокол TCP обеспечивает надежную связь. Сначала устанавливается соединение с получателем. Только после этого посылаются пакеты. Получение каждого пакета подтверждается получателем, при ошибке посылка пакета повторяется. Сообщение аккуратно собирается получателем. Для отправителя и получателя создается впечатление, что пересылаются не пакеты, а сплошной поток байтов, поэтому передачу сообщений по протоколу TCP часто называют передачей потоком. Связь по протоколу TCP больше напоминает телефонный разговор, чем почтовую связь.

Далее сообщением занимается программа, реализующая сетевой (network) протокол. Чаще всего это протокол IP (Internet Protocol). Он добавляет к сооб-

щению адрес отправителя и адрес получателя, и другие сведения. В результате получается IP-пакет.

Наконец, IP-пакет поступает к программе, работающей по канальному (link) протоколу ENET, SLIP, PPP, и сообщение принимает вид, пригодный для передачи по сети.

На стороне получателя сообщение проходит через эти четыре уровня протоколов в обратном порядке, освобождаясь от служебной информации, и доходит до программы, реализующей прикладной протокол.

Какой же адрес заносится в IP-пакет? Каждый компьютер или другое устройство, подключенное к объединению сетей Internet, так называемый хост (host), получает уникальный номер — четырехбайтовое целое число, называемое IP-адресом (IP-address). По традиции содержимое каждого байта записывается десятичным числом от 0 до 255, называемым октетом (octet), и эти числа пишутся через точку: 138.2.45.12 или 17.056.215.38.

IP-адрес удобен для машины, но неудобен для человека. Представьте себе рекламный призыв: "Заходите на наш сайт 154.223.145.26!" Поэтому IP-адрес хоста дублируется доменным именем (domain name).

В доменном имени присутствует краткое обозначение страны: ru — Россия, su — Советский Союз, ua — Украина, de — ФРГ и т.д., или обозначение типа учреждения: com — коммерческая структура, org — общественная организация, edu — образовательное учреждение. Далее указывается регион: msc.ru — Москва, spb.ru — Санкт-Петербург, kcn.ru — Казань, или учреждение: bhv.ru — "БХВ-Петербург", ksu.ru — Казанский госуниверситет, sun.com — SUN Microsystems. Потом подразделение: www.bhv.ru, java.sun.com. Такую цепочку кратких обозначений можно продолжать и дальше.

В Java IP-адрес и доменное имя объединяются в один класс inetAddress пакета java.net. Экземпляр этого класса создается статическим методом getByName (string host) данного же класса, в котором аргумент host— это доменное имя или IP-адрес.

21.2. Работа в WWW

Среди программного обеспечения Internet большое распространение получила информационная система WWW (World Wide Web), основанная на прикладном протоколе HTTP (Hypertext Transfer Protocol). В ней используется расширенная адресация, называемая URL (Uniform Resource Locator). Эта адресация имеет такие схемы:

- protocol://authority@host:port/path/file#ref

- protocol://authority@host:port/path/file/extra_path?info

Здесь необязательная часть authority - это пара имя:пароль для доступа к хосту, host — это IP-адрес или доменное имя хоста. Например:

- http://www.bhv.ru/
- http://132.192.5.10:8080/public/some.html
- ftp://guest:password@lenta.ru/users/local/pub
- ffle:///C:/text/html/index.htm

Если какой-то элемент URL отсутствует, то берется стандартное значение. Например, в первом примере номер порта port равен 80, а имя файла path — какой-то головной файл, определяемый хостом, чаще всего это файл с именем index.html. В третьем примере номер порта равен 21. В последнем примере в форме URL просто записано имя файла index.htm, расположенного на разделе С: жесткого диска той же самой машины.

В Java для работы с URL есть класс URL пакета java.net. Объект этого класса создается одним из 6 конструкторов. В основном конструкторе

URL(String url)

задается расширенный адрес url в виде строки. Кроме методов доступа getXxx(), позволяющих получить элементы URL, в этом классе есть 2 интересных метода:

- openConnection- () - определяет связь с URL и возвращает объект класса URLConnection;
- openStream() - устанавливает связь с URL и открывает входной поток в виде возвращаемого объекта класса inputstream.

Листинг показывает, как легко можно получить файл из Internet, пользуясь методом openStream().

Листинг. Получение Web-страницы

```
import java.net.*;
import j ava.io.*;
Class SimpleURL {
    public static void main(String[] args) {
        try  {
            URL bhv = new URL(" http://www.bhv.ru/ ");
            BufferedReader br = new BufferedReader(
                new InputStreamReader(bhv.openStream()));
            String line;
            while ((line = br.readLine()) != null)
```

```
                System.out.println(line);
                br.close();
            }
            catch(MalformedURLException me) {
                System.err.println("Unknown host: " + me);
                System.exit(0);
            }
            catch(IOException ioe) {
                System.err.println("Input error: " + ioe);
            }
        }
}
```

Если вам надо не только получить информацию с хоста, но и узнать ее тип: текст, гипертекст, архивный файл, изображение, звук, или выяснить длину файла, или передать информацию на хост, то необходимо сначала методом openConnection() создать объект класса URLConnection или его подкласса HttpURLConnection.

После создания объекта соединение еще не установлено, и можно задать параметры связи. Это делается следующими методами:

- setDoOutput(Boolean out) - если аргумент out равен true, то передача пойдет от клиента на хост; значение по умолчанию false;
- setDoinput(Boolean in) - если аргумент in равен true, то передача пойдет с хоста к клиенту; значение по умолчанию true, однано если уже выполнено setDoOutput(true), то значение по умолчанию равно false;
- setUseCaches(Boolean cache) - если аргумент cache равен false, то передача пойдет без кэширования, если true, то принимается режим по умолчанию;
- setDefaultUseCaches(Boolean default) - если аргумент default равен true, то принимается режим кэширования, предусмотренный протоколом;
- setRequestProperty(String name, String value) - добавляет параметр name со значением value к заголовку посылаемого сообщения.

После задания параметров нужно установить соединение методом connect(). После соединения задание параметров уже невозможно. Следует учесть, что некоторые методы доступа getxxxo, которым надо получить свои значения с хоста, автоматически устанавливают соединение, и обращение к методу connect() становится излишним.

Web-сервер возвращает информацию, запрошенную клиентом, вместе с заголовком, сведения из которого можно получить методами getXxx(), например:

- getcontentType() - возвращает строку типа string, показывающую тип пересланной информации, например, "text/html", или null, если сервер его не указал;
- getcontentLength() - возвращает длину полученной информации в байтах или 1, если сервер ее не указал;
- getcontent() - возвращает полученную информацию в виде объекта типа Object;
- getContentEncoding() - возвращает строку типа string с кодировкой полученной информации, или null, если сервер ее не указал.

2 метода возвращают потоки ввода/вывода, созданные для данного соединения:

- getInputStream() - возвращает входной поток типа InputStream;
- getOutputStream() - возвращает выходной поток типа OutputStream.

Прочие методы, а их около двадцати, возвращают различные параметры соединения.

Обращение к методу bhv.openStream() - это, на самом деле, сокращение записи

```
bhv.openConnection().getInputStream()
```

В листинге показано, как переслать строку текста по адресу URL.

Web-сервер, который получает эту строку, не знает, что делать с полученной информацией. Занести ее в файл? Но с каким именем, и есть ли у него право создавать файлы? Переслать на другую машину? Но куда?

Выход был найден в системе CGI (Common Gateway Interface), которая вкратце действует следующим образом. При посылке сообщения мы указываем URL исполнимого файла некоторой программы, размещенной на машине-сервере. Получив сообщение, Web-сервер запускает эту программу и передает сообщение на ее стандартный ввод. Вот программа-то и знает, что делать с полученным сообщением. Она обрабатывает сообщение и выводит результат обработки на свой стандартный вывод. Web-сервер подключается к стандартному выводу, принимает результат и отправляет его обратно клиенту.

CGI-программу можно написать на любом языке: C, C++, Pascal, Perl, PHP, лишь бы у нее был стандартный ввод и стандартный вывод. Можно написать ее и на Java, но в технологии Java есть более изящное решение этой задачи с помощью сервлетов (servlets). CGI-программы обычно лежат на сервере в каталоге cgi-bin.

Листинг. Посылка строки по адресу URL

```
import java.net.*;
import java.io.*;
Class PostURL {
    public static void main(String[] args) {
        String req = "This text is posting to URL";
        try  {
                // Указываем URL нужной CGI-программы
            URL url = new URL(" http://www.bhv.ru/cgi-bin/some.pl ");
                // Создаем объект uc
            URLConnection uc = url.openConnection();
                // Собираемся отправлять
            uc.setDoOutput(true);
                // и получать сообщения
            uc.setDoInput(true);
                // без кэширования
            uc.setUseCaches(false);
                // Задаем тип
            uc.setRequestProperty("content-type", "application/octet-stream");
                // и длину сообщения
            uc.setRequestProperty("content-length", "" + req.length());
                // Устанавливаем соединение
            uc.connect();
                // Открываем выходной поток
            DataOutputStream dos = new DataOutputStream(
                uc.getOutputStreamO);
                // и выводим в него сообщение, посылая его на адрес  URL.
            dos.writeBytes(req);
                // Закрываем выходной поток
            dos.close();
                // Открываем входной поток для ответа сервера
            BufferedReader br = new BufferedReader(new InputStreamReader(
                uc.getInputStream() )) ;
            String res = null;
                // Читаем ответ сервера и выводим его на консоль
            while ((res = br.readLine()) != null)
            System.out.println(res);
            br.close () ;
        }
        catch(MalformedURLException me) {
            System.err.println(me);
        }
        catch(UnknownHostException he)  {
```

```
            System.err.println(he);
        }
        catch(UnknownServiceException se) {
            System.err.println(se);
        }
        catch(IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

21.3. Работа по протоколу TCP

Программы-серверы, прослушивающие свои порты, работают под управлением операционной системы. У машин-серверов могут быть самые разные операционные системы, особенности которых передаются программам-серверам.

Чтобы сгладить различия в реализациях разных серверов, между сервером и портом введен промежуточный программный слой, названный сокетом (socket). Английское слово socket переводится как электрический разъем, розетка. Так же как к розетке при помощи вилки можно подключить любой электрический прибор, лишь бы он был рассчитан на 220 В и 50 Гц, к соке-ту можно присоединить любой клиент, лишь бы он работал по тому же протоколу, что и сервер. Каждый сокет связан (bind) с одним портом, говорят, что сокет прослушивает (listen) порт. Соединение с помощью сокетов устанавливается так.

1. Сервер создает сокет, прослушивающий порт сервера.
2. Клиент тоже создает сокет, через который связывается с сервером, сервер начинает устанавливать (accept) связь с клиентом.
3. Устанавливая связь, сервер создает новый сокет, прослушивающий порт с другим, новым номером, и сообщает этот номер клиенту.
4. Клиент посылает запрос на сервер через порт с новым номером.

После этого соединение становится совершенно симметричным — два сокета обмениваются информацией, а сервер через старый сокет продолжает прослушивать прежний порт, ожидая следующего клиента.

В Java сокет — это объект класса socket из пакета java.io. В классе шесть конструкторов, в которые разными способами заносится адрес хоста и номер порта. Чаще всего применяется конструктор

```
Socket(String host, int port)
```

Многочисленные методы доступа устанавливают и получают параметры сокета. Мы не будем углубляться в их изучение. Нам понадобятся только методы, создающие потоки ввода/вывода:

- getInputStream() — возвращает входной поток типа InputStream;
- getOutputStream() — возвращает выходной поток типа OutputStream.

Приведем пример получения файла с сервера по максимально упрощенному протоколу HTTP.

1. Клиент посылает серверу запрос на получение файла строкой "POST filename HTTP/1.l\n\n", где filename — строка с путем к файлу на сервере.
2. Сервер анализирует строку, отыскивает файл с именем filename и возвращает его клиенту. Если имя файла filename заканчивается наклонной чертой /, то сервер понимает его как имя каталога и возвращает файл index.html, находящийся в этом каталоге.
3. Перед содержимым файла сервер посылает строку вида "HTTP/1.1 code OK\n\n", где code — это код ответа, одно из чисел: 200 — запрос удовлетворен, файл посылается; 400 — запрос не понят; 404 — файл не найден.
4. Сервер закрывает сокет и продолжает слушать порт, ожидая следующего запроса.
5. Клиент выводит содержимое полученного файла в стандартный вывод System, out или выводит код сообщения сервера в стандартный вывод сообщений System, err.
6. Клиент закрывает сокет, завершая связь.

Этот протокол реализуется в клиентской программе листинга и серверной программе листинга.

Листинг. Упрощенный HTTP-клиент

```
import java.net.*;
import java.io.*;
import java.util.*;
Class Client {
    public static void main(String[] args) {
        if (args.length != 3) {
            System.err.println("Usage: Client host port file");
            System.exit(0) ;
        }
        String host = args[0];
        int port = Integer.parseInt(args[1]);
        String file = args[2];
        try  {
```

```
            Socket sock = new Socket(host, port);
            PrintWriter pw = new PrintWriter(new OutputStreamWriter(
            sock.getOutputStreamf)), true);
            pw.println("POST " + file + " HTTP/1.l\n");
            BufferedReader br = new BufferedReader(new InputStreamReader(
                sock.getInputStream() ) ) ;
            String line = null;
            line = br.readLine();
            StringTokenizer st = new StringTokenizer(line);
            String code = null;
            if ((st.countTokens() >= 2) && st.nextToken().equals("POST")) {
                if ((code = st.nextToken()) != "200") {
                    System.err.println("File not found, code = " + code);
                    System.exit (0);
                }
            }
            while ((line = br.readLine()) != null) System.out.println{line);
            sock.close();
        }
        catch(Exception e) {
            System.err.println(e);
        }
    }
}
```

Закрытие потоков ввода/вывода вызывает закрытие сокета. Обратно, закрытие сокета закрывает и потоки.

Для создания сервера в пакете java.net есть класс serversocket. В конструкторе этого класса указывается номер порта

```
ServerSocket(int port)
```

Основной метод этого класса accept () ожидает поступления запроса. Когда запрос получен, метод устанавливает соединение с клиентом и возвращает объект класса socket, через который сервер будет обмениваться информацией с клиентом.

Листинг. Упрощенный HTTP-сервер

```
import j ava.net.*;
import java.io.*;
import j ava.uti1.*;
Class Server {
```

```
    public static void main(String[] args) {
        try {
            ServerSocket ss = new ServerSocket(Integer.parseInt(args[0]));
            while (true)
                new HttpConnect(ss.accept());
        }
        catch(ArrayIndexOutOfBoundsException ae) {
            System.err.println("Usage: Server port");
            System.exit(0);
        }
        catch(IOException e) {
            System.out.println(e);
        }
    }
}
Class HttpConnect extends Thread
{
    private Socket sock;
    HttpConnect(Socket s) {
        sock = s;
        setPriority(NORM_PRIORITY - 1);
        start {} ;
    }
    public void run() {
        try  {
            PrintWriter pw = new PrintWriter(new OutputStreamWriter(
                sock.getOutputStream()}, true);
            BufferedReader br = new BufferedReader(new InputStreamReader(
                sock.getInputStream() ) ) ;
            String req = br.readLine();
            System.out.println("Request: " + req);
            StringTokenizer st = new StringTokenizer(req);
            if ((st.countTokens() >= 2) && st.nextToken().equals("POST")) {
                if ((req = st.nextToken()).endsWith("/") II req.equals(""))
                req += "index.html";
                try {
                    File f = new File(req);
                    BufferedReader bfr = new BufferedReader(new FileReader(f));
                    char[] data = new char[(int)f.length()];
                    bfr.read(data);
                    pw.println("HTTP/1.1 200 OK\n");
                    pw.write(data);
```

```
                    pw.flush();
                }
                catch(FileNotFoundException fe) {
                    pw.println("HTTP/1.1 404 Not FoundXn");
                }
                catch(IOException ioe) {
                    System.out.println(ioe);
                }
            }
            else pw.println("HTTP/l.l 400 Bad RequestW);
            sock.close();
        }
        catch(IOException e) {
            System.out.println(e);
        }
    }
}
```

Вначале следует запустить сервер, указав номер порта, например:

```
Java Server 8080
```

Затем надо запустить клиент, указав IP-адрес или доменное имя хоста, номер порта и имя файла:

```
Java Client localhost 8080 Server.Java
```

Сервер отыскивает файл Server.java в своем текущем каталоге и посылает его клиенту. Клиент выводит содержимое этого класса в стандартный вывод и завершает работу. Сервер продолжает работать, ожидая следующего запроса.

Замечание по отладке.

Программы, реализующие стек протоколов TCP/IP, всегда создают так называемую "петлю" с адресом 127.0.0.1 и доменным именем localhost. Это адрес самого компьютера. Он используется для отладки приложений клиент-сервер. Вы можете запускать клиент и сервер на одной машине, пользуясь этим адресом.

21.4. Работа по протоколу UDP

Для посылки дейтаграмм отправитель и получатель создают сокеты дейтаграммного типа. В Java их представляет класс DatagramSocket. В классе три конструктора:

- DatagramSocket() - создаваемый сокет присоединяется к любому свободному порту на локальной машине;
- DatagramSocket(int port) — создаваемый сокет присоединяется к порту port на локальной машине;
- DatagramSocket(int port, InetAddress addr) - создаваемый сокет при соединяется к порту port; аргумент addr - один из адресов локальной машины.

Класс содержит массу методов доступа к параметрам сокета и, кроме того, методы отправки и приема дейтаграмм:

- send(DatagramPacket pack) - отправляет дейтаграмму, упакованную в пакет pack;
- receive(DatagramPacket pack) - дожидается получения дейтаграммы и заносит ее в пакет pack.

При обмене дейтаграммами соединение обычно не устанавливается, дейтаграммы посылаются наудачу, в расчете на то, что получатель ожидает их. Но можно установить соединение методом

```
connect(InetAddress addr, int port)
```

При этом устанавливается только одностороннее соединение с хостом по адресу addr и номером порта port — или на отправку или на прием дейтаграмм. Потом соединение можно разорвать методом

```
disconnect()
```

При посылке дейтаграммы по протоколу JJDP сначала создается сообщение в виде массива байтов, например,

```
String mes = "This is the sending message.";
byte[] data = mes.getBytes();
```

Потом записывается адрес - объект класса inetAddress, например:

```
InetAddress addr = InetAddress.getByName (host);
```

Затем сообщение упаковывается в пакет - объект класса DatagramPacket. При этом указывается массив данных, его длина, адрес и номер порта:

```
DatagramPacket pack = new DatagramPacket(data, data.length, addr, port)
```

Далее создается дейтаграммный сокет

```
DatagramSocket ds = new DatagramSocket()
```

и дейтаграмма отправляется

```
ds.send(pack)
```

После посылки всех дейтаграмм сокет закрывается, не дожидаясь какой-либо реакции со стороны получателя:

```
ds.close ()
```

Прием и распаковка дейтаграмм производится в обратном порядке, вместо метода send() применяется метод receive (DatagramPacket pack).

В листинге показан пример класса Sender, посылающего сообщения, набираемые в командной строке, на localhost, порт номер 1050. Класс Recipient, описанный в листинге 19.6, принимает эти сообщения и выводит их в свой стандартный вывод.

Листинг. Посылка дейтаграмм по протоколу UDP

```
import java.net.*;
import java.io.*;
Class Sender {
    private String host;
    private int port;
    Sender(String host, int port){
    this.host = host;
    this.port = port;
}
private void sendMessage(String mes) {
    try  {
        byte[] data = mes.getBytes();
        InetAddress addr = InetAddress.getByName(host);
        DatagramPacket pack = new DatagramPacket(data, data.length, addr, port);
        DatagramSocket ds = new DatagramSocket();
        ds.send(pack);
        ds.close();
    }
    catch(IOException e) {
        System.err.println(e);
    }
}
public static void main(String[] args) {
    Sender sndr = new Sender("localhost", 1050);
    for (int k = 0; k < args.length; k++) sndr.sendMessage(args[k]);
}
```

Листинг. Прием дейтаграмм по протоколу UDP

```
import j ava.net.*;
```

```
import java.io.*;
Class Recipient {
    public static void main(String[] args) (
        try {
            DatagramSocket ds = new DatagramSocket(1050);
            while (true) {
                DatagramPacket pack =
                    new DatagramPacket(new byte[1024], 1024);
                ds.receive(pack);
                System.out.println(new String(pack.getData()));
            }
        }
        catch(Exception e) {
            System.out.println(e);
        }
    }
}
```

22. Архиватор jar

Для упаковки нескольких файлов в один архивный файл, со сжатием или без сжатия, в технологии Java разработан формат JAR. Имя архивного jar-файла может быть любым, но обычно оно получает расширение jar. Способ упаковки и сжатия основан на методе ZIP. Название JAR (Java ARchive) перекликается с названием известной утилиты TAR (Tape ARchive), разработанной в UNIX.

Отличие jar-файлов от zip-файлов только в том, что в первые автоматически включается каталог META-INF, содержащий несколько файлов с информацией об упакованных в архив файлах.

Архивные файлы очень удобно использовать в апплетах, поскольку весь архив загружается по сети сразу же, одним запросом. Все файлы апплета с байт-кодами, изображениями, звуковые файлы упаковываются в один или несколько архивов. Для их загрузки достаточно в теге <appiet> указать имена архивов в параметре archive, например:

```
<applet code = "MillAnim.Class" archive = "first.jar, second.jar"
width = "100%" height = "100%"></applet>
```

Основной файл MillAnim.Class должен находиться в каком-либо из архивных файлов first.jar или second.jar. Остальные файлы отыскиваются в архивных файлах, а если не найдены там, то на сервере, в том же каталоге, что и HTML-файл. Впрочем, файлы апплета можно упаковать и в zip-архив, со сжатием или без сжатия.

Архивные файлы удобно использовать и в приложениях (applications). Все файлы приложения упаковываются в архив, например, appl.jar. Приложение выполняется прямо из архива, интерпретатор запускается с параметром -jar, например:

```
Java -jar appl.jar
```

Имя основного класса приложения, содержащего метод main (), указывается в файле MANIFEST.MF, речь о котором пойдет чуть ниже.

Архивные файлы удобны и просты для компактного хранения всей необходимой для работы программы информации. С файлами архива можно работать прямо из архива, не распаковывая их, с помощью классов пакета java.util.jar.

Jar-архивы создаются с помощью классов пакета java.util.jar или с помощью утилиты командной строки jar.

23. Связь с базами данных через JDBC

Большинство информации хранится не в файлах, а в базах данных. Приложение должно уметь связываться с базой данных для получения из нее информации или для помещения информации в базу данных. Дело здесь осложняется тем, что СУБД (системы управления базами данных) сильно отличаются друг от друга и совершенно по-разному управляют базами данных. Каждая СУБД предоставляет свой набор функций для доступа к базам данных, и приходится для каждой СУБД писать свое приложение Но что делать при работе по сети, когда неизвестно, какая СУБД управляет базой на сервере?

Выход был найден корпорацией Microsoft, создавшей набор интерфейсов ODBC (Open Database Connectivity) для связи с базами данных, оформленных как прототипы функций языка С. Эти прототипы одинаковы для любой СУБД, они просто описывают набор действий с таблицами базы данных. В приложение, обращающееся к базе данных, записываются вызовы функций ODBC. Для каждой системы управления базами данных разрабатывается так называемый драйвер ODBC, реализующий эти функции для конкретной СУБД. Драйвер просматривает приложение, находит обращения к базе данных, передает их СУБД, получает от нее результаты и подставляет их в приложение. Идея оказалась очень удачной, и использование ODBC для работы с базами данных стало общепринятым.

Фирма SUN подхватила эту идею и разработала набор интерфейсов и классов, названный JDBC, предназначенный для работы с базами данных. Эти интерфейсы и классы составили пакет java.sql, входящий в J2SDK Standard Edition, и его расширение javax.sql, входящее в J2SDK Enterprise Edition.

Кроме классов с методами доступа к базам данных для каждой СУБД необходим драйвер JDBC — промежуточная программа, реализующая методы JDBC. Существуют 4 типа драйверов JDBC.

- Драйвер, реализующий методы JDBC вызовами функций ODBC. Это так называемый мост (bridge) JDBC-ODBC. Непосредственную связь с базой при этом осуществляет драйвер ODBC.
- Драйвер, реализующий методы JDBC вызовами функций API самой СУБД.
- Драйвер, реализующий методы JDBC вызовами функций сетевого протокола, независимого от СУБД. Этот протокол должен быть, затем, реализован средствами СУБД.
- Драйвер, реализующий методы JDBC вызовами функций сетевого протокола СУБД.

Перед обращением к базе данных следует установить нужный драйвер, например, мост JDBC-ODBC:

```
try {
    Class dr = sun.jdbc.odbc.JdbcOdbcDriver.Class;
}
catch(ClassNotFoundException e) {
    System.err.println("JDBC-ODBC bridge not found " + e);
}
```

После того как драйвер установлен, надо связаться с базой данных. Методы связи описаны в интерфейсе connection. Экземпляр класса, реализующего этот интерфейс, можно получить одним из статических методов getConnection() класса DriverManager, например:

```
String url = "jdbc:odbc:mydb";
String login = "habib";
String password = "lnF4vb";
Connection qon = DriverManager.getConnection(url, login, password);
```

Обратите внимание на то, как формируется адрес базы данных url. Он начинается со строки "jdbc:", потом записывается подпротокол (subprotocol), в данном примере используется мост JDBC-ODBC, поэтому записывается "odbc:". Далее указывается адрес (subname) по правилам подпротокола, здесь просто имя локальной базы "mydb". Второй и третий аргументы — это имя и пароль для соединения с базой данных.

Если в вашей вычислительной системе установлен пакет javax.sql, то вместо класса DriverManager лучше использовать интерфейс DataSource.

Связавшись с базой данных, можно посылать запросы. Запрос хранится в объекте, реализующем интерфейс statement. Этот объект создается методом createStatement(), описанным в интерфейсе connection. Например:

```
Statement st = con.createStatement();
```

Затем запрос (query) заносится в этот объект методом execute () и потом выполняется методом getResultSet(). В простых случаях это можно сделать одним методом executeQuery(), например:

```
ResultSet rs = st.executeQuery("SELECT name, code FROM tbl");
```

Здесь из таблицы tbl извлекается содержимое двух столбцов name и code и заносится в объект rs класса, реализующего интерфейс ResultSet.

SQL-операторы INSERT, UPDATE, DELETE, CREATE TABLE и другие в простых случаях выполняются методом executeUpdate().

Остается методом next() перебрать элементы объекта rs - строки полученных столбцов - и извлечь данные многочисленными методами getXxx() интерфейса ResultSet:

```
while (rs.next()) {
    emp[i] = rs.getString("name") ;
    num[i] = rs.getlnt("code");
    i++;
}
```

Методы интерфейса ResultSetMetaData позволяют узнать количество полученных столбцов, их имена и типы, название таблицы, имя ее владельца и прочие сведения о представленных в объекте rs сведениях. Если объект st получен методом

```
    Statement st = con.createStatement(
    ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_OPDATABLE);
```

то можно перейти к предыдущему элементу методом previous(), к первому элементу - методом first(), к последнему - методом last(). Можно также изменять объект rs методами updateXxx() и даже изменять, удалять и добавлять соответствующие строки базы данных. Не все драйверы обеспечивают эти возможности, поэтому, надо проверить реальный тип объекта rs методами rs.getType() и rs.getConcurrency().

Интерфейс Statement расширен интерфейсом PreparedStatement, тоже позволяющим изменять объект ResultSet методами setxxxo.

Интерфейс Preparedstatement, в свою очередь, расширен интерфейсом caiiablestatement, в котором описаны методы выполнения хранимых процедур.

24. Сервлеты

Технология CGI. Ее суть в том, что сетевой клиент, обычно браузер, посылает Web-серверу информацию вместе с указанием программы, которая будет обрабатывать эту информацию. Web-сервер, получив информацию, запускает программу, передает информацию на ее стандартный ввод и ждет окончания обработки. Результаты обработки программа отправляет на свой стандартный вывод, Web-сервер забирает их оттуда и отправляет клиенту. Написать программу можно на любом языке, лишь бы Web-сервер мог взаимодействовать с ней.

В технологии Java такие программы оформляются как сервлеты (servlets). Это название не случайно похоже на название "апплеты". Сервлет на Web-сервере играет такую же роль, что и апплет в браузере, расширяя его возможности.

Чтобы Web-сервер мог выполнять сервлеты, в его состав должна входить JVM и средства связи с сервлетами. Обычно все это поставляется в виде отдельного модуля, встраиваемого в Web-сервер. Существует много таких модулей:. Они называются на жаргоне сервлетными движками (servlet engine).

Основные интерфейсы и классы, описывающие сервлеты, собраны в пакет javax.servlet. Расширения этих интерфейсов и классов, использующие конкретные особенности протокола HTTP, собраны в пакет javax.servlet.http. Еще два пакета— javax.servlet.jsp и javax.servlet.jsp.tagext — предназначены для связи сервлетов со скриптовым языком JSP (JavaServer Pages). Все эти пакеты входят в состав J2SDK Enterprise Edition. Они могут поставляться и отдельно как набор JSDK (Java Servlet Development Kit). Сервлет-ный движок должен реализовать эти интерфейсы.

Основу пакета javax.servlet составляет интерфейс servlet, частично реализованный в абстрактном классе GenericServiet и его абстрактном подклассе HttpServiet. Основу этого интерфейса составляют 3 метода:

- init(Servietconfig config) - задает начальные значения сервлету, играет ту же роль, что и метод init() в апплетах;
- service(ServletRequest req, ServletResponse resp) - выполняет обработку поступившей сервлету информации req и формирует ответ resp;
- destroy() - завершает работу сервлета.

Вся работа сервлета сосредоточена в методе service(). Это единственный метод, не реализованный в классе GenericServiet. Достаточно расширить свой класс от класса. GenericServiet и реализовать в нем метод service(), чтобы получить собственный сервлет. Сервлетный движок, встроенный в Web-сервер, реализует интерфейсы ServletRequest и ServletResponse. Он создает объекты

req и resp, заносит всю поступившую информацию в объект req и передает этот объект сервлету вместе с пустым объектом resp. Сервлет принимает эти объекты как аргументы req и resp метода service о, обрабатывает информацию, заключенную в req, и оформляет ответ, заполняя объект resp. Движок забирает этот ответ и через Web-сервер отправляет его клиенту.

Основная информация, заключенная в объекте req, может быть получена методами read() и readLine() Из байтового потока класса ServletInputStream, непосредственно расширяющего класс inputstream, или из символьного потока класса BufferedReader. Эти потоки открываются, соответственно, методом req.getInputStream() или методом req.getReader (). Дополнительные характеристики запроса можно получить многочисленными методами getXxx() объекта req. Кроме того, класс GenericServlet реализует массу методов getXxx(), позволяющих получить дополнительную информацию о конфигурации клиента.

Интерфейс servietResponse описывает симметричные методы для формирования ответа. Метод getoutputstreamo открывает байтовый поток класса ServletOutputStream, непосредственно расширяющего класс OutputStream. Метод getWriter() открывает символьный поток класса PrintWriter.

Итак, реализуя метод service(), надо получить информацию из входного потока объекта req, обработать ее и результат записать в выходной поток объекта resp.

Очень часто в объекте req содержится запрос к базе данных. В таком случае метод service о обращается через JDBC к базе данных и формирует ответ resp из полученного объекта ResultSet.

Протокол HTTP предлагает несколько методов передачи данных: GET, POST, PUT, DELETE. Для их использования класс GenericServlet расширен классом HttpServlet, находящимся В пакете javax.servlet.http. В этом классе есть методы для реализации каждого метода передачи данных:

- doGet(HttpServletRequest req, HttpServletResponse resp)
- doPost(HttpServletRequest req, HttpServletResponse resp)
- doPut(HttpServletRequest req, HttpServletResponse resp)
- doDelete(HttpServletRequest req, HttpServletResponse resp)

Для работы с конкретным HTTP-методом передачи данных достаточно расширить свой класс от класса HttpServlet и реализовать один из этих методов. Метод service() переопределять не надо, в классе HttpServlet он только определяет, каким HTTP-методом передан запрос клиента, и обращается к соответствующему методу doXxxo. Аргументы перечисленных методов req и resp - это объекты, реализующие Интерфейсы HttpServletRequest и HttpServletResponse, расширяющие интерфейсы ServletRequest и ServietResponse, соответственно.

Интерфейс HttpServletRequest к тому же описывает множество методов getxxx(), позволяющих получить дополнительные свойства запроса req.

Интерфейс HttpServletResponse описывает методы addXxx() и setXxx(), дополняющие ответ resp, и статические константы с кодами ответа Web-сервера.

Применение сервлета позволило "облегчить" клиент - браузер не загружает апплет, а только отправляет запрос и получает ответ. Вся обработка запроса ведется в сервлете на сервере.

В системе J2SDKEE (Java 2 SDK Enterprise Edition) HTML-файл и сервлет образуют один Web-компонент. Они упаковываются в один файл с расширением war (web archive) и помещаются в так называемый Web-контейнер, управляемый Web-сервером, расширенным средствами J2SDKEE.

25. Java на сервере

Тенденция написания сетевых программ — побольше функций возложить на серверную часть программы и поменьше оставить клиентской части, сделав клиент "тонким", а сервер "толстым". Это позволяет, с одной стороны, использовать клиентскую часть программы на самых старых и маломощных компьютерах, а с другой стороны, облегчает модификацию программы — все изменения достаточно сделать только в одном месте, на сервере.

Сервер выполняет все больше функций, как говорят, служб или сервисов (services). Он и отправляет клиенту Web-страницы, и выполняет сервлеты, и связывается с базой данных, и обеспечивает транзакции. Такой многофункциональный сервер называется сервером приложений (application server). Большой популярностью сейчас пользуются серверы приложений WebLogic фирмы BEA Systems, IAS (Inprise Application Server) фирмы Borland, WebSphere фирмы IBM, OAS (Oracle Application Server). Важной характеристикой сервера приложений является способность расширять свои возможности путем включения новых модулей. Это удобно делать с помощью компонентов.

Фирма SUN Microsystems предложила свою систему компонентов EJB (Enterprise JavaBeans), включенную в Java 2 SDK Enterprise Edition. Подобно тому, как графические компоненты JavaBeans реализуют графический интерфейс пользователя, размещаясь в графических контейнерах, компоненты EJB реализуют различные службы на сервере, располагаясь в EJB-контейнерах. Этими контейнерами управляет EJB-сервер, включаемый в состав сервера приложений. В составе J2SDKEE EJB-сервер — это программа j2ee. Серверу приложений достаточно запустить эту программу, чтобы использовать компоненты EJB.

В отличие от JavaBeans у компонентов EJB не может быть графического интерфейса и ввода с клавиатуры. У них может отсутствовать даже консольный вывод. Контейнер EJB занимается не размещением компонентов, а созданием и удалением их объектов, связью с клиентами и другими компонентами, проверкой прав доступа и обеспечением транзакций.

Программы, оформляемые как EJB, могут быть двух типов: EntityBean и sessionBean. Они реализуют соответствующие интерфейсы из пакета javax.ejb. Первый тип EJB-компонентов удобен для создания программ, обращающихся к базам данных и выполняющих сложную обработку полученной информации. Компоненты этого типа могут работать сразу с несколькими клиентами. Второй тип EJB-компонентов более удобен для организации взаимодействия с клиентом. Компоненты этого типа бывают двух видов: сохраняющие свое состояние от запроса к запросу (stateful) и теряющие это состояние (stateless). Методами интерфейсов EntityBean и SessionBean контейнер EJB управляет поведением

экземпляров класса. Если достаточно стандартного управления, то можно сделать пустую реализацию этих методов.

Кроме класса, реализующего интерфейс EntityBean или SessionBean, для создания компонента EJB необходимо создать еще два интерфейса, расширяющие интерфейсы вовноте и EjBObject. Первый интерфейс (home interface) служит для создания объекта EJB своими методами create(), для поиска и связи с этим объектом в процессе работы, и удаления его методом remove о. Второй интерфейс (remote interface) описывает методы компонента EJB. Интересная особенность технологии EJB — клиентская программа не образует объекты компонента EJB. Вместо этого она создает объекты home-и remote-интерфейсов и работает с этими объектами. Реализация home- и remote-интерфейсов, создание объектов компонента EJB и взаимодействие с ними остается на долю контейнера EJB.