МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Озерский технологический институт — филиал НИЯУ МИФИ

Вл. Пономарев

Введение в Java для программистов

Учебно-методическое пособие

Озерск, 2013

УДК 681.3.06
П 56

Пономарев В.В. Введение в Java для программистов. Учебно-методическое пособие. Редакция 1 (02.02.2013). Озерск: ОТИ НИЯУ МИФИ, 2013. — 65 с., ил.

В пособии кратко описывается язык программирования Java.

Пособие предназначено для студентов-программистов, обучающихся по специальности 230105 — «Программное обеспечение вычислительной техники и автоматизированных систем».

Рецензенты:
1.
2.

УТВЕРЖДЕНО
Редакционно-издательским
Советом ОТИ НИЯУ МИФИ

Содержание

Язык Java. Историческая справка

Язык Java является прямым потомком языков C и C++. Синтаксис Java почти полностью совпадает с синтаксисом C, а объектно-ориентированные свойства Java унаследованы от C++. В отличие от C++, объектно-ориентированные свойства интегрированы в Java, так, что любая программа Java является объектно-ориентированной.

Первоначальная версия языка (1991-1992 г.г.) называлась Oak (дуб), создателями являются Джеймс Гослинг (James Gosling), Патрик Ноутон (Patrick Naughton), Крис Варт (Chris Warth) и Эд Франк (Ed Frank) из компании Sun Microsystems; в 1995 г. язык был переименован в Java.

Изначально Java создавался как платформенно-независимый язык для внедрения программного обеспечения в электронные устройства различного назначения (типа микроволновых печей или дистанционных пультов управления), однако в дальнейшем стала очевидной его полезность при создании программного обеспечения для Интернет, который примерно в это же время стал получать широкое распространение. Этому способствовали такие свойства языка, как безопасность и мобильность.

С помощью Java создаются два основных типа программ — приложения (обычные программы) и апплеты. Апплет — это небольшое приложение, предназначенное для передачи по Интернет и исполнения web-браузером, поддерживающим Java. Язык обеспечивает защиту, ограничивая апплет средой его выполнения, и не позволяя ему получить доступ к другим частям клиентского компьютера.

Мобильность Java основана на использовании так байт-кода, в который компилируются все программы Java. Байт-код — это оптимизированный набор команд, исполняемый специальной системой, называемой виртуальной Java-машиной (JVM, Java Virtual Machine). Любая система, в которой установлена JVM, способна выполнять программы Java.

Начальные сведения

Для программирования на языке Java в системе должно быть установлено дополнительное программное обеспечение.

JDK (*Java Development Kit*) — пакет утилит, необходимых для разработки программных модулей на Java.

JRE (*Java Runtime Environment*) — пакет утилит, необходимых для выполнения программных модулей на Java.

Для написания Java-программ дополнительно может быть использована среда разработки, такая, как *Eclipse* или *NetBeans*.

Создание Java-программы вручную

Без использования среды разработки текст программы создается при помощи обычного текстового редактора, а компилируется и выполняется при помощи командной строки.

Пример простейшей программы:

Пример кода 1 — Простейшая программа на Java

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

Важное отличие от программы на C++ заключаются в том, что Java-программа может содержать только описания классов (а также специальные строки, являющиеся аналогами директивы `#include` и предназначенные для указания на используемые библиотеки).

Еще одно отличие заключается в том, что если класс описывает точку входа в программу, то тогда в классе должен быть описан статический метод `main`, как показано в примере кода 1. Параметром этого метода является обязательный массив строк аргументов `args`.

Заметим также, что действия программы, не описываемые синтаксисом языка, выполняются только методами других классов. Так, вывод на терминал строки "Hello!" выполняется методом "System.out.println".

Текст должен сохраняться с именем, в точности соответствующим названию класса, и расширением ".java". Так, для примера кода 1 название файла должно быть "hello.java" с точностью до регистра.

Компиляция модуля производится при помощи команды:

```
javac hello.java
```

Здесь javac — имя файла компилятора (javac.exe). При необходимости вместо этого имени указывается полный путь, например:

```
c:\progra~1\java\jdk1.6.0_01\bin\javac hello.java
```

Кроме этого, название компилируемого модуля указывается полностью, включая расширение.

Результатом компиляции файла "hello.java" будет файл "hello.class".

Для выполнения этого файла используется команда:

```
java hello
```

Обратим внимание, что для выполнения модуля используется другой файл, а именно, java (java.exe), а расширение ".class" указывать нельзя.

Создание Java-апплета

Второй пример программирования на Java показывает создание простейшего апплета:

Пример кода 2 — Простейший апплет

```
import java.applet.*;
import java.awt.*;
public class helloa extends Applet {
  public void paint(Graphics g) {
    g.drawString("Hello!", 20, 30);
  }
}
```

Заметим, что кроме описания класса, здесь используются две строки, включающие библиотеки applet и awt, необходимые для создания и работы апплета. Класс апплета helloa является расширением (то есть наследником) базового класса Applet, определенного в библиотеке applet. Для вывода строки "Hello!" используется графический контекст g типа Graphics, который определен в библиотеке awt.

Переопределенный виртуальный метод paint класса Applet рисует в области апплета на странице html строку "Hello!" в точке с координатами (20, 30). Подробно особенности создания апплетов будут рассмотрены нами позднее в соответствующем разделе.

Приведенный пример кода следует сохранить в файл "helloa.java".

После компиляции класса апплета будет получен файл "helloa.class":

```
javac helloa.java
```

Использовать апплет можно только на странице html. Поэтому для тестирования апплета нужно создать следующую тестовую страницу (в виде файла "helloa.html"):

Пример кода 3 — Тестовая html-страница для тестирования апплета

```
<HTML><BODY>
<APPLET CODE="helloa" WIDTH=100 HEIGHT=100></APPLET>
</BODY></HTML>
```

Из примера кода 3 видно, что апплет размещается на странице html при помощи тега `<applet>`, в котором, в частности, указывается класс апплета, а также его размер в пикселях.

При попытке открыть данную страницу прямым образом (например, двойным щелчком на файл), скорее всего, появится сообщение системы безопасности Windows, предупреждающее о потенциальной опасности страницы, так как она содержит выполняемый код.

При выполнении апплета при помощи web-сервера, например, такого, как apache, сообщение службы безопасности появляться не должно.

Типы данных

В языке Java определено 8 простых типов (то есть типов, не являющимися объектами). Эти типы примерно соответствуют аналогичным простым типам языка C++, но в отличие от C++, в языке Java размер типов строго гарантирован. При помощи простых типов и классов могут быть сконструированы другие (составные) типы любой сложности.

Простые типы Java приведены в таблице 1.

Таблица 1 — Простые типы Java

Тип	Описание	Размер	Диапазон
byte	целый	8 бит	–128…127
short	целый	16 бит	–32768…32767
int	целый	32 бита	$-2^{31}\ldots2^{31}-1$
long	целый	64 бита	$-2^{63}\ldots2^{63}-1$
float	вещественный	32 бита	$\pm(3{,}4^{38}\ldots3{,}4^{38})$
double	вещественный	64 бита	$\pm(1{,}7^{308}\ldots1{,}7^{308})$
char	символьный	16 бит	Unicode
boolean	логический		false и true

Примечания: 1) тип short хранится в памяти «задом наперед», то есть старший бит первым; 2) все числовые типы знаковые.

Заметим, что тип char в Java не совпадает с аналогичным типом C++. Для обеспечения необходимой мобильности Java-программ символьные значения представлены символами Unicode. Чтобы присвоить значение символьной переменной, можно использовать два варианта:

```
char c = '\u0041'; // код знака A (LATIN-1)
char d = 'Я';
```

Java — это язык со строгой типизацией. Автоматические (неявные) преобразования (conversion) типов в нем выполняются только для совместимых типов и только в сторону увеличения размера типа.

Так, в следующем фрагменте кода содержится ошибка в строке 3:

```
byte b;
int a = 0;
b = a;
```

Нельзя присвоить переменной "b" значение переменной "a", которая имеет больший размер. При необходимости нужно выполнить явное приведение (cast) типа, например:

```
byte b;
int a = 1;
b = (byte)a;
```

Массивы

Массив объявляется одним из следующих способов:

Пример кода 4 — Объявления массивов

```
public class arrays {
  public static void main(String[] args) {
    int a[];
    a = new int[5];
    int b[] = new int[5];
    int c[] = { 0, 1, 2, 3, 4 };
    // альтернативный синтаксис
    int[] d;
    int[] e = new int[5];
    int[] f = { 0, 1, 2, 3, 4 };
    System.out.println(f[4]);
  }
}
```

Как и в C++, массивы индексируются от нуля. В отличие от C++, массивы в Java более безопасны в том смысле, что при попытке выйти за границу индекса массива генерируется исключение.

Память под массив выделяется при помощи операции `new` (то есть массивы всегда расположены в динамической памяти, но не являются динамическими). Если значения элементов массива задаются непосредственно в строке объявления, операция new вызывается автоматически.

Многомерные массивы — это массивы массивов. При этом можно создать непрямоугольный массив, например, треугольный:

Пример кода 5 — Треугольный массив

```
public class mdarray {
  public static void main(String[] args) {
    int a[][] = new int[3][];
    a[0] = new int[1];
    a[1] = new int[2];
    a[2] = new int[3];
    int i, j, k = 0;
    for (i = 0; i < 3; i++) {
      for (j = 0; j < i + 1; j++) {
        a[i][j] = k++;
        System.out.print(a[i][j] + " ");
      }
      System.out.println();
    }
  }
}
```

Заметим, что для многомерных массивов выделять память нужно не только для массива в целом, но и для всех его отдельных элементов.

Строки в Java не являются массивами символов.

Размер массива возвращает свойство `length`, которое применяется к переменной. Следующий пример кода поясняет сказанное:

Пример кода 6 — Использование свойства length

```
public class array_length {
  public static void main(String[] args) {
    int[] a = { 0, 1, 2, 3, 4 };
    for (int i = 0; i < a.length; i++) {
      System.out.println(a[i]);
    }
  }
}
```

Перечисления и константы

Перечисления объявляются примерно так же, как в языке C, но есть и отличия. Во-первых, перечисления объявляются только внутри класса, но не внутри метода. Во-вторых, перечисления могут иметь тип доступа `public`, `protected` или `private` и видны, таким образом, внутри или вне класса, в котором они определены. В третьих, в связи с вышесказанным, значения переменных типа перечисления должны задаваться с указанием квалификатора: имени класса и (или) имени перечисления:

Пример кода 7 — Использование перечисления

```
public class enums {
  enum xx { A, B, C };
  public static void main(String[] args) {
    xx x = xx.A;
    int b;
    if (x == xx.A) b = 1; else b = 0;
    System.out.println(b);
    System.out.println(x);
  }
}
```

В отличие от языка C, элементы перечисления определяют значения, в точности совпадающие со своими идентификаторами, но не определяют никаких числовых значений, поэтому нельзя задать такое значение ни для какого элемента. Последний оператор в примере кода 7 выводит значение элемента перечисления `xx.A`, равное "A".

В Java нет препроцессора, соответственно, нет макроподстановок и возможности задать константы с его помощью. Вместо этого можно задать переменные, значения которых изменить нельзя:

Пример кода 8 — Использование констант

```
public class constants {
  final static int MAX = 3; // константная переменная
  public static void main(String[] args) {
    int[] a = new int[MAX];
    System.out.println(a.length);
    // недопустимо: MAX = 4;
  }
}
```

Операторы и операции

Java наследует все операторы и операции языка C. Тем не менее, существуют некоторые различия.

Логические выражения могут иметь только два значения, а именно, false и true, причем эти значения не эквивалентны никаким числовым (то есть арифметическим) значениям. Поэтому операторы, использующие логические выражения, не будут правильными, если вместо логических выражений используются арифметические.

Например, в условном выражении оператора `if` (равно как и в условных выражениях операторов `while` и `do`) нельзя использовать просто переменную, которая не является логической, как в языке C. Поэтому следующий фрагмент кода на Java является недопустимым:

```
int a = 1;
if (a) ;
```

Правильно будет так:

```
int a = 1;
if (a != 0) ;
```

Второе важное отличие касается логических операций. Как известно, в языке C операция `&` выполняет побитовое AND, а операция `|` — побитовое OR. В языке Java эти операции являются либо побитовыми, либо логическими, в зависимости от типов операндов:

Пример кода 9 — Побитовые и логические операции

```
public class bitwise_and_logical {
  public static void main(String[] args) {
    int a = 10, b = 3, c = 0;
    c = a & b;
    System.out.println(c);   // выводит 2
    boolean d = true, e = true, f = false;
    f = d & e;
    System.out.println(f);   // выводит true
  }
}
```

При этом в операторе вида

```
c = a & b;
```

оба операнда и принимающая результат переменная должны иметь одинаковый (или совместимый) тип.

Для выполнения логических операций AND и OR в Java используются также операции `&&` и `||` соответственно. Они выполняются точно так, как в языке C.

Разница между операциями `&` и `&&` заключается в том, что первая из них обязательно вычисляет оба своих операнда перед выполнением, а вторая, как и в языке C, сначала вычисляет левый операнд, и только ес

ли левый операнд недостаточен для вычисления результата операции в целом, вычисляется правый операнд. Такое поведение логических операций называется short-circuit.

Следующий пример кода поясняет сказанное:

Пример кода 10 — Быстрое вычисление результата логической операции

```
public class short_circuit {
  public static void main(String[] args) {
    int a = 1, b = 0, c = 0;
    if (b != 0 && a / b == 1) ;
      else System.out.println("Done");
    if (b != 0 & a / b == 1) ;
      else System.out.println("Done");
  }
}
```

Здесь в первом операторе `if` используется логический оператор `&&`, который, вычислив левый операнд (`b != 0`), равный false, далее не вычисляет правый операнд, поскольку результат этой операции равен false независимо от значения правого операнда.

Во втором операторе `if` используется оператор `&`, который требует вычисления как левого, так и правого операндов, поэтому при выполнении этого оператора генерируется исключение, так как при вычислении правого операнда (`a / b == 1`) происходит деление на ноль.

В Java добавлена новая операция `>>>`, отличающаяся от операции `>>` тем, что она не расширяет знаковый (старший) бит. Операция сдвига вправо фактически выполняет целочисленное деление на два, а результат этих операций будет различаться для отрицательных чисел:

Пример кода 11 — Сдвиги вправо (целочисленное деление на два)

```
public class shifts {
  public static void main(String[] args) {
    int a = -64;
    int b;
    b = a >> 1;
    System.out.println(b);
    b = a >>> 1;
    System.out.println(b);
  }
}
```

Программа выводит числа –32 и 217483616. Экспериментально установлено, что в некоторых версиях языка Java применить операции сдвига влево или вправо можно только к типам `int` и `long`.

В Java нет оператора `goto`. Вместо этого операторы `break` и `continue` имеют вариант написания, когда они могут выполнять выход за пределы поименованного меткой блока `{}`. Следующий пример показывает применение подобного оператора `break`:

Пример кода 12 — Безусловный переход при помощи break

```
public class jumps {
  public static void main(String[] args) {
    one: for (int i = 0; i < 3; i++) {
      System.out.println("loop one");
      for (int j = 0; j < 3; j++) {
        System.out.println("loop two");
        break one;
      }
    }
    System.out.println("main-end");
  }
}
```

При выполнении оператора `break` в этом примере происходит безусловный переход за границу блока, помеченного меткой `one`, то есть осуществляется выход из внешнего (а не из внутреннего цикла). Оператор `continue` аналогичного вида выполняет переход к следующей итерации указанного (как правило, внешнего) цикла.

В Java нет указателей («сущего проклятия» C-программистов) и, соответственно, нет операции `->` («стрелка»).

Классы

Классы — главная часть объектно-ориентированного языка. Но если в языке C++ использование классов не является обязательным, то в языке Java описание класса является единственным способом создания программного модуля.

Более того, единственным местом, в котором можно написать код на языке Java, является *какой-нибудь метод какого-нибудь класса*. В пределах Java-программы не существует никаких самостоятельных, «висящих в воздухе» функций (не принадлежащих никакому классу). Если же такая функция нужна программисту, то следует описать *статический* метод подходящего для цели класса. Единственными функциями, которые могут быть использованы в пределах Java-программы, являются только методы каких-либо объектов.

Классы Java в целом похожи на классы языка C++. Описание класса Java имеет примерно следующий вид:

```
[доступ] class имя [extends имя][implements имя [, имя ...]] {
  // перечисления
  [доступ] enum имя { ид [, ид ...] };
  . . .
  // переменные
  [доступ] тип имя [ = значение][, имя [ = значение] ...];
  . . .
  // методы
  [доступ] тип имя(аргументы) { /* код тела метода */ }
  . . .
}
```

Имеется ряд отличий, направленных на обеспечение необходимой гибкости, надежности и мобильности Java-программ.

1) Описание класса не имеет секций **private**, **public** или **protected**. Вместо этого доступ указывается для каждого элемента класса, а также для класса или интерфейса в целом.

2) Методы класса определяются целиком в описании класса. Таким образом, описание класса всегда является полным.

3) Используется одиночное наследование классов при помощи ключевого слова **extends** (расширяет) и множественное наследование интерфейсов при помощи ключевого слова **implements** (реализует).

4) Создать объект класса можно только при помощи операции **new**.

Иначе говоря, объекты всегда располагаются в динамической памяти и не могут быть статическими. Переменная, обозначающая представителя класса, является ссылкой на объект, а не самим объектом.

5) Класс может быть определен как абстрактный при помощи модификатора **abstract**. Как и в языке C++, абстрактный класс не может порождать объекты.

6) Класс может быть определен таким образом, что его нельзя будет использовать в качестве базового. Для этой цели используется модификатор **final**, который нельзя использовать с модификатором **abstract**.

7) Класс Java может иметь несколько (в том числе перегруженных) конструкторов, но деструкторов для классов Java не предусмотрено.

Вместо этого в классах Java используется метод **finalize**, который вызывается так называемым «сборщиком мусора» перед окончательным уничтожением объекта. Метод **finalize** может не вызываться при выходе объекта из области видимости, как деструктор C++.

8) Для вызова конструктора базового класса используется ключевое слово **super**:

Пример кода 13 — Вызов конструктора при помощи super

```
class A {
  A() {
    System.out.println("Base constructior.");
  }
}
class B extends A {
  B() {
    super();
    System.out.println("Derive constructior.");
  }
}
public class superuse {
  public static void main(String[] args) {
    B b = new B();
  }
}
```

9) Класс может содержать один или несколько блоков `static {}`, которые выполняются один раз во время загрузки класса. Они предназначены для инициализации статических переменных. Можно сказать, что эти блоки являются статическими конструкторами.

Пример кода 14 — Блок static

```
class withstatic {
  static int a = 0;
  static { a = 1; }
}
public class staticuse {
  public static void main(String[] args) {
    System.out.println(withstatic.a);
    withstatic.a = 2;
    System.out.println(withstatic.a);
  }
}
```

При первом обращении к элементу `a` класса `withstatic` выполняется блок `static`. Программа выводит 1, затем 2.

Методы классов

Любая функция, определенная в классе, становится его методом. Как было сказано, методы являются единственными контейнерами кода.

Методы Java имеют следующие особенности.

1) Все методы являются виртуальными по умолчанию:

Пример кода 15 — Полиморфизм

```
class figure {
  void draw() {
    System.out.println("figure");
  }
}
class square extends figure {
  void draw() {
    System.out.println("square");
  }
}
class circle extends figure {
  void draw() {
    System.out.println("circle");
  }
}
public class polymorph {
  public static void main(String args[]) {
    figure f = new square();
    f.draw();
    f = new circle();
    f.draw();
  }
}
```

2) Методы с модификатором `final` нельзя переопределить.

3) Метод с модификатором `abstract` должен быть определен в каком-нибудь производном классе.

4) Методы базовых классов можно вызвать при помощи ключевого слова `super`:

Пример кода 16 — Вызов метода при помощи super

```
class A {
  void draw() {
    System.out.println("Base draw.");
  }
}
class B extends A {
  void draw() {
    super.draw();
    System.out.println("Derived draw.");
  }
}
public class superusem {
  public static void main(String[] args) {
    B b = new B();
    b.draw();
  }
}
```

5) Перегрузка операций запрещена.

6) Параметры методов передаются только по значению.

Класс Math

Этот класс определяет статические методы для вычисления математических функций: трансцендентных, экспоненциальных, округления, вычисления максимума и минимума, генерирования псевдослучайных значений. Список методов приведен в приложении А.

В качестве примера приведем пример вычисления псевдослучайного целого числа в диапазоне [a…b]:

Пример кода 17 — Вычисление псевдослучайного целого

```
public class randomm {
  public static void main(String args[]) {
    int a = 2, b = 7, m;
    m = a + (int)Math.floor((Math.random() * (double)(b - a + 1)));
    System.out.println(m);
  }
}
```

Пояснения. Метод `Math.random` возвращает псевдослучайное целое число типа `double` в диапазоне [0…1). Умножая его на b−a+1, получаем вещественное число в диапазоне [0…b−a+1). Метод `Math.floor` отсекает дробную часть числа, после чего получаем целое значение типа `double` в диапазоне [0…b−a]. Приводя это значение к целому типу и прибавляя "a", получаем целое число в диапазоне [a…b].

Класс String

Этот класс определяет методы для работы со строками. Объект типа `String` после инициализации является константным.

Создание пустой строки:

```
String s = new String();
```

Создание строки из литерала:

```
String s = "abc";
String s = new String("abc");
```

Создание строки из объекта `String`:

```
String s = "abc";
String b = new String(s);
```

Конкатенация строк:

```
String s = "abc";
String b = "123" + s + new String("xyz");
```

Конкатенация строк с другими типами:

```
int a = 44;
String s = "123" + a + 2.2;
```

Длина строки:

```
String s = "abc";
int a = s.length;           // a = 3
int a = "abcd".length;      // a = 4
```

Извлечение символа строки (позиция символа от нуля):

```
String s = "abc";
char c = s.charAt(1);       // c = 'b';
char d = "abc".charAt(1);   // d = 'b';
```

Проверка равенства строк:

```
String s = "22";
String b = "22";
int k = 22;
if (s.equals(b)) System.out.println("equal");
if (s.equals("22")) System.out.println("equal");
if (s.equals(Integer.toString(22))) System.out.println("equal");
if (s.equals(Integer.toString(k))) System.out.println("equal");
String c = "abc";
String d = "ABC";
if (!c.equals(d)) System.out.println("not equal");
if (c.equalsIgnoreCase(d)) System.out.println("equal");
```

Проверка равенства строк и равенства строковых объектов:

```
String s = "abc";
String b = new String(s);
String c = s;
String d = "a" + "bc";
if (s.equals(b)) System.out.println("equal");
if (!(s == b)) System.out.println("not equal");
if (s == c) System.out.println("equal");
if (s == d) System.out.println("equal"); // equal !
```

Инициализация из массива символов и байт:

```
char c[] = { 'a', 'b', 'c' };
String s = new String(c);
String a = new String(c, 1, 2);
byte b[] = { 97, 98, 99 };
String c = new String(b);
```

Извлечение символов в массив символов и байт:

```
char c[] = new char[3];
"abcdef".getChars(3, 6, c, 0);
System.out.println(c);
byte[] b = "abcdef".getBytes();
for (int i = 0; i < b.length; i++) {
    System.out.println(b[i]);
}
```

Сравнение областей двух строк:

```
String a = "abcdef";
String b = "defabc";
if (a.regionMatches(3, b, 0, 3)) System.out.println("equal");
```

Параметры — индекс в первой строке, вторая строка, индекс во второй строке, число сравниваемых символов.

Сравнение с началом или концом строки:

```
System.out.println("abcdef".startsWith("abc")); // true
System.out.println("abcdef".endsWith("def"));   // true
```

Сравнение строк:

```
System.out.println("bc".compareTo("bc"));   // 0
System.out.println("bc".compareTo("bcd"));  // -1
System.out.println("bc".compareTo("bd"));   // -1
System.out.println("bc".compareTo("abc"));  // 1
```

Поиск подстроки:

```
System.out.println("aa00aa00aa".indexOf("bb"));     // -1
System.out.println("aa00aa00aa".indexOf("00"));     // 2
System.out.println("aa00aa00aa".lastIndexOf("00")); // 6
```

Извлечение подстроки:

```
System.out.println("0123456789".substring(5));      // 56789
System.out.println("0123456789".substring(5, 8));   // 567
```

Замена символа:

```
System.out.println("a1a1a1".replace('a', '0'));     // 010101
```

Отрезание пробелов:

```
System.out.println("  abc  ".trim());   // abc
```

Преобразование в строку:

```
byte b = 97;
System.out.println(String.valueOf(b));  // 97
```

Изменение регистра:

```
System.out.println("ABC".toLowerCase());    // abc
System.out.println("abc".toUpperCase());    // ABC
```

Класс Arrays

Класс Arrays предназначен для выполнения операций с массивами. Для доступа к классу нужно импортировать "`java.util.*`".

Заполнение массива одинаковыми значениями:

```
int a[] = new int[5];
Arrays.fill(a, 1);
arrayDisplay(a);        // arrayDisplay не принадлежит Arrays
```

Метод `fill` (как и другие методы) перегружен для разных типов.

Заполнение части массива:

```
int a[] = new int[5];
Arrays.fill(a, 2, 4, 1);    // 0 0 1 1 0
arraydisplay(a);
```

Сортировка массива (или его части):

```
int a[] = { 2, 1, 4, 3, 0 };
Arrays.sort(a);
arraydisplay(a);
```

Бинарный поиск в сортированном массиве:

```
int a[] = { 0, 1, 2, 3, 4 };
System.out.println(Arrays.binarySearch(a, 5));  // -6
System.out.println(Arrays.binarySearch(a, 3));  // 3
```

Сравнение массивов:

```
int a[] = { 0, 1, 2, 3, 4 };
int b[] = { 0, 1, 2, 3, 4 };
System.out.println(Arrays.equals(a, b));  // true
int c[] = { 0, 1, 2, 3, 4, 5 };
System.out.println(Arrays.equals(a, c));  // false
```

Оболочки простых типов

Оболочки простых типов предназначены для формирования объектов из простых типов и выполнения разных операций с ними.

Есть следующие числовые оболочки:

`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`.

Создание объекта типа `Integer` (при создании объекта из строки может выбрасываться исключение `NumberFormatException`):

```
Integer a = new Integer(33);
Integer b = new Integer("33");
try {
    Integer c = new Integer("33.0");
    Integer d = new Integer("33u");
} catch(NumberFormatException e) {}
```

Преобразования в другие числовые типы:

```
byte b   = a.byteValue();
short c  = a.shortValue();
long d   = a.longValue();
float e  = a.floatValue();
```

```
double f = a.doubleValue();
```

Преобразования из строки (также выбрасывается исключение):

```
int a = Integer.parseInt("33");
a = Integer.parseInt("21", 16); // 16-ричная система счисления
a = Integer.decode("33");
a = Integer.valueOf("33");
```

Преобразования в строку:

```
System.out.println(Integer.toBinaryString(33)); // 100001
System.out.println(Integer.toOctalString(33));  // 41
System.out.println(Integer.toHexString(33));    // 21
System.out.println(Integer.toString(33, 8));    // 41
System.out.println(Integer.toString(33, 16));   // 21
System.out.println(Integer.toString(33, 2));    // 100001
System.out.println(Integer.toString(33));       // 33
```

Дополнительно оболочки имеют методы **equals** и **compareTo**.

Оболочка Character

Определяет константы:
MAX_RADIX — наибольшее основание
MIN_RADIX — наименьшее основание
MAX_VALUE — наибольшее значение символа
MIN_VALUE — наименьшее значение символа

isDefined(char) возвращает **true**, если символ — символ Unicode.

isDigit(char) возвращает **true**, если символ — цифра.

isISOControl(char) возвращает **true**, если символ — управляющий.

isLetter(char) возвращает **true**, если символ — буква.

isLetterOrDigit(char) возвращает **true**, если символ буква или цифра.

isSpaceChar(char) возвращает **true**, если символ — пробел.

isLowerCase(char) возвращает **true**, если символ в нижнем регистре.

isUpperCase(char) возвращает **true**, если символ в верхнем регистре.

toLowerCase(char) переводит символ в нижний регистр.

toUpperCase(char) переводит символ в верхний регистр.

forDigit(int n, int radix) возвращает цифровой символ, связанный со значением **n** в системе счисления **radix**.

digit(char d, int radix) возвращает целое значение, связанное с символом **d** в системе счисления **radix**.

Дополнительно есть методы **equals**, **compareTo** и **hashCode**.

Оболочка Boolean

Определяет константы TRUE и FALSE, обозначающие истинные и ложные объекты. Методы:

```
booleanValue()
equals(Object)
toString()
valueOf(String)
```

Коллекции

Коллекция — это множество объектов. Для работы с коллекциями используется пакет `java.util`.

Коллекции работают на основе интерфейсов.

Интерфейс Collection

Описывает множество объектов, объекты могут дублироваться.

`boolean add(Object o)` добавляет элемент в коллекцию.

`boolean addAll(Collection c)` добавляет коллекцию в коллекцию.

`void Clear()` очищает коллекцию.

`boolean contains(Object o)` возвращает `true`, если `o` в коллекции.

`boolean containsAll(Collection c)` возвращает `true`, если коллекция содержит все элементы `c`.

`boolean equals(Object o)` возвращает true, если объекты равны.

`int hashCode()` возвращает хеш-код коллекции.

`boolean isEmpty()` возвращает `true`, если коллекция пуста.

`Iterator iterator()` возвращает итератор коллекции.

`boolean remove(Object o)` удаляет элемент коллекции.

`boolean removeAll(Collection c)` удаляет множество элементов.

`boolean retainAll(Collection c)` удаляет элементы, за исключением `c`.

`int size()` возвращает число элементов.

`Object[] toArray()` возвращает массив копий элементов коллекции.

Интерфейс List

Описывает список (нумерованных) элементов.

`void add(int index, Object o)` вставляет `o` в позицию `index`.

`boolean addAll(int index, Collection c)` добавляет коллекцию.

`Object get(int index)` возвращает объект в позиции `index`.

`int indexOf(Object o)` возвращает позицию объекта `o` или –1.

`int lastIndexOf(Object o)` возвращает позицию объекта `o` или –1.

`ListIterator listIterator()` возвращает итератор, установленный на начало списка.

`ListIterator listIterator(int index)` возвращает итератор, установленный в позицию `index`.

`Object remove(int index)` удаляет элемент списка.

`Object set(int index, Object o)` записывает объект в позицию `index`.

`List subList(int start, int end)` возвращает часть списка.

Интерфейс Set

Расширяет поведение интерфейса `Collection`, запрещая дублирование элементов. Не добавляет никаких новых методов.

Интерфейс SortedSet

Описывает множество, сортированное в возрастающем порядке.

`Comparator comparator()` возвращает компаратор множества или `null`, если для множества используется естественное упорядочение.

`Object first()` возвращает первый элемент множества.

`Object last()` возвращает последний элемент множества.

`SortedSet headSet(Object end)` возвращает начальную часть множества, в которой элементы меньше, чем `end`.

`SortedSet tailSet(Object start)` возвращает конечную часть множества, в которой элементы больше или равны `start`.

`SortedSet subSet(Object start, Object end)` возвращает подмножество, в котором элементы находятся между `start` и `end`–1.

Классы коллекций

`AbstractCollection` реализует большую часть интерфейса `Collection`.

`AbstractList` реализует большую часть интерфейса `List`.

`AbstractSequentialList` расширяет `AbstractList` для реализации коллекций, которые используют последовательный доступ к элементам.

`LinkedList` реализует связный список, расширяя предыдущий класс.

`ArrayList` реализует динамический массив, расширяя `LinkedList`.

`AbstractSet` расширяет `AbstractCollection` и реализует большую часть интерфейса `Set`.

`HashSet` расширяет `AbstractSet` для реализации хэш-таблиц.

`TreeSet` реализует набор, хранящийся в виде дерева, расширяя класс `AbstractSet`.

Пакеты

Пакет является механизмом именования и управления видимостью.

Классы пакета могут быть видимы вне его или не видимы.

Элементы классов пакета могут быть доступны только элементам классов пакета или классам вне пакета.

Вообще говоря, любой класс находится внутри пакета.

Если программист не указывает пакет, то пакетом является пакет по умолчанию (*default package*), не имеющий имени. Необходимость пакетов возникает в случае большого числа классов.

Пакет объявляется при помощи ключевого слова `package`:

```
package mypackage;
```

Здесь `mypackage` — имя пакета.

Для хранения пакетов Java использует каталоги файловой системы.

Например, все class-файлы пакета `mypackage` должны располагаться в каталоге `mypackage` (при этом регистр имеет значение).

Инструкция **package** может быть включена одновременно в несколько файлов Java. Все классы, определенные в этих файлах, включаются в указанный пакет.

Для создания иерархии пакетов используется нотация

```
package name1.name2.name3;
```

Здесь пакет **name3** расположен внутри пакета **name2**, который, в свою очередь, расположен внутри пакета **name1**. То же самое можно сказать и о каталогах. Например, пакет name3 должен быть расположен внутри каталога **name1\name2\name3**.

Переменная окружения CLASSPATH указывает на корневой каталог иерархии каталогов, в которых хранятся пакеты. Значение этой переменной определяет, будут ли доступны для выполнения классы пакета.

В качестве примера рассмотрим создание простого пакета.

Пусть класс располагается в каталоге "**c:\java-03\test**".

Название пакета **test**, название класса **packtest**.

Код класса примерно следующий:

```
package test;
public class packteSt {
  public static void main(String[] args) {
    System.out.println("package hello.");
  }
}
```

Скомпилировав класс, получим файл **packtest.class**, который располагается также в каталоге "**c:\java-03\test**".

Для выполнения класса нужно:

- добавить (установить) в CLASSPATH путь "**c:\java-03**";
- перейти в каталог "**c:\java-03**";
- выполнить команду "**java test.packtest**".

Видимость классов пакета следующая.

Класс пакета, объявленный как **public**, виден внутри и вне пакета.

Класс пакета, объявленный без **public**, виден во всех подклассах пакета, но не виден вне пакета.

Классы пакета могут быть включены в другие классы и пакеты при помощи инструкции **import**:

```
import пакет[.пакет].(класс | *);
```

Интерфейсы

Определение интерфейса подобно определению класса. Различие заключается в том, что объявления методов не имеют тела (вместо тела записывается точка с запятой).

Интерфейсы могут объявлять переменные-константы.

Пример описания интерфейса:

```
interface IFigure {
  void draw();
}
```

Интерфейсы удобно использовать для описания констант. Следующий интерфейс описывает несколько констант:

```
public interface WeekofdayConstants {
  int Monday = 0;
  int Tuesday = 0;
  int Wednesday = 0;
  int Thursday = 0;
  int Friday = 0;
  int Satturday = 0;
  int Sunday = 0;
}
```

Следующий класс может использовать эти константы:

```
class Weekdays implements WeekofdayConstants {
  //
}
```

Если класс наследует интерфейс, он должен определять все его методы, либо объявлять себя абстрактным.

Обработка исключений

Общий вид обработки исключений такой же, как в C++:

```
try {
} catch (Exception e) {
} catch (Exception e) {
} [finally {
}]
```

Блок `finally` не является обязательным. Если этот блок есть, то он выполняется после обработки исключения в любом случае (независимо от того, исключение обработано или нет).

В отличие от C++, исключения — это объекты, наследующие базовый тип `Throwable`. Этот тип имеет два производных класса: `Exception` и `Error`. У класса `Exception` есть подкласс `RuntimeException`.

Исключения типа `Error` свидетельствуют о катастрофической ошибке и относятся к среде времени исполнения.

Исключения типа `RuntimeException` описывают такие ошибки времени исполнения, как деление на ноль или выход за границы массива.

Исключения типа `Exception` используются программистом для описания собственных подклассов исключений.

Оператор `throw` выбрасывает исключения, производные от `Throwable`.

Пример выбрасывания исключения:

```
throw new NullPointerException("какое-то сообщение");
```

Пакет AWT

Пакет AWT (Abstract Windowing Toolkit) был создан для поддержки апплетов. Классы пакета позволяют создавать окна и элементы управления, обрабатывать события, а также рисовать и управлять шрифтами.

Обработка событий

В Java используется механизм делегирования событий (*delegation event model*). Источник генерирует событие и посылает его блоку прослушивания события (*listener*). Блок прослушивания обрабатывает его и возвращает управление. Чтобы получать уведомления о событиях, блок прослушивания должен быть зарегистрирован в источнике событий.

Пусть, например, мы хотим получать уведомления о событиях клавиатуры. Тогда мы должны создать блок прослушивания, который описывает реакцию на событие и зарегистрировать его. Следующий пример кода показывает обработку события при помощи апплета.

Пример кода 18 — Обработка события клавиатуры в апплете

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class keylisten extends Applet implements KeyListener {
  String msg;
  public void init() {
    addKeyListener(this);
    requestFocus();
  }
  public void start() { }
  public void stop() { }
  public void paint(Graphics g) {
    g.drawString(msg, 10, 20);
  }
  public void keyPressed(KeyEvent ke) { }
  public void keyReleased(KeyEvent ke) { }
  public void keyTyped(KeyEvent ke) {
    msg = "";
    msg += ke.getKeyChar();
    repaint();
  }
}
```

Здесь класс `keyevents` наследует интерфейс блока прослушивания `KeyListener` и определяет три метода:

`keyPressed` (клавиша нажата),

`keyReleased` (клавиша отпущена) и

`keyTyped` (введена алфавитно-цифровая клавиша).

В методе `init` апплет регистрирует себя в качестве блока прослушивания при помощи метода `addKeyListener`. Метод `requestFocus` требует, чтобы апплет получил фокус.

Классы событий

Класс `EventObject` (находящийся в `java.util`) является суперклассом всех событий. Пакет `awt` содержит подкласс `AWTEvent`, являющийся суперклассом событий AWT. Пакет `java.awt.event` определяет несколько типов событий, которые генерируются элементами интерфейса.

При генерировании события обязательным параметром является источник (*source*) типа `Object`. Получить источник можно при помощи метода `getSource`. Другие параметры зависят от типа события.

В таблице 2 приведены основные классы событий. Интерфейсы прослушивания имеют названия, в которых `Event` заменено `Listener`.

Таблица 2 — Классы событий

Класс	События
`ActionEvent`	нажатие кнопки, выбор в меню или списке
`AdjustmentEvent`	манипуляция с линейкой прокрутки
`ComponentEvent`	перемещение, изменение размера, видимости
`ContainerEvent`	добавление или удаление элемента контейнера
`FocusEvent`	приобретение или потеря фокуса компонентом
`InputEvent`	абстрактный суперкласс событий ввода-вывода
`ItemEvent`	выбор флажка или элемента списка
`KeyEvent`	события клавиатуры
`MouseEvent`	события мыши
`TextEvent`	изменение текста в поле ввода
`WindowEvent`	события окна

Классы-адаптеры событий

Для упрощения обработки событий в Java есть классы-адаптеры.

В следующем примере кода используется адаптер `keyListener`.

Пример кода 19 — Обработка события клавиатуры с помощью адаптера

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class keyadapter extends Applet {
  String msg;
  public void init() {
    addKeyListener(new myKeyListener(this));
    requestFocus();
  }
  public void start() {}
  public void stop() {}
  public void paint(Graphics g) {
    g.drawString(msg, 10, 20);
  }
}
```

```
class myKeyListener extends KeyAdapter {
  keyadapter ka;
  public myKeyListener(keyadapter ka) {
    this.ka = ka;
  }
  public void keyTyped(KeyEvent ke) {
    ka.msg = "";
    ka.msg += ke.getKeyChar();
    ka.repaint();
  }
}
```

Здесь класс `myKeyListener`, производный от `KeyAdapter`, описывает обработку только одного интересующего нас события `keyTyped`.

Анонимные классы обработчиков

Анонимный класс обработчика создается в блоке вида

```
new TypeAdapter() { тело анонимного класса }
```

Следующий пример показывает использование анонимного класса.

Пример кода 20 — Анонимный класс обработчика события клавиатуры

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class keyanonim extends Applet {
  String msg;
  public void init() {
    addKeyListener(new KeyAdapter() {
      // тело анонимного класса
      public void keyTyped(KeyEvent ke) {
        msg = "";
        msg += ke.getKeyChar();
        repaint();
      }
    } );
    requestFocus();
  }
  public void start() {}
  public void stop() {}
  public void paint(Graphics g) {
    g.drawString(msg, 10, 20);
  }
}
```

Окна

Часть иерархии оконных классов приведена на рисунке 1.

`Component` — абстрактный класс, описывающий наиболее общие свойства визуального компонента. Он управляет событиями, положением, размером, перерисовкой окна, отвечает за шрифт и цвет.

Класс `Container` позволяет одни окна вкладывать в другие. Контейнер управляет вложенными окнами с помощью менеджера компоновки (*layout manager*), такого, как FlowLayout.

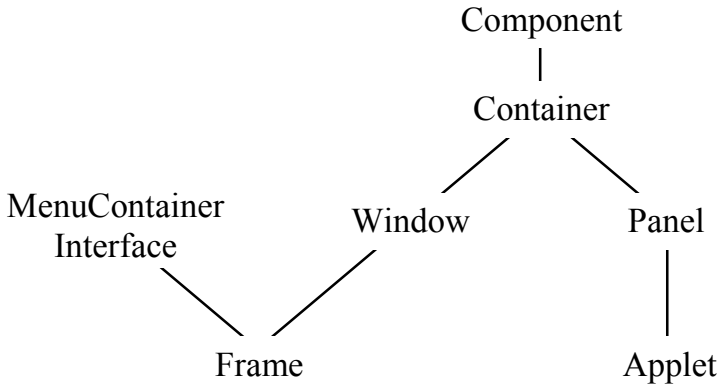

Рисунок 1 — Иерархия оконных классов

Класс `Panel` — реализация класса `Container`. Этот класс является суперклассом класса `Applet`, на основе которого создаются апплеты.

Окна создаются на основе класса `Frame`. Окно этого класса имеет заголовок и рамку. Следующий пример показывает создание оконного приложения с обработчиком событий клавиатуры.

Пример кода 21 — Приложение с простым окном

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class winapp extends Frame {
  public static void main(String[] args) {
    winapp wa = new winapp();
    wa.setBackground(new Color(212, 208, 200)); // цвет
    wa.setLocation(new Point(100, 100));        // положение
    wa.setSize(new Dimension(300, 300));        // размер
    wa.setTitle("winapp");                      // заголовок
    wa.setVisible(true);
  }
  public winapp() {
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    } );
    addKeyListener(new KeyAdapter() {
      public void keyPressed(KeyEvent ke) {
        if (ke.getKeyCode() == KeyEvent.VK_ESCAPE) {
          System.exit(0);
        }
      }
    } );
  }
}
```

В методе `main` устанавливаются основные параметры окна. Конструктор `winapp` создает два анонимных класса обработчиков. Первый закрывает окно при помощи кнопки «Закрыть», а второй — при помощи клавиши «Escape».

Элементы управления

Элемент управления нужно создать, вставить (добавить) в контейнер, прикрепить к нему блок прослушивания.

В следующем примере кода в апплете создается кнопка «OK».

Пример кода 22 — Окно с кнопкой

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class buttona extends Applet implements ActionListener {
  String msg = "";
  Button b;
  public void init() {
    setLayout(null);
    b = new Button();
    add(b);
    b.setLabel("OK");
    b.setBounds(20, 60, 80, 27);
    b.addActionListener(this);
  }
  public void actionPerformed(ActionEvent ae) {
    if (ae.getSource() == b) {
      msg = ae.getActionCommand();
    } else {
      msg = "Not defined.";
    }
    repaint();
  }
  public void start() {}
  public void stop() {}
  public void paint(Graphics g) {
    g.drawString(msg, 10, 50);
  }
}
```

Метод `setLayout` с параметром `null` указывает, что программист сам будет размещать элементы управления. Метод `add` добавляет кнопку в окно (контейнер) апплета. Метода `setLabel` задает надпись элемента управления (любого, у которого она есть). При создании кнопки надпись можно указать в качестве параметра:

```
b = new Button("OK");
```

Метод `setBounds` задает положение (20, 60) и размеры (80, 27) элемента управления. Метод `getActionCommand`, используемый в блоке прослушивания, возвращает надпись на кнопке. Определить нажатую кнопку можно не только по надписи на ней, но и по ссылке, которую возвращает, например, метод `add`. В примере применены оба метода.

Метки (надписи) создаются конструктором `Label`. Параметр, если есть — надпись. Второй параметр, если есть — `LABEL.LEFT`, `LABEL.RIGHT` или `LABEL.CENTER` (выравнивание). Метод `setLabel` меняет надпись.

Метки не генерируют событий.

Конструкторы флажков (и переключателей) следующие:

```
Checkbox()
Checkbox(String s)
Checkbox(String s, boolean on)
Checkbox(String s, boolean on, CheckboxGroup gr)
Checkbox(String s, CheckboxGroup gr, boolean on)
```

Параметры: `s` — надпись, `on` — начальное состояние, `gr` — группа.

Если параметр `gr` равен `null`, создается флажок.

Чтобы создать группу переключателей, сначала нужно создать группу `gr` конструктором `CheckboxGroup`. Делается это так:

```
CheckboxGroup gr;
Checkbox a, b, c;
gr = new CheckboxGroup();
a = new Checkbox("Option 1", true, gr);
b = new Checkbox("Option 2", false, gr);
c = new Checkbox("Option 3", false, gr);
```

Интерфейс `ItemListener` блока прослушивания флажка содержит один метод `itemStateChanged(ItemEvent ie)`. Чтобы получить состояние флажка, используется метод `getState`, возвращающий `boolean`:

```
public void itemStateChanged(ItemEvent ie) {
  if (a.getState())
    // флажок включен
  else
    // флажок выключен
}
```

Состояние флажка можно также получить методом `getStateChange`, применяя его к событию:

```
public void itemStateChanged(ItemEvent ie) {
  if (ie.getStateChange() == 1)
    // флажок включен
  else
    // флажок выключен
}
```

Элемент управления `Choice` соответствует выпадающему (раскрывающемуся) списку. Конструктор `Choice` не имеет параметров. Элементы добавляются методом `add` или `addItem` с параметром типа `String`. В списке элементы отображаются в порядке добавления.

К элементу управления `Choice` применяются методы:

`int getItemCount()` — возвращает количество элементов

`String getItem(int index)` — возвращает название по номеру

`String getSelectedItem()` — возвращает выбранный элемент

`int getSelectedIndex()` — возвращает номер выбранного элемента

`void select(int index)` — выбирает элемент по номеру

`void select(String item)` — выбирает элемент по названию

Элементы нумеруются с нуля. Блок прослушивания — `ItemListener`.

Обычные списки представлены классом **List**, для которого определено три конструктора:

```
List()
List(int numRows)
List(int numRows, boolean multipleSelect)
```

Параметр **numRows** определяет высоту списка в строчках. Параметр **multipleSelect**, равный **true**, задает список с множественным выбором.

Элементы списка добавляются методом **add**:

```
void add(String name)
void add(String name, int index)
```

Второй метод добавляет элемент в указанную позицию. Если позиция равна –1, элемент добавляется в конец.

К списку **List** применяются методы:

int getItemCount() — возвращает количество элементов

String getItem(int index) — возвращает название по номеру

String getSelectedItem() — возвращает выбранный элемент

String[] getSelectedItems() — возвращает массив элементов

int getSelectedIndex() — возвращает номер выбранного элемента

int[] getSelectedIndexes() — возвращает массив номеров

void select(int index) — выбирает элемент по номеру

Блоки прослушивания — **ActionListener** и **ItemListener**. Первый используется для отслеживания двойного щелчка на элемент, а второй — для отслеживания одиночного щелчка (выбор или отмена выбора).

Следующий пример кода показывает обработку событий списка.

Пример кода 23 — События списка

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class lista extends Applet
                implements ActionListener, ItemListener {
  String msg = "";
  List a;
  public void init() {
    a = new List(3);
    add(a);
    a.add("Item 1");
    a.addActionListener(this);
    a.addItemListener(this);
  }
  public void actionPerformed(ActionEvent ae) {
    msg = "Command " + a.getSelectedItem();
    repaint();
  }
  public void itemStateChanged(ItemEvent ie) {
    msg = "Selected " + a.getSelectedItem();
    repaint();
  }
```

```
  public void start() {}
  public void stop() {}
  public void paint(Graphics g) {
    g.drawString(msg, 10, 100);
  }
}
```

Однострочное поле ввода описывает класс **TextField**. Конструкторы:

```
TextField()
TextField(int numChars)
TextField(String s)
TextField(String s, int numChars)
```

Параметр **numChars** задает ширину поля в символах, строковый параметр **s** задает начальный текст. К полю **TextField** применяются методы:

String getText() — возвращает текст

void setText(String s) — устанавливает текст

String getSelectedText() — возвращает выделенный текст

void select(int startIndex, int endIndex) — выделяет текст

boolean isEditable() — истина, если поле редактируемое

void setEditable(boolean canEdit) — управляет редактируемостью

void setEchoChar(char c) — устанавливает символ-заменитель

boolean echoCharIsSet() — признак установки символа-заменителя

char getEchoChar() — возвращает символ-заменитель

Три последних метода используются при создании полей для ввода секретной информации (пароля).

Блок прослушивания — **ActionListener**. Поле вырабатывает событие при нажатии клавиши «Enter».

Многострочное поле ввода описывает класс **TextArea**. Конструкторы:

```
TextArea()
TextArea(String s)
TextArea(int numLines, int numChars)
TextArea(String s, int numLines, int numChars)
TextArea(String s, int numLines, int numChars, int sBars)
```

Параметр **numLines** задает высоту поля в строчках, **sBars** — линейки прокрутки (**SCROLLBARS_NONE**, **SCROLLBARS_BOTH**, **SCROLLBARS_HORIZONTAL_ONLY**, **SCROLLBARS_VERTICAL_ONLY**).

Методы класса **TextArea** включают в себя методы **TextField**, за исключением методов, управляющих символами-заменителями. Кроме этого, есть следующие методы:

void append(String s) — добавляет строку в конец

void insert(String s, int index) — вставляет строку в позицию

void replaceRange(String s, int startIndex, int endIndex) — заменяет текст от позиции **startIndex** до позиции **endIndex-1**.

Блок прослушивания — **FocusListener** (получение и потеря фокуса).

Меню

Меню реализуется следующими классами:

`MenuBar` — соответствует строке меню в верхней области окна;

`Menu` — соответствует выпадающему меню (списку);

`MenuItem` — соответствует пункту меню.

`CheckboxMenuItem` — соответствует флажку в меню.

Использование этих классов поясняет следующий пример.

Пример кода 24 — Меню окна

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class MenuApp extends Frame {
  String msg = "start";
  CheckboxMenuItem chm;
  public static void main(String[] args) {
    MenuApp ma = new MenuApp("Menu");
    ma.setSize(new Dimension(300, 300));
    ma.setVisible(true);
  }
  public MenuApp(String title){
    super(title);
    MenuBar mb = new MenuBar();
    setMenuBar(mb);
    Menu file = new Menu("File");
    MenuItem item;
    file.add(chm = new CheckboxMenuItem("Check"));
    file.add(item = new MenuItem("Open"));
    mb.add(file);
    menuHandler handler = new menuHandler(this);
    item.addActionListener(handler);
    chm.addItemListener(handler);
  }
  public void paint(Graphics g) {
    g.drawString(msg, 10, 200);
    if (chm.getState()) {
      msg = "Checked";
    } else {
      msg = "Unchecked";
    }
    g.drawString(msg, 10, 240);
  }
}
class menuHandler implements ActionListener, ItemListener {
  MenuApp ma;
  public menuHandler(MenuApp ma) { this.ma = ma; }
  public void actionPerformed(ActionEvent ae) {
    String a = ae.getActionCommand();
    if (a.equals("Open")) { ma.msg = "Open"; }
    ma.repaint();
  }
  public void itemStateChanged(ItemEvent ie) { ma.repaint(); }
}
```

Менеджеры компоновки

Обычно размещением элементов управления в окне управляет менеджер компоновки. Менеджер выбирается при помощи метода

```
void setLayout(LayoutManager layoutObj)
```

Если параметр метода равен `null`, элементы позиционируются вручную (так, как было показано выше).

Если между менеджером и контейнером необходимо оставить пустое пространство (границу), в классе окна (апплета) следует определить метод `getInsets` следующим образом:

```
public Insets getInsets() {
  return new Insets(10, 10, 10, 10);
}
```

Здесь параметры конструктора `Insets` задают ширину границы сверху, слева, снизу и справа соответственно.

Менеджер FlowLayout

Это менеджер поточной компоновки, являющийся также менеджером по умолчанию. Конструкторы:

```
FlowLayout()
FlowLayout(int how)
FlowLayout(int how, int horz, int vert)
```

Первый конструктор выравнивает элементы по центру с расстоянием 5 пикселей. Второй конструктор позволяет выбрать выравнивание:

```
FlowLayout.LEFT
FlowLayout.CENTER
FlowLayout.RIGHT
```

Третий конструктор задает также расстояние между элементами.

Менеджер BorderLayout

Задает пять зон размещения (рисунок 2).

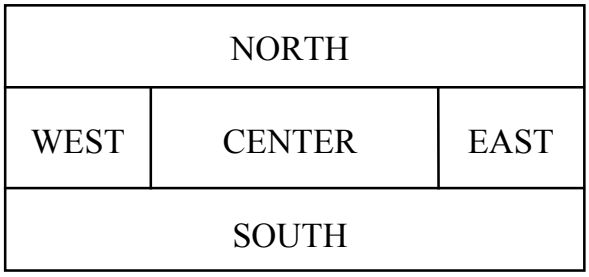

Рисунок 2 — Зоны менеджера BorderLayout

Конструкторы:

```
BorderLayout()
BorderLayout(int horz, int vert)
```

При добавлении элемента в методе `add` этого менеджера указывается зона размещения. Например, следующий код

```
add(new Button("West"), BorderLayout.WEST)
```

размещает кнопку в зоне `WEST`.

Менеджер GridLayout

Это менеджер сеточного или табличного размещения.

Конструкторы:

```
GridLayout()
GridLayout(int numRows, int numCols)
GridLayout(int numRows, int numCols, int horz, int vert)
```

Первый конструктор создает размещение с одним столбцом. Второй конструктор задает размещение в `numCols` столбцах и `numRows` строках.

Третий конструктор задает также расстояние между элементами.

Менеджер CardLayout

Это менеджер карточного размещения (имеется ввиду колода игральных карт). Конструкторы:

```
CardLayout()
CardLayout(int horz, int vert)
```

При использовании менеджера нужно создать панель (класс `Panel`), которая содержит колоду, и панель для каждой карты в колоде.

Панели карт добавляются к панели колоды с присвоением имени каждой карте:

```
void add(Component panelObj, Object name)
```

Активизация карты (то есть перемещение ее наверх) производится одним из методов:

```
void first(Container deck)
void last(Container deck)
void next(Container deck)
void previous(Container deck)
void show(Container deck, String cardName)
```

Здесь `deck` — ссылка на панель колоды. Методы показывают соответственно первую, последнюю, следующую, предыдущую или произвольную карту (по ее имени).

Диалоговые окна

Диалоговые окна (диалоги) могут быть модальными и немодальными. Модальные диалоги приостанавливают выполнение программы до их закрытия. Немодальные диалоги выполняются параллельно с выполнением основных окон. Диалоги расширяют класс `Dialog`.

Конструкторы класса `Dialog`:

```
Dialog(Frame parentWindow, boolean mode)
Dialog(Frame parentWindow, String title, boolean mode)
```

Здесь `parentWindow` — это ссылка на родительское окно (которое может быть как первичным, так и вторичным, то есть диалогом). Параметр `mode`, равный `true`, указывает на создание модального диалога. Следующий пример кода показывает создание модального диалога.

Пример кода 25 — Диалог

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
public class DialogApp extends Frame implements ActionListener {
  public static void main(String[] args) {
    DialogApp da = new DialogApp();
    da.setTitle("DialogApp");
    da.setSize(new Dimension(100, 100));
    da.setVisible(true);
  }
  public DialogApp(){
    Object b = add(new Button("Dialog"));
    ((Button)b).addActionListener(this);
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    } );
  }
  public void actionPerformed(ActionEvent ae) {
    OkDialog od = new OkDialog(this);
    od.setVisible(true);
  }
}
class OkDialog extends Dialog implements ActionListener {
  public OkDialog(Frame parent) {
    super(parent, "OK Dialog", true);
    setSize(new Dimension(80, 80));
    Object b = add(new Button("OK"));
    ((Button)b).addActionListener(this);
  }
  public void actionPerformed(ActionEvent ae) {
    dispose(); // освобождает память, занимаемую диалогом
  }
}
```

Для файлового диалога используется конструктор `FileDialog`:

```
FileDialog(Frame parent, String title)
FileDialog(Frame parent, String title, int how)
FileDialog()
```

Здесь параметр `how` может принимать значения `FileDialog.LOAD` или `FileDialog.SAVE`. Третий конструктор создает диалог для чтения файла.

Получить путь к файлу и его имя можно при помощи методов:

```
String getDirectory()
String getFile()
```

Апплеты

Апплеты являются подклассами `Applet`. Они должны импортировать `java.applet` и `java.awt`. Апплет выполняется браузером. В отличие от приложений, выполнение апплета не начинается с метода `main`. При выполнении апплета вызываются следующие его методы:

```
init
start
paint
```

При завершении апплета вызываются его методы:

```
stop
destroy
```

Скелетная схема апплета, таким образом, имеет следующий вид:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="applet-name" width="w" heigth="h"></applet>
*/
public class AppletBase extends Applet {
  public void init() { }
  public void start() { }
  public void stop() { }
  public void destoy() { }
  public void paint(Graphics g) { }
}
```

Метод `init` соответствует конструктору, метод `destroy` — деструктору. Метод `start` запускает выполнение апплета, метод `stop` — останавливает его. Метод `paint` рисует изображение апплета. Принудительно метод `paint` вызывается методом `repaint`, имеющий варианты:

```
repaint()
repaint(long maxDelay)
repaint(int left, int top, int width, int height)
repaint(long maxDelay, int left, int top, int width, int height)
```

Обычно изображение апплета рисуется методом `paint` с помощью переменных. Однако рисовать в апплете можно в любом методе после того, как получен графический контекст при помощи `getGraphics`.

Тег <applet>

```
<APPLET
  WIDTH="пиксели" HEIGHT="пиксели"
  CODE="файл класса апплета"
  [CODEBASE="базовый url файла класса апплета"]
  [NAME="имя апплета"]
  [ALT="альтернативный текст"]
  [ALIGN="LEFT|RIGHT|TOP|BOTTOM|MIDDLE|BASELINE"]
  [VSPACE="пиксели"][HSPACE="пиксели"]
> текст html, отображаемый при отсутствии java
  [<PARAM NAME="имя1" VALUE="значение1"> VALUETYPE="DATA|REF|OBJECT"]
  [<PARAM NAME="имя2" VALUE="значение2"> VALUETYPE="DATA|REF|OBJECT"]
</APPLET>
```

Параметр `CODEBASE` задает путь к файлу класса апплета, если он располагается не в текущем каталоге. Параметр `NAME` задает имя апплета для его поиска при помощи метода `getApplet`, определенного в интерфейсе `AppletContext`. Параметр `ALT` задает текст, который выводится в случае, если браузер не может отобразить апплет. Параметр `ALIGN` задает выравнивание апплета относительно окружения. Параметры `VSPACE` и `HSPACE` задают интервалы вокруг области апплета.

Передача параметров в апплет

Апплеты могут получать параметры из содержащей их html-страницы. Следующий пример показывает передачу параметра.

Пример кода 26 — Параметры апплета

```
import java.applet.*;
import java.awt.*;
/*
<APPLET CODE="paramapplet" WIDTH=300 HEIGHT=100>
<PARAM NAME="text" VALUE="Hello!">
</APPLET>
*/
public class paramapplet extends Applet {
  String s;
  public void start() {
    s = getParameter("text");
    if (s == null) s = "Not found.";
  }
  public void paint(Graphics g) {
    g.drawString(s, 20, 40);
  }
}
```

Здесь в теге `<applet>` определен параметр с именем «text» и значением «Hello!». При страте апплета параметр считывается при помощи метода `getParamater`. Если при этом значение не равно `null`, то параметр принят и может быть использован.

Загрузка документа в апплет

Апплеты могут загружать (на свое место в html-странице) документы html. Прежде всего нужно получить путь к документу. Он складывается обычно из пути к странице апплета или к файлу апплета и названия загружаемого документа.

Метод `getDocumentBase` возвращает путь к странице апплета, а метод `getCodeBase` — путь к файлу апплета.

Документ загружается методом `getDocument`, определенным в интерфейсе `AppletContext`. Первый параметр метода — это абсолютный путь к документу (URL). Второй параметр (если есть) указывает место расположения документа ("`_self`", "`_blank`", "`_top`" или "`_parent`").

Следующий пример кода демонстрирует загрузку документа.

Пример кода 27 — Загрузка документа в апплет

```
import java.applet.*;
import java.awt.*;
import java.net.*;
/*
<APPLET CODE="docapplet" WIDTH=100 HEIGHT=100></APPLET>
*/
public class docapplet extends Applet {
  public void start() {
    AppletContext ac = getAppletContext();
    URL url = getCodeBase();
    try {
      ac.showDocument(new URL(url + "test.html"));
    } catch(MalformedURLException e) {
      showStatus("URL not found.");
    }
  }
}
```

Для загрузки апплета используется html-страница "applet.html":

```
<HTML><HEAD><STYLE>BODY{background-color:black}</STYLE></HEAD><BODY>
<APPLET CODE="docapplet" WIDTH=100 HEIGHT=100></APPLET>
</BODY></HTML>
```

Загружаемый документ — html-страница "test.html":

```
<HTML><BODY>
<SPAN style="color:yellow;background-color:black">Some text</SPAN>
</BODY></HTML>
```

Следующий пример демонстрирует загрузку и отображение рисунка из файла, расположенного в месте файла апплета.

Пример кода 28 — Загрузка изображения в апплет

```
import java.applet.*;
import java.awt.*;
import java.net.*;
/*
<APPLET CODE="imageapplet" WIDTH=100 HEIGHT=100></APPLET>
*/
public class imageapplet extends Applet {
  Image img;
  public void start() {
    URL url = getCodeBase();
    try {
      img = getImage(new URL(url + "test.gif"));
    } catch(MalformedURLException e) {
      img = null;
    }
  }
  public void paint(Graphics g) {
    if (img != null) g.drawImage(img, 0, 0, this);
  }
}
```

Загрузить изображение можно также с помощью другой модификации метода `getImage`:

```
img = getImage(getCodeBase(), "test.gif");
```

При этом исключение "`MalformedURLException`" не выбрасывается, поэтому данный код должен быть размещен вне блока `try` или с другим блоком `catch`. Вместо указания имени рисунка можно использовать чтение параметра апплета, например:

```
img = getImage(getCodeBase(), getParameter("image"));
```

Потоки

Java обеспечивает встроенную поддержку многопоточного программирования. Потоки позволяют нескольким частям программы выполняться одновременно, конкурируя друг с другом. Таким образом, многопоточность — это специализированная форма многозадачности.

Многопоточная система Java построена на классе `Thread` и связанном с ним интерфейсе `Runnable`. Важнейшие методы класса `Thread`:

`isAlive` — возвращает признак активности потока;
`join` — ожидает завершения потока;
`resume` — возобновляет выполнение потока;
`run` — точка входа в поток;
`sleep` — выключает поток на некоторое время;
`start` — запускает поток на выполнение;
`suspend` — приостанавливает выполнение потока.

В каждой программе есть главный, первичный поток, который создается автоматически. Другие потоки создает программист для выполнения различных задач программы. В следующем примере показано, как можно управлять главным потоком.

Пример кода 29 — Управление главным потоком

```
public class MainThread {
  public static void main(String[] args) {
    Thread t = Thread.currentThread();
    System.out.println("Main thread: " + t);
    t.setName("my thread");
    System.out.println("New name: " + t);
    try {
      for (int n = 5; n > 0; n--) {
        System.out.println(n);
        Thread.sleep(1000);
      }
    } catch(InterruptedException e) {
      System.out.println("Main thread exception.");
    }
  }
}
```

Здесь `t` — ссылка на текущий (главный) поток. С помощью метода `setName` потоку задается новое имя. В цикле с помощью метода `sleep` поток 5 раз выключается на одну секунду.

Заметим, что использование методов класса `Thread` может вызывать исключение `InterruptedException`, поэтому используется блок `try`.

Вывод значения переменной `t` имеет примерно следующий вид:

```
[main,5,main]
```

Здесь первое слово `main` обозначает имя потока, второе — имя группы потоков. 5 — текущий приоритет потока.

Для создания нового потока обычно используют интерфейс `Runnable`.

В следующем примере создается вторичный поток.

Пример кода 30 — Создание вторичного потока

```
class NewThread implements Runnable {
  Thread t;
  Boolean b = false;
  NewThread() {
    t = new Thread(this, "second");
    t.start();
  }
  public void run() {
    while ( true ) {
      if ( b ) break;
      System.out.println(t);
      try {
        Thread.sleep(250);
      } catch(InterruptedException e) {  }
    }
  }
}
public class SecondThread {
  public static void main(String[] args) {
    NewThread s = new NewThread();
    try {
      Thread.sleep(1500);
      s.b = true;
      Thread.sleep(500);
    } catch(InterruptedException e) {  }
  }
}
```

Вторичный поток описывает класс `NewThread`. Конструктор класса создает новый поток, параметры задают класс и имя потока. Затем поток запускается методом `start`, который фактически вызывает метод `run`.

Метод `run` описывает действия потока. В примере поток просто ждет завершения, периодически проверяя значение флага `b`.

Главный поток (метод `main`) сначала дает возможность вторичному потоку выполняться некоторое время, затем устанавливает признак завершения вторичного потока и ждет его завершения.

Одной из проблем при работе с потоками является их завершение.

При завершении главного потока завершается работа всей программы. Если к этому моменту не все вторичные потоки завершат свое исполнение, то они будут завершены принудительно, при этом возможно зависание исполнительной системы Java. Для синхронизации завершения потоков используются методы **isAlive** и **join**.

Метод **isAlive** возвращает **true**, если поток, на котором метод вызывается, находится в стадии выполнения. Метод **join** ждет, когда поток, на котором он вызывается, завершит свое выполнение. В следующем примере показана модификация метода **main** из предыдущего примера, использующая метод **isAlive**.

Пример кода 31 — Ожидание завершения вторичного потока

```
public class SecondThreadIsAlive {
  public static void main(String[] args) {
    NewThread s = new NewThread();
    try {
      Thread.sleep(1500);
    } catch(InterruptedException e) {  }
    s.b = true;
    while ( s.t.isAlive() ) ;
  }
}
```

Следующий пример показывает использование метода **join**, который является более предпочтительным.

Пример кода 32 — Ожидание соединения потоков

```
public class SecondThread {
  public static void main(String[] args) {
    NewThread s = new NewThread();
    try {
      Thread.sleep(1500);
      s.b = true;
      s.t.join();
    } catch(InterruptedException e) {  }
    System.out.println(s.t.isAlive());
  }
}
```

Следует обратить внимание на то, что метод **isAlive** не выбрасывает исключения, а методы **sleep** и **join** могут это сделать.

Приоритеты потоков в Java задаются константами:

```
Thread.MIN_PRIORITY = 1
Thread.NORM_PRIORITY = 5
Thread.MAX_PRIORITY = 10
```

Метод **setPriority** задает новый приоритет потока, метод **getPriority** возвращает текущий приоритет.

Одной из важных задач многопоточного программирования, как известно, является синхронизация потоков, то есть обеспечение нахождения в критической секции только одного потока.

В Java синхронизация обеспечивается с помощью монитора. Монитор — это объект, который используется для обеспечения взаимоисключающей блокировки. Когда поток получает блокировку, говорят, что он вошел в монитор. При этом другие потоки, пытающиеся войти в блокированный монитор, будут приостановлены до его освобождения.

Каждый объект Java имеет собственный неявный связанный с ним монитор. Чтобы войти в монитор объекта, вызывают метод с ключевым словом **synchronized**. Следующий пример демонстрирует использование такого метода.

Пример кода 33 — Синхронизированный метод

```
class NewThread implements Runnable {
  SynchronizedThread s;
  Thread t;
  Boolean b = true;
  NewThread(SynchronizedThread s, String n) {
    this.s = s;
    t = new Thread(this, n);
    t.start();
  }
  public void run() {
    while ( b ) ;
  }
  synchronized void access() {
    System.out.println(t.getName() + "=" + s.a);
    try {
      Thread.sleep(500);
    } catch(InterruptedException e) {  }
    s.a++;
    b = false;
  }
}
public class SynchronizedThread {
  int a = 0;
  public static void main(String[] args) {
    SynchronizedThread s = new SynchronizedThread();
    NewThread s1 = new NewThread(s, "one");
    NewThread s2 = new NewThread(s, "two");
    s1.access();
    s2.access();
    try {
      s1.t.join();
      s2.t.join();
    } catch(InterruptedException e) {  }
    System.out.println("main=" + s.a);
  }
}
```

Здесь метод **access** класса потока **NewThread** является синхронизированным. При его вызове поток входит в монитор, предотвращая одновременный доступ двух потоков к разделяемой переменной "**a**".

Другой способ синхронизации заключается в использовании оператора **synchronized**. Следующий пример показывает этот прием.

Пример кода 34 — Оператор synchronized

```
class NewThread implements Runnable {
  SynchronizedThread s;
  Thread t;
  Boolean b = true;
  NewThread(SynchronizedThread s, String n) {
    this.s = s;
    t = new Thread(this, n);
    t.start();
  }
  public void run() {
    System.out.println(t.getName() + "=" + s.a);
    try {
      Thread.sleep(500);
    } catch(InterruptedException e) {  }
    synchronized(s) {
      s.a++;
    }
  }
}
```

Здесь синхронизируемый объект — переменная "`s`", ссылка на объект программы. Оператор `synchronized` запрещает одновременный доступ к этому объекту одновременно двум (или более) потокам.

Не всегда указанные приемы позволяют достичь необходимого результата. Рассмотрим классическую задачу о писателях и читателях. Некоторые потоки-писатели записывают сообщения. С другой стороны, некоторые потоки-читатели, пытаются эти сообщения прочитать.

В простейшем случае достаточно одного писателя и одного читателя. Для решения этой задачу, можно создать следующий набор классов.

Класс `AC` обеспечивает синхронизированный доступ к сообщению:

```
class AC {
  PC p;
  AC(PC p) {
    this.p = p;
  }
  synchronized int read() {
    System.out.println("Read=" + p.a);
    return p.a;
  }
  synchronized void write(int a) {
    p.a = a;
    System.out.println("Write=" + a);
  }
}
```

В качестве сообщения здесь выступает переменная "`a`" основного класса программы `PC` (который будет описан далее). Методы класса читают и записывают эту переменную, обеспечивая синхронизацию.

Классы писателя `Producer` и читателя `Consumer` почти одинаковы. Оба класса создают потоки и используют объект класса `AC` для непрерывной записи и чтения сообщений. Количество сообщений ограничено пятью:

```
class Producer implements Runnable {
  AC a;
  Producer(AC a) {
    this.a = a;
    new Thread(this, "producer").start();
  }
  public void run() {
    int n = 1;
    while ( true ) {
      a.write(n++);
      if ( n > 5 ) break;
    }
  }
}
class Consumer implements Runnable {
  AC a;
  Consumer(AC a) {
    this.a = a;
    new Thread(this, "consumer").start();
  }
  public void run() {
    while ( true ) {
      if ( a.read() == 5 ) break;
    }
  }
}
```

Класс основной программы создает представителя классов PC и AC, а также писателя и читателя, которые сразу начинают свою работу:

```
public class PC {
  int a = 0;
  public static void main(String[] args) {
    PC p = new PC();
    AC a = new AC(p);
    new Producer(a);
    new Consumer(a);
  }
}
```

Результат работы этой программы непредсказуем. Программа может, например, записать подряд 5 сообщений и прочитать только пятое, или зациклиться на чтении какого-нибудь сообщения, номер которого меньше пяти.

На самом деле здесь требуется механизм для взаимодействия потоков. Действительно, поток-читатель не должен читать сообщение, пока оно не записано. Поток-писатель, в свою очередь, не должен писать сообщение, пока оно не прочитано.

Очевидно, ситуацию можно улучшить в конкретном случае, управляя выключением потоков при помощи метода **`sleep`**, однако этот подход принципиально неверный.

Для взаимодействия потоков в Java предусмотрены следующие методы класса **`Thread`**:

wait — переводит вызывающий поток в состояние ожидания, выводит его из монитора и ждет уведомления от другого потока при помощи метода **notify**.

notify — возобновляет выполнение потока, вызвавшего метод **wait**.

Рассмотрим, как используются указанные методы для организации правильного взаимодействия потоков писателя и читателя. Изменения касаются только класса **AC**:

```
class AC {
  PC p;
  Boolean isSet = false;
  AC(PC p) {
    this.p = p;
  }
  synchronized int read() {
    if ( ! isSet ) {
      try {
        wait();
      } catch(InterruptedException e) {}
    }
    System.out.println("Read=" + p.a);
    isSet = false;
    notify();
    return p.a;
  }
  synchronized void write(int a) {
    if ( isSet ) {
      try {
        wait();
      } catch(InterruptedException e) {}
    }
    p.a = a;
    System.out.println("Write=" + a);
    isSet = true;
    notify();
  }
}
```

Здесь используется логическая переменная **isSet**, которая указывает на то, что сообщение записано. Первоначальное значение этой переменной равно лжи, что означает, что первоначально не записано никакого сообщения.

Метод **read**, обнаружив, что значение не записано, при помощи метода **wait** переходит в состояние ожидания (выходя при этом из монитора). Поскольку монитор свободен, метод **write** входит в него, записывает сообщение, устанавливает флаг **isSet**, уведомляет при помощи метода **notify**, что сообщение записано.

Теперь метод **read** выходит из состояния ожидания, читает сообщение, сбрасывает флаг **isSet**, и уведомляет о выходе из монитора при помощи метода **notify**.

Вспомогательные классы

Лексический анализатор

С помощью класса `StringTokenizer` выполняется лексический разбор строки на составляющие её лексемы. Лексема — часть строки, имеющая некоторый смысл в контексте разбора. Примером является анализ текста программы. Конструкторы класса:

```
StringTokenizer(String str)
StringTokenizer(String str, String delimeters)
StringTokenizer(String str, String delimeters, boolean delimAsToken)
```

Второй параметр конструктора — это символы, отделяющие лексемы друг от друга, например, пробел, запятая, точка с запятой. Третий параметр указывает, что символы-разделители сами являются лексемами; такими символами являются, например, знаки операций.

Основные методы класса `StringTokenizer`:

`int countTokens` — возвращает число оставшихся лексем;

`boolean hasMoreElements`, `boolean hasMoreTokens` — возвращают `true`, если в строке еще есть лексемы;

`Object nextElement` — возвращает следующую лексему;

`String nextToken` — возвращает следующую лексему.

Методы `hasMoreElements` и `nextElement` отличаются тем, что реализуют интерфейс `Enumeration`.

Следующий пример демонстрирует разбор выражения.

Пример кода 35 — Лексический разбор

```
import java.util.StringTokenizer;
public class tokens {
  static String s = "5*2+3";
  public static void main(String[] args) {
    StringTokenizer st = new StringTokenizer(s, "*+", true);
    while (st.hasMoreTokens()) {
      System.out.println(st.nextToken());
    }
  }
}
```

Результатом работы программы является вывод отдельных символов заданного выражения.

Битовые массивы

Класс `BitSet` создает массив битов. Массив может увеличиваться в размере настолько, насколько это необходимо.

Конструкторы:

```
BitSet()
BitSet(int size)
```

Второй конструктор задает массив заданного размера.

Основные методы класса `BitSet`:

`void and(BitSet bitSet)` — выполняет операцию AND;

`void or(BitSet bitSet)` — выполняет операцию OR;

`void xor(BitSet bitSet)` — выполняет операцию XOR;

`void andNot(BitSet bitSet)` — выполняет операцию AND NOT (очищает биты вызывающего объекта, которые соответствуют единичным битам в массиве). Во всех приведенных выше операциях результат записывается в вызывающий объект;

`boolean get(int index)` — возвращает значение бита `index`;

`void clear(int index)` — устанавливает нулевое значение бита `index`;

`void set(int index)` — устанавливает единичное значение бита `index`.

Классы для работы с датой

Класс `Date` инкапсулирует дату и время. Конструкторы:

```
Date()
Date(long milliseconds)
```

Первый конструктор создает объект с текущими датой и временем. Второй конструктор задает дату и время из расчета числа миллисекунд, прошедших с даты 01/01/1970.

Основные методы:

`boolean after(Date date)` — возвращает `true`, если вызывающий объект содержит более позднюю дату, чем аргумент;

`boolean before(Date date)` — возвращает `true`, если вызывающий объект содержит более раннюю дату, чем аргумент;

`long getTime()` — возвращает число миллисекунд, прошедших с даты 01/01/1970;

`long setTime(long milliseconds)` — устанавливает текущую дату, эквивалентную числу миллисекунд, прошедших с даты 01/01/1970.

Класс `GregorianCalendar` представляет конкретную реализацию абстрактного класса `Calendar`. С его помощью можно оперировать такими понятиями, как год, месяц, число и т.п. Конструкторы:

```
GregorianCalendar(int year, int month, int day)
GregorianCalendar(int year, int month, int day, int hours, int mins)
GregorianCalendar(int year, int month, int day, int hours, int mins,
int seconds)
```

Параметр `year` задает год. Номер месяца `month` задается от нуля (январь) до одиннадцати (декабрь). Параметр `day` задает день месяца. Первый конструктор устанавливает время, равное полуночи. Другие конструкторы задают конкретное время.

Следующий пример показывает использование классов даты и григорианского календаря.

Пример кода 36 — Дата и календарь

```
import java.util.*;
public class dates {
  public static void main(String[] args) {
    Date d = new Date(86400000);
    System.out.println(d);
    GregorianCalendar gc = new GregorianCalendar(2011, 4, 1);
    System.out.println(gc.get(Calendar.YEAR));
    System.out.println(gc.get(Calendar.MONTH));
    System.out.println(gc.get(Calendar.DATE));
    System.out.println(gc.isLeapYear(2012));
  }
}
```

Здесь использованы константы класса `Calendar`. Программа выводит сначала дату 01/02/1970 (86400000 мс в точности равно одним суткам), а затем установленные компоненты года, месяца и дня месяца. В заключение определяется, является ли високосным 2012 год.

Генератор псевдослучайных чисел

Класс `Random` является генератором случайных чисел. Конструкторы:

```
Random()
Random(long seed)
```

Второй конструктор задает начальное значение случайного числа. Следующий пример поясняет разницу в использовании конструкторов.

Пример кода 36 — Случайные числа

```
import java.util.*;
public class randoms {
  public static void main(String[] args) {
    Random a = new Random(), b = new Random(1);
    System.out.println(a.nextInt());
    System.out.println(b.nextInt());
    a = new Random(); b = new Random(1);
    System.out.println(a.nextInt());
    System.out.println(b.nextInt());
  }
}
```

Программа выводит всего три разных числа: второе и четвертое числа совпадают, поскольку для их генерации использовано одно и то же начальное значение `1`.

Класс `Random` определяет следующие основные методы:

`void nextBytes(byte v[])` — заполняет указанный массив случайными значениями.

`X nextX()` — возвращает следующее случайное число типа `x`; при этом `x` может иметь тип `boolean`, `double`, `float`, `int`, `long`.

Наблюдение за объектами

Класс `Observable` используется для создания подклассов, за которыми могут наблюдать другие части программы. Когда объект этого типа изменяется, наблюдающие классы уведомляются об этом. Наблюдающий класс должен реализовывать интерфейс `Observer`.

Основные методы класса `Observable`:

`void addObserver(Observer o)` — добавляет наблюдателя в список;

`void clearChanged()` — делает объект неизменяемым;

`int countObservers()` — возвращает число наблюдателей;

`boolean hasChanged()` — признак изменения наблюдаемого объекта;

`void notifyObservers()` — уведомляет всех наблюдателей об изменении;

`void setChanged()` — устанавливает измененное состояние объекта.

Интерфейс `Observer` определяет только один метод:

`void update(Observable o, Object a)` — вызывается, когда наблюдаемый объект изменился. Параметры: наблюдаемый объект и аргумент, передаваемый методу `notifyObservers`.

Следующий пример демонстрирует наблюдение за объектом.

Пример кода 37 — Наблюдение за объектом

```
import java.util.*;
class Watcher implements Observer {
  public void update(Observable o, Object a) {
    System.out.println("count=" + ((Integer)a).intValue());
  }
}
class Watched extends Observable {
  public int count;
  void start() {
    for (count = 0; count < 10; count++) {
      setChanged();
      notifyObservers(new Integer(count));
      try {
        Thread.sleep(250);
      } catch(Exception e) { }
    }
  }
}
public class observe {
  public static void main(String[] args) {
    Watched a = new Watched();
    Watcher w = new Watcher();
    a.addObserver(w);
    a.start();
  }
}
```

Здесь новое состояние наблюдаемого объекта типа `Watched` передается наблюдателю типа `Watcher` методом `notifyObservers`.

Ввод и вывод

Поддержку операций ввода и вывода обеспечивает пакет `java.io`, в котором определено около пятидесяти классов. Большинство классов этого пакета обеспечивает работу с потоками.

Работа с файлами и файловой системой

Класс `File` обеспечивает работу непосредственно с файлами и файловой системой. Каталог рассматривается как файл с дополнительным свойством `list`, которое представляет собой список имен файлов.

Конструкторы класса `File`:

```
File(String directoryPath)
File(String directoryPath, String fileName)
File(File dirObject, String fileName)
```

Здесь `directoryPath` — путь к файлу, `fileName` — имя файла, `dirObject` — объект типа `File`, указывающий на каталог. Следующий пример показывает использование конструктора и некоторых элементов класса `File`.

Пример кода 38 — Исследование файловой системы

```
import java.io.*;
class dirlist {
  public static void main(String[] args) {
    String d = "C:/";
    File f = new File(d);
    if ( f.isDirectory() ) {
      String[] s = f.list();
      for (int i = 0; i < s.length; i++) {
        File a = new File(d + "/" + s[i]);
        System.out.print(s[i]);
        if ( a.isDirectory() ) {
          System.out.println(" - DIR");
        } else if (a.isFile()) {
          System.out.println(" - FILE");
          System.out.println("AbsolutePath=" + a.getAbsolutePath());
          System.out.println("ParentDir=" + a.getParent());
          System.out.println("CanRead=" + a.canRead());
          System.out.println("CanWrite=" + a.canWrite());
          System.out.println("IsHidden=" + a.isHidden());
          System.out.println("LastModified=" + a.lastModified());
          System.out.println("Size=" + a.length());
        }
      }
    }
  }
}
```

Как видно из примера, методы класса `File` позволяют определить основные характеристики файла или каталога. Заметим, что при задании пути к файлу можно использовать как прямые, так и обратные слеши, но при использовании последних их нужно дублировать, как и в Си.

Метод `renameTo` переименовывает файл, если это возможно. Метод `delete` удаляет файл или каталог (если каталог пуст). Метод `mkdir` создает каталог, а метод `mkdirs` создает каталог и его родительские каталоги.

Интерфейс `FilenameFilter` позволяет ограничить количество файлов, возвращаемых методом `list`. Интерфейс определяет метод `accept`. Использование фильтра файлов поясняет следующий пример.

Пример кода 39 — Фильтрация файлов

```
import java.io.*;
class filt implements FilenameFilter {
  String e;
  public filt(String e) {
    this.e = "." + e;
  }
  public boolean accept(File dir, String name) {
    return name.endsWith(e);
  }
}
class filefilt {
  public static void main(String[] args) {
    String d = "C:/";
    File f = new File(d);
    FilenameFilter t = new filt("txt");
    String[] s = f.list(t);
    for (int i = 0; i < s.length; i++) {
      System.out.println(s[i]);
    }
  }
}
```

Здесь класс `filt` задает фильтр для файлов типа "txt".

Байтовые потоки

Четыре абстрактных класса формируют поточный ввод и вывод.

`Классы InputStream` и `OutputStream` предназначены для байтовых потоков, классы `Reader` и `Writer` — для символьных.

Методы класса `InputStream`:

`int available()` — возвращает число байт, доступных для чтения;

`void close()` — закрывает источник;

`void mark(int numBytes)` — размещает маркер в текущую точку входного потока, который остается действительным, пока не считано указанное количество байт;

`boolean markSupported()` — возвращает `true`, если входной поток поддерживает методы `mark` и `reset`;

`int read()` — возвращает числовое представление очередного байта или –1, если встретился конец потока;

`int read(byte[] buffer)` — читает `buffer.length` байтов в буфер `buffer` и возвращает считанное количество байтов;

`int read(byte[] buffer, int offset, int numBytes)` — читает в буфер указанное количество байтов, начиная с позиции `offset`;

`void reset()` — переустанавливает входной указатель на предварительно установленный маркер;

`long skip(long numBytes)` — пропускает указанное количество байтов.

Методы класса `OutputStream`:

`void close()` — закрывает выходной поток;

`void flush()` — вызывает запись файловых буферов на диск;

`void write(int b)` — записывает в выходной поток один байт;

`void write(byte[] buffer)` — записывает буфер в выходной поток;

`void write(byte[] buffer, int offset, int numBytes)` — записывает в выходной поток указанное количество байтов в позицию `offset`.

Класс `FileInputStream` реализует методы класса `InputStream` за исключением `mark` и `reset`. Два его основных конструктора могут выбрасывать исключение `FileNotFoundException`:

```
FileInputStream(String filePath)
FileInputStream(File fileObject)
```

Следующий пример показывает использование этого класса.

Пример кода 40 — Чтение потока байт

```
import java.io.*;
class fisdemo {
  public static void main(String[] args) throws Exception {
    InputStream f = new FileInputStream("fisdemo.java");
    int size = f.available();
    System.out.println("Available: " + size);
    int m, n = size / 2;
    byte[] b = new byte[n];
    m = f.read(b);
    System.out.println(new String(b, 0, n));
    System.out.println("Available: " + (size - m));
    while ( f.available() > 0 ) {
      System.out.print( (char) f.read() );
    }
  }
}
```

Здесь выполняется чтение файла приведенного кода и вывод первой половины с помощью буфера, а оставшейся части — посимвольно. Заметим, что методы класса `FileInputStream` также могут выбрасывать исключение `IOException`.

Для записи потоков байт используется класс `FileOutputStream`, реализующий методы класса `OutputStream`. Конструкторы:

```
FileOutputStream(String filePath)
FileOutputStream(File fileObject)
FileOutputStream(File fileObject, boolean append)
```

Третий конструктор открывает файл в режиме добавления.

Следующий пример показывает применение класса `FileOutputStream`.

Пример кода 41 — Запись потока байт

```
import java.io.*;
class fosdemo {
  public static void main(String[] args) throws Exception {
    String s = "This is a sample of text.\n";
    byte[] b = s.getBytes();
    OutputStream f = new FileOutputStream("t1.txt");
    for ( int i = 0; i < b.length; i++ ) {
      f.write(b[i]);
    }
    f.close();
    f = new FileOutputStream("t2.txt");
    f.write(b);
    f.close();
  }
}
```

Здесь текст записывается посимвольно и из буфера.

Кроме описанных, пакет `java.io` содержит также несколько классов для фильтрованного и буферизованного ввода и вывода, которые обладают бóльшими возможностями.

Символьные потоки

Классы байтовых потоков не позволяют выполнять операции с символами *Unicode*. Эти операции выполняются классами символьного ввода и вывода.

Абстрактные классы `Reader` и `Writer` определяют такие же методы, что и аналогичные классы байтовых потоков. Для чтения символьных потоков используется класс `FileReader`. Следующий пример показывает применение этого класса.

Пример кода 42 — Чтение потока символов

```
import java.io.*;
class frdemo {
  public static void main(String[] args) throws Exception {
    FileReader fr = new FileReader("frdemo.java");
    BufferedReader br = new BufferedReader(fr);
    String s;
    while ( (s = br.readLine()) != null ) {
      System.out.println(s);
    }
    fr.close();
  }
}
```

Здесь также используется класс `BufferedReader` для буферизованного чтения файла. Исходный текст кода читается и выводится построчно.

Для записи символьных потоков используется класс `FileWriter`. Следующий пример показывает его применение.

Пример кода 43 — Запись потока символов

```
import java.io.*;
class fwdemo {
  public static void main(String[] args) throws Exception {
    String s = "This is a sample of text.\n";
    char[] b = new char[s.length()];
    s.getChars(0, s.length(), b, 0);
    FileWriter fw = new FileWriter("t3.txt");
    fw.write(b);
    fw.flush();
    fw.close();
  }
}
```

Сериализация

Сериализация (*serialization*) — это запись состояния объекта в форме байтового потока. Обратный процесс восстановления объекта из потока байтов называется десериализацией (*deserialization*).

Средства сериализации могут работать только с объектами, реализующими интерфейс `Serializable`. Данный интерфейс не имеет никаких элементов и только указывает, что класс может быть сериализован.

Интерфейс `ObjectOutput` расширяет интерфейс `DataOutput` и поддерживает сериализацию объектов. Его методы:

`void close()` — закрывает поток;

`void flush()` — записывает буферы потока на диск;

`void writeObject(Object o)` — записывает в поток объект.

Кроме перечисленных, класс обладает методами `write`.

Класс `ObjectOutputStream` расширяет интерфейс `OutputStream` и реализует интерфейс `ObjectOutput`. Следующий пример показывает, как с помощью данного класса сериализовать объект.

Пример кода 44 — Сериализация объекта в файл

```
import java.io.*;
class datas implements Serializable {
  public int a;
  public String b;
  datas(int a, String b) { this.a = a; this.b = b; }
}
class oser {
  public static void main(String[] args) {
    try {
      datas d = new datas(1, "data");
      FileOutputStream fos = new FileOutputStream("object.bin");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      oos.writeObject(d);
      oos.flush();
      oos.close();
    } catch(Exception e) {}
  }
}
```

Здесь создается объект простейшего класса **datas**, который записывается в файл "**object.bin**". Процесс восстановления этого объекта из файла демонстрирует следующий пример.

Пример кода 45 — Восстановление объекта из файла

```
import java.io.*;
class odes {
  public static void main(String[] args) {
    try {
      datas d;
      FileInputStream fis = new FileInputStream("object.bin");
      ObjectInputStream ois = new ObjectInputStream(fis);
      d = (datas) ois.readObject();
      ois.close();
      System.out.println(d.a);
      System.out.println(d.b);
    } catch(Exception e) {}
  }
}
```

Здесь используется класс **ObjectInputStream**, который расширяет интерфейс **InputStream** и реализует интерфейс **ObjectInput**.

Работа с базами данных

Структура JDBC

Модель JDBC похожа на модель ODBC: программы взаимодействуют с диспетчером драйверов JDBC, который использует подсоединенные драйверы для организации взаимодействия с базой данных. Для работы достаточно использовать JDBC API (рисунок 1).

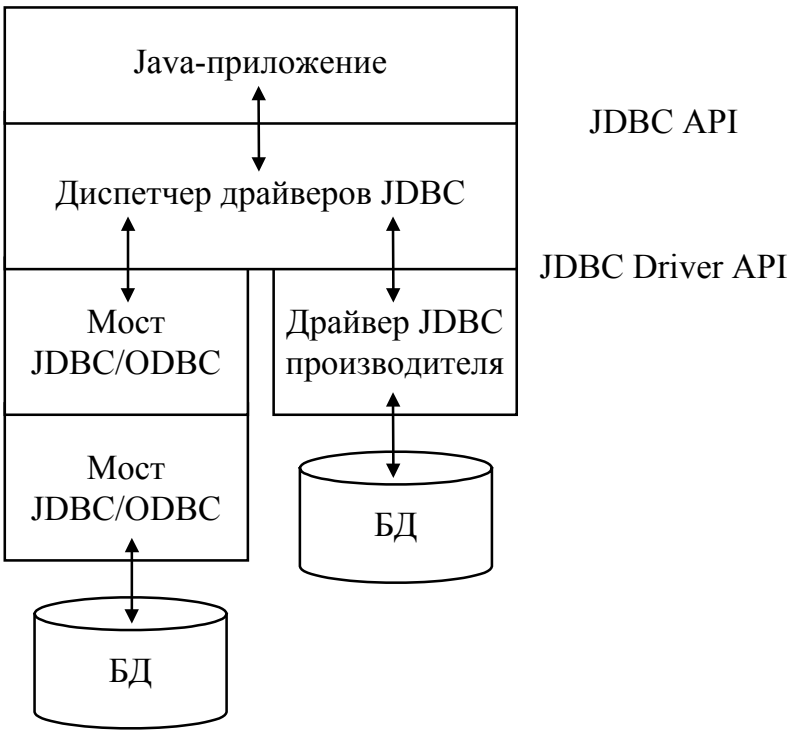

Рисунок 1 — Взаимодействие JDBC и базы данных

Типы JDBC-драйверов

Драйвер типа 1. Транслирует JDBC в ODBC и использует драйвер ODBC для взаимодействия с базой данных. Таким драйвером является, например, мост JDBC/ODBC.

Драйвер типа 2. Создается на языке Java для взаимодействия с клиентским API базы данных. Требует дополнительное ПО.

Драйвер типа 3. Создается на основе библиотеки Java, в которой используется независимый от базы данных протокол взаимодействия сервера и базы.

Драйвер типа 4. Представляет собой библиотеку Java, которая транслирует вызовы JDBC-запросы непосредственно в протокол конкретной базы данных.

Большинство поставщиков баз данных применяют драйверы 3 и 4.

Типичные примеры использования JDBC

Согласно традиционной модели клиент/сервер, графический пользовательский интерфейс реализуется на стороне клиента, а база данных располагается на стороне сервера (рисунок 2).

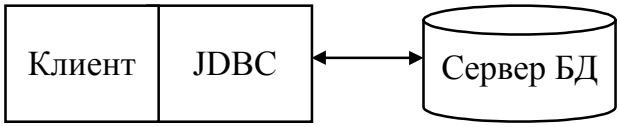

Рисунок 2 — Приложение клиент/сервер

Более современная архитектура на основе трехуровневой или многоуровневой модели предполагает обращение клиента не к серверу БД, а к промежуточному слою, на котором формируется бизнес-логика, которая осуществляет подключение и запросы к базе данных (рисунок 3).

Рисунок 3 — Трехуровневая модель

Конфигурирование JDBC

Прежде всего требуется база данных (сервер базы данных), совместимая с JDBC. Таковыми являются, например, IBM DB2, Microsoft SQL Server, Oracle, PostgreSQL. Далее нужно создать тестовую базу данных, с названием, например, JAVATEST. Рекомендуется также установить базу Apache Derby, которая является частью JDK 6.

Для установления соединения с базой данных нужно указать ряд параметров в виде URL:

```
jdbc:название_протокола:другие_сведения ,
```

например:

```
jdbc:derby://localhost:1527/JAVATEST;create=true
jdbc:postgresql:/JAVATEST
```

Далее нужно получить JAR-файл, в котором находится драйвер для базы данных. Если используется Derby, требуется файл derbyclient.jar.

При запуске программы, обращающейся к базе данных, требуется добавить JAR-файл драйвера в параметр classpath (для компиляции JAR-файл не нужен).

Команда для запуска программы из командной строки:

```
java -classpath .;jar-файл_драйвера имя_программы
```

Прежде, чем подключиться к базе данных, часто нужно запустить сервер базы данных. Для базы данных Derby нужно выполнить следующие действия:

1. Открыть командную строку и выбрать каталог базы данных.

2. Найти файл derbyrun.jar. В одних версиях он находится в каталоге jdk/db/lib, в других — в каталоге JavaDB. Будем обозначать каталог, содержащий lib/derbyrun.jar как derby.

3. Выполнить команду

```
java -jar derby/lib/derbyrun.jar server start
```

4. Необходимо удостовериться, что база данных работает должным образом. Создать файл ij.properties с такими строками:

```
ij.driver=org.apache.derby.jdbc.ClientDriver
ij.protocol=jdbc:derby://localhost:1527/
ij.database=JAVATEST;create=true
```

В другой командной строке запустить интерактивный инструментарий ij для написания сценариев Derby, выполнив команду:

```
java -jar derby/lib/derbyrun.jar ij -p ij.properties
```

Теперь можно выполнять команды SQL, например:

```
CREATE TABLE greetings (message CHAR(20));
INSERT INTO TABLE greetings VALUES ('Hello, World!');
SELECT * FROM greetings;
DROP TABLE greetings;
```

Чтобы выйти, введите команду

```
EXIT;
```

5. Чтобы остановить сервер базы данных, ввести команду

```
java -jar derby/lib/derbyrun.jar server shutdown
```

Регистрация класса драйвера

Некоторые JAR-файлы JDBC (например, драйвер Derby) автоматически регистрируют класс драйвера. JAR-файл может автоматически зарегистрировать класс драйвера, если он содержит файл

```
META-INF/services/java.sql.Driver
```

Чтобы проверить это, нужно распаковать JAR-файл драйвера.

Если JAR-файл не поддерживает автоматическую регистрацию, нужно узнать имена классов драйвера JDBC поставщика. Обычными именами являются:

```
org.apache.derby.jdbc.ClientDriver
org.postgresql.Driver
```

Есть два способа регистрации драйвера посредством DriverManager.

Первый — загрузить класс драйвера в программу, например:

```
Class.forName("org.postgresql.Driver");
```

Второй — задать свойство jdbc.drivers при помощи команды

```
java -Djdbc.drivers=org.postgresql.Driver имя_программы
```

Соединение с базой данных

Следующий код устанавливает соединение с базой данных:

```
String url = "jdbc:postgresql:JAVATEST";
String user = "dbuser";
String pass = "secret";
Connection conn = DriverManager.getConnection(url, user, pass);
```

Работа с изображениями

Изображение — это прямоугольный графический объект. Изображения в Java представлены объектами класса `Image`, являющегося частью пакета `java.awt.image`.

Создание, загрузка и просмотр изображений

Изображение создается методом `createImage` класса `Component`. Есть две формы этого метода:

```
Image createImage(ImageProducer)
Image createImage(int width, int height)
```

В первом случае изображение создается объектом класса, реализующего интерфейс `ImageProducer`. Во втором случае создается пустое изображение заданного размера. Пример:

```
Canvas c = new Canvas();
Image img = c.createImage(100, 100);
```

На пустом изображении можно рисовать, если получить графический контекст изображения. Следующий пример показывает, как нарисовать красный пиксель на черном фоне на изображении внутри коричневого апплета.

```
/* <APPLET code="image_test" width="300" height="300"></APPLET> */
import java.applet.*;
import java.awt.*;
public class image_test extends Applet {
  Image img;
  public void init() {
    setBackground(new Color(112, 64, 32));
    img = createImage(100, 100);
    Graphics g = img.getGraphics();
    g.setColor(Color.BLACK);
    g.fillRect(0, 0, 100, 100);
    g.setColor(Color.RED);
    g.fillRect(1, 1, 1, 1);
  }
  public void paint(Graphics g) {
    g.drawImage(img, 10, 10, null);
  }
}
```

Этот пример показывает также, как изображение выводится в окно при помощи метода `drawImage`. Последний параметр метода задает объект типа `ImageObserver`, контролирующий изображение.

Изображение можно также загрузить из файла при помощи метода

```
Image getImage(URL url)
Image getImage(URL url,String imageName)
```

Пример загрузки изображения в апплет:

```
/* <APPLET code="image_test" width="300" height="300"></APPLET> */
import java.applet.*;
import java.awt.*;
import java.net.*;
public class image_test extends Applet {
  Image img;
  public void init() {
    setBackground(new Color(112, 64, 32));
    try {
      img = getImage(new URL(getCodeBase() + "ok.gif"));
    } catch(Exception e) {}
  }
  public void paint(Graphics g) {
    g.drawImage(img, 10, 10, null);
  }
}
```

В папке класса апплета должен находиться файл картинки **ok.gif**.

Интерфейс ImageObserver

Этот интерфейс используется для уведомления о выводе изображения. Он определяет один метод **imageUpdate**, с помощью которого можно определить текущее состояние вывода изображения. Объект наблюдателя позволяет параллельно с загрузкой выполнять другие действия.

Следующий пример показывает, как можно переопределить метод **imageUpdate** класса **Applet**.

```
/* <APPLET code="image_test" width="300" height="300"></APPLET> */
import java.applet.*;
import java.awt.*;
import java.net.*;
public class image_test extends Applet {
  Image img;
  public void init() {
    setBackground(new Color(112, 64, 32));
    try {
      img = getImage(new URL(getCodeBase() + "ok.gif"));
    } catch(Exception e) {}
  }
  public void paint(Graphics g) {
    g.drawImage(img, 10, 10, this);
  }
  public boolean imageUpdate(Image img, int flags,
                             int x, int y, int w, int h) {
    if ((flags & SOMEBITS) != 0) {
      // рисовать частичные данные
      repaint(x, y, w, h);
    } else {
      // рисовать весь апплет
      repaint();
    }
    return (flags & (ALLBITS | ABORT)) == 0;
  }
}
```

Класс MediaTracker

`MediaTracker` создает объект, который будет параллельно проверять состояние произвольного числа изображений. Для отслеживания состояния загрузки изображения используется метод `addImage`.

```
void addImage(Image img, int imgID)
void addImage(Image img, int imgID, int width, int height)
```

Идентификатор изображения `imgID` должен быть уникальным. Для проверки состояния вызывают метод `checkID` с идентификатором изображения в качестве параметра. Метод `checkAll` проверяет загрузку всех изображений. Следующий пример показывает использование класса.

```
/* <APPLET code="image_test" width="300" height="300"></APPLET> */
import java.applet.*;
import java.awt.*;
import java.net.*;

public class image_test extends Applet {
  Image img;
  MediaTracker t;
  public void init() {
    setBackground(new Color(112, 64, 32));
    try {
      img = getImage(new URL(getCodeBase() + "ok.gif"));
      t = new MediaTracker(this);
      t.addImage(img, 0);
    } catch(Exception e) {}
  }
  public void paint(Graphics g) {
    if (t.checkID(0, true)) {
      g.drawImage(img, 10, 10, null);
    }
  }
}
```

Интерфейс ImageProducer

С помощью этого интерфейса создаются объекты, которые поставляют данные для объектов. `MemoryImageSource` — это класс, который создает изображение из массива данных. Примером класса, который реализует этот интерфейс, является `MemoryImageSource`.

```
/* <APPLET code="image_test" width="300" height="300"></APPLET> */
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
public class image_test extends Applet {
  Image img;
  public void init() {
    Dimension d = getSize();
    int w = d.width;
    int h = d.height;
    int pixels[] = new int[w * h];
    int i = 0;
```

```
    for (int y = 0; y < h; y++) {
      for (int x = 0; x < w; x++) {
        int r = (x ^ y) & 0xFF;
        int g = (x * 2 ^ y * 2) & 0xFF;
        int b = (x * 4 ^ y * 4) & 0xFF;
        pixels[i++] = (254 << 24) | (r << 16) | (g << 8) | b;
      }
    }
    // параметры - ширина, высота, пиксели, смещение, строк
    img = createImage(new MemoryImageSource(w, h, pixels, 0, w));
  }
  public void paint(Graphics g) {
    g.drawImage(img, 0, 0, null);
  }
}
```

Интерфейс ImageConsumer

Интерфейс **ImageConsumer** является противоположностью интерфейса **ImageProducer**. Класс **PixelGrabber** реализует этот интерфейс. В примере с помощью объекта этого класса берется существующее изображение и строится массив пиксельных данных.

```
/* <APPLET code="image_test" width="300" height="300"></APPLET> */
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
import java.net.*;
public class image_test extends Applet {
  Image img, ima;
  MediaTracker t;
  public void init() {
    setBackground(new Color(112, 64, 32));
    try {
      img = getImage(new URL(getCodeBase() + "ok.gif"));
      t = new MediaTracker(this);
      t.addImage(img, 0);
      t.waitForID(0);
      int w = img.getWidth(null), h = img.getHeight(null);
      int s = w * h, pixels[] = new int[s];
      PixelGrabber g = new PixelGrabber(img, 0, 0, w, h, pixels, 0, w);
      g.grabPixels();
      for (int i = 0; i < s; i++) {
        pixels[i] = ~pixels[i] | 0xFF000000;
      }
      ima = createImage(new MemoryImageSource(w, h, pixels, 0, w));
    } catch(Exception e) {}
  }
  public void paint(Graphics g) {
    g.drawImage(ima, 0, 0, null);
    g.drawImage(img, 100, 100, null);
  }
}
```

Фильтр CropImageFilter

Этот фильтр вырезает из исходного изображения прямоугольную область. В примере из исходного изображения вырезается левая верхняя часть.

```
/* <APPLET code="image_test" width="300" height="300"></APPLET> */
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
import java.net.*;
public class image_test extends Applet {
  Image img, ima;
  MediaTracker t;
  CropImageFilter f;
  FilteredImageSource s;
  public void init() {
    setBackground(new Color(112, 64, 32));
    try {
      img = getImage(new URL(getCodeBase() + "ok.gif"));
      t = new MediaTracker(this);
      t.addImage(img, 0);
      t.waitForID(0);
      f = new CropImageFilter(0, 0, 31, 25);
      s = new FilteredImageSource(img.getSource(), f);
      ima = createImage(s);
      t.addImage(ima, 1);
      t.waitForAll();
    } catch(Exception e) {}
  }
  public void paint(Graphics g) {
    g.drawImage(ima, 0, 0, null);
    g.drawImage(img, 100, 100, null);
  }
}
```