Иллюстрированный самоучитель по Java

Классы-оболочки

Работа со строками

Классы-коллекции

Классы-утилиты

Классы-оболочки

- ## Классы-оболочки
  Java – полностью объектно-ориентированный язык. Это означает, что все, что только можно, в Java представлено объектами. | Восемь примитивных типов нарушают это правило. Они оставлены в Java из-за многолетней привычки к числам и символам.
- ## Числовые классы. Класс Boolean.
  В каждом из шести числовых классов-оболочек есть статические методы преобразования строки символов типа string представляющей число, в соответствующий примитивный тип: Byte.parseByte(), Double.parseDouble(), Float.parseFloat(), Integer.parselnt(), Long.parseLong(), Short.parseShort().
- ## Класс Character
  В этом классе собраны статические константы и методы для работы с отдельными символами. | Статический метод: | digit(char ch, in radix) | …переводит цифру ch системы счисления с основанием radix в ее числовое значение типа int.
- ## Класс BigInteger
  Все примитивные целые типы имеют ограниченный диапазон значений. В целочисленной арифметике Java нет переполнения, целые числа приводятся по модулю, равному диапазону значений. | Для того чтобы было можно производить целочисленные вычисления с любой разрядностью, в состав Java API введен класс BigInteger, хранящийся в пакете java.math.
- ## Класс Big Decimal
  Класс BigDecimal расположен В пакете java.math. | Каждый объект этого класса хранит два целочисленных значения: мантиссу вещественного числа в виде объекта класса BigInteger, и неотрицательный десятичный порядок числа типа int.
- ## Класс Class
  Класс Object, стоящий во главе иерархии классов Java, представляет все объекты, действующие в системе, является их общей оболочкой. Всякий объект можно считать экземпляром класса Object. | Класс с именем class представляет характеристики класса, экземпляром которого является объект.

Классы-оболочки

Java – полностью объектно-ориентированный язык. Это означает, что все, что только можно, в Java представлено объектами.

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

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

На рис. 4.1 показана одна из ветвей иерархии классов Java. Для каждого примитивного типа есть соответствующий класс. Числовые классы имеют общего предка – абстрактный класс Number, в котором описаны шесть методов, возвращающих числовое значение, содержащееся в классе, приведенное к соответствующему примитивному типу: **bytevalue ()**,**doubievalue ()**, **floatValue ()**, **intValue()**, **longValue ()**, **shortValue ()**. Эти методы переопределены в каждом из шести числовых классов-оболочек.

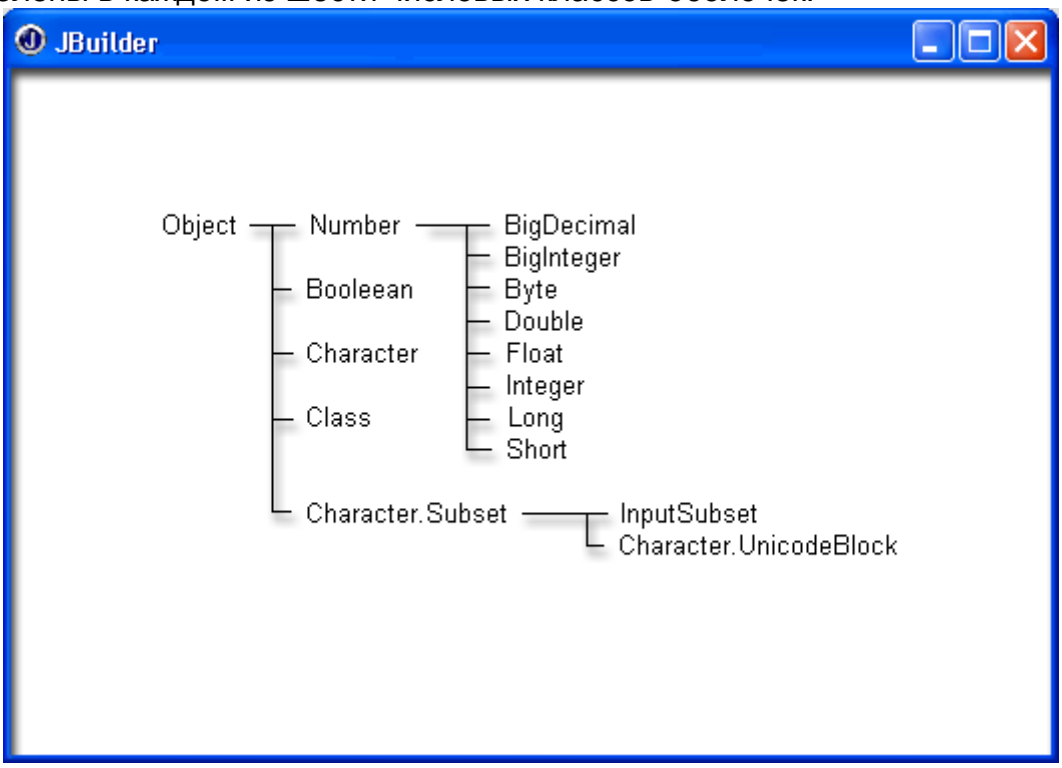

Рис. 4.1. Классы примитивных типов

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

- 0, если значения равны;
- отрицательное число (-1), если числовое значение в данном объекте меньше, чем в объекте-аргументе;

- положительное число (+1), если числовое значение в данном объекте больше числового значения, содержащегося в аргументе.

Что полезного в классах-оболочках?

Числовые классы. Класс Boolean.

В каждом из шести числовых классов-оболочек есть статические методы преобразования строки символов типа string представляющей число, в соответствующий примитивный тип:**Byte.parseByte()**, **Double.parseDouble()**, **Float.parseFloat()**, **Integer.parseInt()**,**Long.parseLong()**, **Short.parseShort()**. Исходная строка типа string, как всегда в статических методах, задается как аргумент метода. Эти методы полезны при вводе данных в поля ввода, обработке параметров командной строки, т. е. всюду, где числа представляются строками цифр со знаками плюс или минус и десятичной точкой.

В каждом из этих классов есть статические константы MAX_VALUE и MIN_VALUE, показывающие диапазон числовых значений соответствующих примитивных типов. В классах Double и Float есть еще константы POSITIVE_INFINITY, NEGATIVE_INFINITY, NaN, о которых шла речь в **главе 1**, и логические методы проверки isNan(), isInfinite().

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

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

В листинге 4.1 показано применение этих методов, а рис. 4.2 демонстрирует вывод результатов.

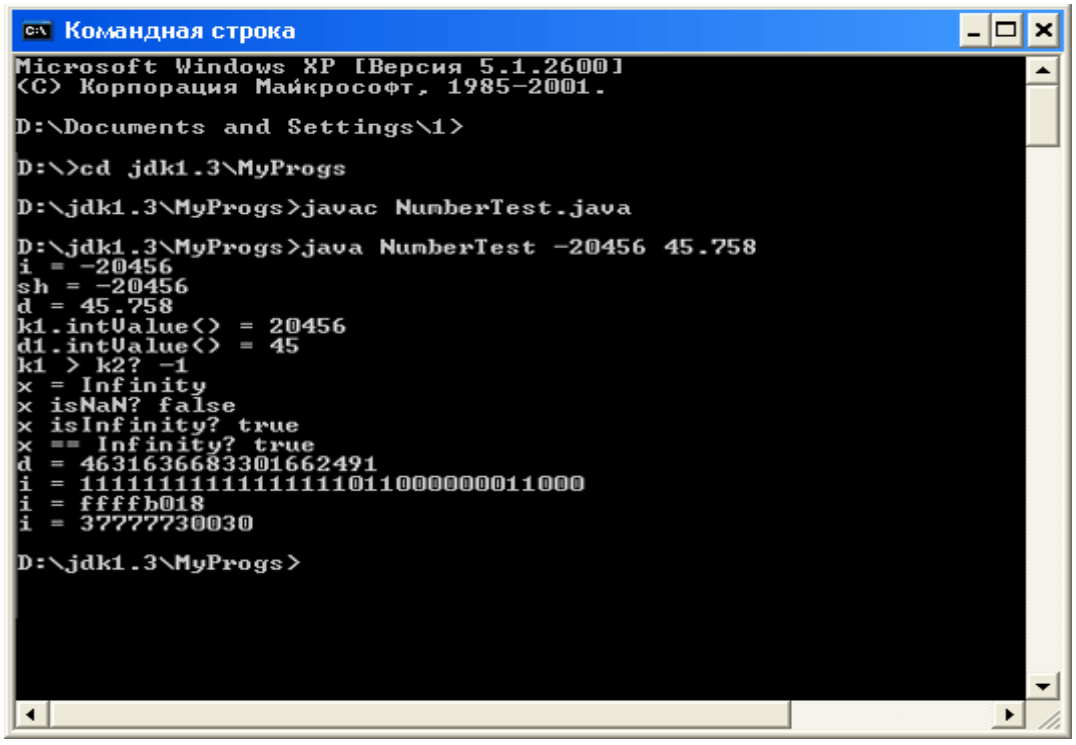

Рис. 4.2. Методы числовых классов

Листинг 4.1. Методы числовых классов.

```
class NumberTest{
public static void main(String[] args){
int i = 0;
short sh = 0;
double d = 0;
Integer k1 = new Integer(55);
Integer k2 = new Integer(100);
Double dl = new Double(3.14);
try{
i = Integer.parselnt(args[0]);
sh = Short.parseShort(args[0]);
d = Double.parseDouble(args[1]);
dl = new Double(args[1]);
kl = new Integer(args[0]);
}catch(Exception e){}
double x = 1.0/0.0;
System.out.println("i = " + i);
System.outjprintln("sh – " + sh);
System.out.println("d. = " + d);
System.out.println("kl.intValue() = " + kl.intValue());
System.out.println("dl.intValue() '= "'+ dl.intValuei));
System.out.println("k1 > k2? " + kl.compareTo(k2));
System.out.println ("x = " + x);
System.out.println("x isNaN? " + Double.isNaN(x));
System.out.println("x islnfinite? " + Double.islnfinite(x));
System.out.println("x == Infinity? " +
(x == Double.POSITIVE_INFINITY));
System.out.println("d = " + Double.doubleToLongBits(d));
System.out.println("i = " + Integer.toBinaryString(i));
System.out.println("i = " + Integer.toHexString(i));
System.out.println("i = " + Integer.toOctalString(i));
}
}
```

Методы **parseint()** и конструкторы классов требуют обработки исключений, поэтому в листинг 4.1 вставлен блок **try{} catch(){}**. Обработку исключительных ситуаций мы разберем в главе 16.

Класс Boolean

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

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

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

Класс Character

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

Статический метод:

```
digit(char ch, in radix)
```

…переводит цифру ch системы счисления с основанием radix в ее числовое значение типа int.

Статический метод:

```
forDigit(int digit, int radix)
```

…производит обратное преобразование целого числа digit в соответствующую цифру (тип char) в системе счисления с основанием radix.

Основание системы счисления должно находиться в диапазоне от Character.MIN_RADIX до Character.MAX_RADIX.

Метод **tostring()** переводит символ, содержащийся в классе, в строку с тем же символом.

Статические методы **toLowerCase()**, **touppercase()**, **toTitieCase()** возвращают символ, содержащийся в классе, в указанном регистре. Последний из этих методов предназначен для правильного перевода в верхний регистр четырех кодов Unicode, не выражающихся одним символом.

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

- **isDef ined()** – выясняет, определен ли символ в кодировке Unicode;
- **isDigit()** – проверяет, является ли символ цифрой Unicode;
- **isidentifierignorable()** – выясняет, нельзя ли использовать символ в идентификаторах;
- **isisocontroi()** – определяет, является ли символ управляющим;
- **isJavaidentifierPart()** – выясняет, можно ли использовать символ в идентификаторах;
- **isjavaidentifierstart()** – определяет, может ли символ начинать идентификатор;
- **isLetter()** – проверяет, является ли символ буквой Java;
- **IsLetterOrDigit()** – Проверяет, является ли символ буквой или цифрой Unicode;
- **isLowerCase()** – определяет, записан ли символ в нижнем регистре;
- **isSpaceChar()** – выясняет, является ли символ пробелом в смысле Unicode;
- **isTitieCase()** – проверяет, является ли символ титульным;
- **isUnicodeIdentifierPart()** – выясняет, можно ли использовать символ в именах Unicode;
- **isunicodeidentifierstart()** – проверяет, является ли символ буквой Unicode;
- **isUpperCase()** – проверяет, записан ли символ в верхнем регистре;
- **isWhitespace()** – выясняет, является ли символ пробельным.

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

Листинг 4.2 демонстрирует использование этих методов, а на рис. 4.3 показан вывод этой программы.

Листинг 4.2. Методы класса Character в программе CharacterTest.

```
class CharacterTest{
public static void main(String[] args){
char ch = '9';
Character cl = new Character(ch);
System.out.println("ch = " + ch);
System.out.println("cl.charValue() = " +
c1.charValue());
System.out.println("number of 'A' = " +
Character.digit('A', 16}};
System.out.println("digit for 12 = " +
Character.forDigit(12, 16}};
System.out.printlnC'cl = " + cl.toString());
System.out.println("ch isDefined? " +
Character.isDefined(ch));
System.out.println("ch isDigit? " +
Character.isDigit(ch));
System.out.println("ch isldentifierlgnorable? " +
Character.isldentifierlgnorable(ch));
System.out.println("ch isISOControl? " +
Character.isISOControl(ch));
System.out.println("ch isJavaldentifierPart? " +
Character.isJavaldentifierPart(ch));
System.out.println("ch isJavaldentifierStart? " +
Character.isJavaldentifierStart(ch));
System.out.println("ch isLetter? " +
Character.isLetter(ch));
System.out.println("ch isLetterOrDigit? " +
Character.isLetterOrDigit(ch));
System.out.println("ch isLowerCase? " +
Character.isLowerCase(ch));
System.out.println("ch isSpaceChar? " +
Character.isSpaceChar(ch));
System.out.println("ch isTitleCase? " +
Character.isTitleCase(ch));
System.out.println("ch isUnicodeldentifierPart? " +
Character.isUnicodeldentifierPart(ch));
System.out.println("ch isUnicodeldentifierStart? " +
Character.isUnicodeldentifierStart(ch));
System.out.println("ch isUpperCase? " +
Character.isUpperCase(ch));
System.out.println("ch isWhitespace? " +
Character.isWhitespace(ch)); } }
```

В класс Character вложены классы **Subset** и **UnicodeBlock**, причем класс Unicode и еще один класс, **inputSubset**, являются расширениями класса Subset, как это видно на рис. 4.1. Объекты этого класса содержат подмножества Unicode.

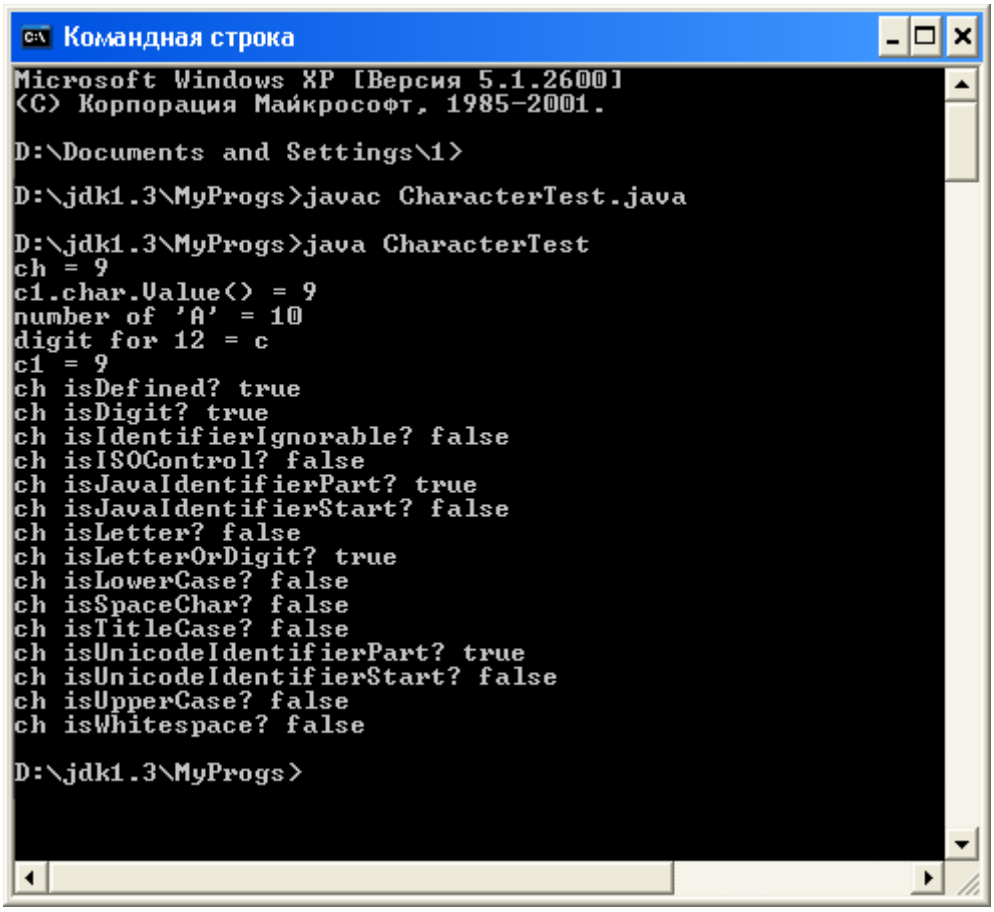

Рис. 4.3. Методы класса Character в программе CharacterTest

Вместе с классами-оболочками удобно рассмотреть два класса для работы со сколь угодно большими числами.

Класс Biglnteger

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

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

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

Перед выполнением операции числа выравниваются по длине распространением знакового разряда.

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

Две константы – ZERO и ONE – моделируют нуль и единицу в операциях с объектами класса Biglnteger.

Метод **toByteArray()** преобразует объект в массив байтов.

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

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

В листинге 4.3 приведены примеры использования данных методов, а рис. 4.4 показывает результаты выполнения этого листинга.

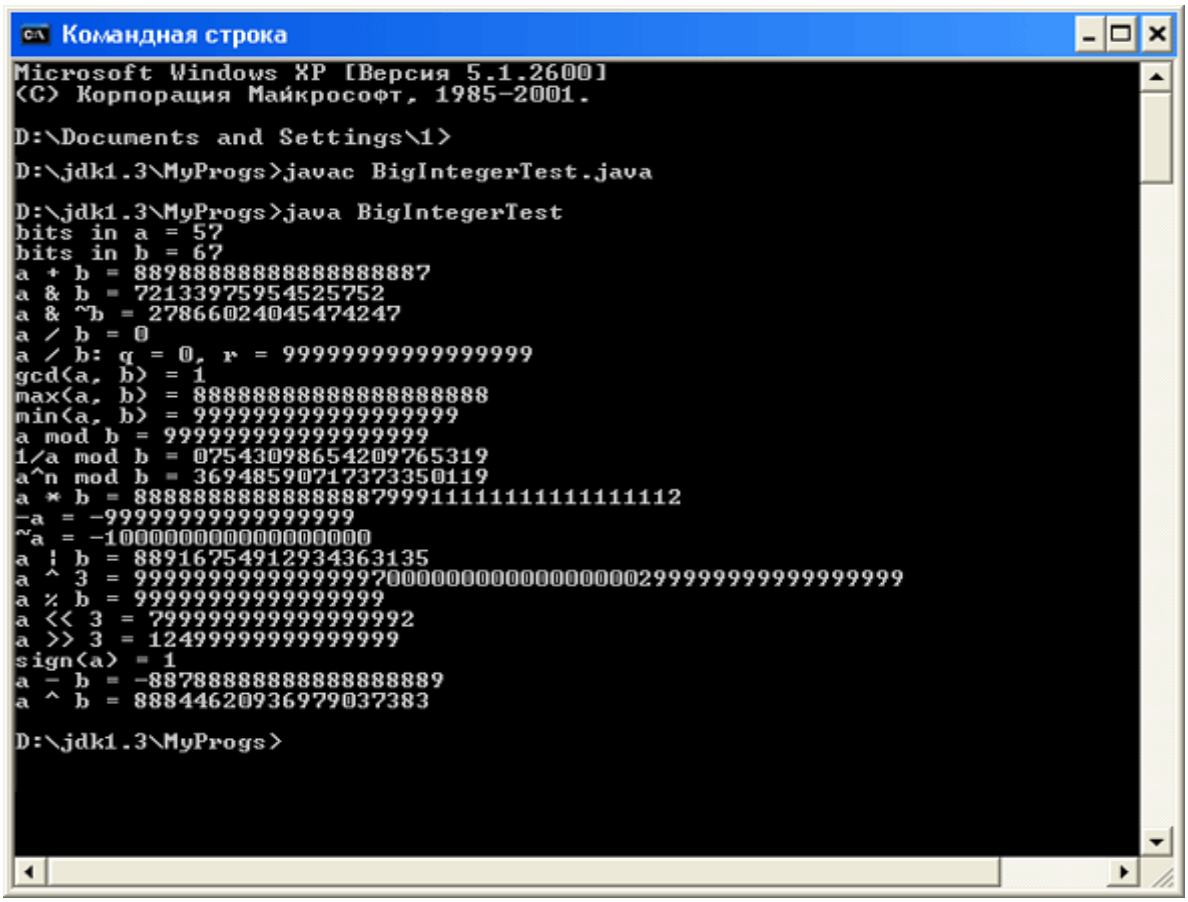

Рис. 4.4. Методы класса BigInteger в программе BigIntegerTest

Листинг 4.3. Методы класса BigInteger в программе BigIntegerTest.

```
import Java.math.BigInteger;
class BigIntegerTest{
public static void main(String[] args){
BigInteger a = new BigInteger("99999999999999999");
BigInteger b = new BigInteger("88888888888888888888");
System.out.println("bits in a = " + a.bitLength());
System.out.println("bits in b = " + b.bitLengthO);
System.out.println("a + b = " + a.add(b));
System.out.println("a & b = " + a.and(b));
System.out.println("a & ~b = " + a.andNot(b));
System.out.println("a / b = " + a.divide(b));
BigInteger[] r = a.divideAndRemainder(b);
System.out.println("a / b: q = " + r[0] + ", r = " + r[l]);
System.out.println("gcd(a, b) = " + a.gcd(b));
System.out.println("max(a, b) = " + a.max(b));
System.out.printin("min(a, b) = " + a.min(b));
System.out.println("a mod b = " + a.mod(b));
System.out.println("I/a mod b = " + a.modInverse(b));
System.out.println("алп mod b = " + a.modPow(a, b));
System.out.println("a * b = " + a.multiply(b));
System.out.println("-a = " + a.negate());
System, out .println ("~a = " + a.not());
System.out.println("a | b = " + a.or(b));
System.out.println("a л 3 = " + a.pow(3));
System.out.println("a % b = " + a.remainder(b));
System.out.println("a << 3 = " + a.SHIFTLeft(3)};
System.out.println("a >> 3 = " + a.SHIFTRight(3));
System.out.println("sign(a) = " + a.signum());
System.out.println("a - b = " + a.subtract(b));
System.out.println("a ^ b = " + a.xor(b));
}
}
```

Обратите внимание на то, что в программу листинга 4.3 надо импортировать пакет Java.math.

Класс Big Decimal

Класс BigDecimal расположен В пакете java.math.

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

Например, для числа 76.34862 будет храниться мантисса 7 634 862 в объекте класса BigInteger, и порядок 5 как целое число типа int. Таким образом, мантисса может содержать любое количество цифр, а порядок ограничен значением константы integer.MAX_VALUE. Результат операции над объектами класса BigDecimal округляется по одному из восьми правил, определяемых следующими статическими целыми константами:

- **ROUND_CEILING** – округление в сторону большего целого;
- **ROUND_DOWN** – округление к нулю, к меньшему по модулю целому значению;
- **ROUND_FLOOR** – округление к меньшему целому;
- **ROUND_HALF_DOWN** – округление к ближайшему целому, среднее значение округляется к меньшему целому;
- **ROUND_HALF_EVEN** – округление к ближайшему целому, среднее значение округляется к четному числу;
- **ROOND_HALF_UP** – округление к ближайшему целому, среднее значение округляется к большему целому;

- **ROUND_UNNECESSARY** – предполагается, что результат будет целым, и округление не понадобится;
- **ROUND_UP** – округление от нуля, к большему по модулю целому значению.

В классе BigDecimal четыре конструктора:

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

При использовании третьего из перечисленных конструкторов возникает неприятная особенность, отмеченная в документации. Поскольку вещественное число при переводе в двоичную форму представляется, как правило, бесконечной двоичной дробью, то при создании объекта, например, BigDecimal(0.1), мантисса, хранящаяся в объекте, окажется очень большой. Она показана на рис. 4.5. Но при создании такого же объекта четвертым конструктором, BigDecimal ("0.1"), мантисса будет равна просто 1.

В Классе переопределены методы **doubleValue()**, **floatValue()**, **intValue()**, **longValue()**.

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

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

Листинг 4.4 показывает примеры использования этих методов, а рис. 4.5 – вывод результатов.

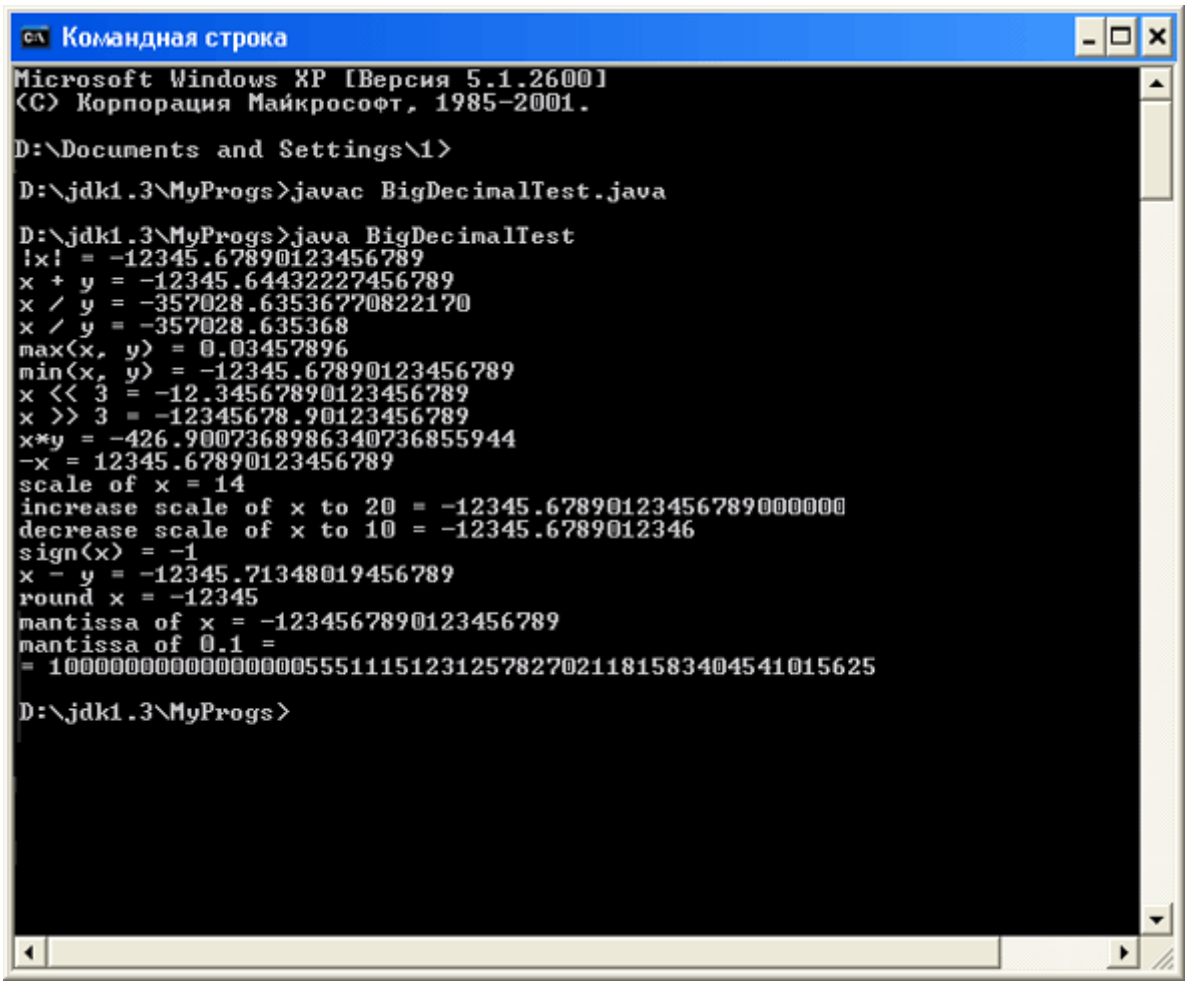

Рис. 4.5. Методы класса BigDecimal в программе BigDecimalTest

Листинг 4.4. Методы класса BigDecimal В программе BigDecimalTest.

```
import java.math.*;
class BigDecimalTest{
public static void main,(String [] args) {
BigDecimal x = new BigDecimal("-12345.67890123456789");
BigDecimal y = new BigDecimal("345.7896e-4");
BigDecimal z = new BigDecimal(new Biglnteger("123456789"),8);
System.out.println("|x| = " + x.abs());
System.out.println("x + y = " + x.add(y));
System.out.println("x / y = " + x.divide(y, BigDecimal.ROUND__DOWN));
System.out.println("x / y = " +
x.divide(y, 6, BigDecimal.ROUND_HALF_EVEN));
System.out.println("max(x, y) = " + x.max(y));
System.out.println("min(x, y) = " + x.min(y));
System.out.println("x << 3 = " * x.movePointLeft(3));
System.out.println("x >> 3 = " + x.mpvePQintRight(3));
System.out.println("x * y = " + x.multiply(y));
System.out.println("-x = " + x.negate());
System.out.println("scale of x = " + x.scale());
System.out.println("increase scale of x to 20 = " + x.setScale(20));
System.out.println("decrease scale of x to 10 = " +
x.setScale (10, BigDecimal.ROUND_HALF__UP));
System.out.println("sign(x) = " + x.signum());
System.out.println("x - y = " + x.subtract(y)};
System.out.println("round x = " + x.toBiglnteger());
System.out.println("mantissa of x = " + x.unscaledValue());
System.out.println("mantissa of 0.1 =\n= " +
new BigDecimal(0.1).unscaledValue()); } }
```

Приведем еще один пример. Напишем простенький калькулятор, выполняющий четыре арифметических действий с числами любой величины. Он работает из командной строки. Программа представлена в листинге 4.5, а примеры использования калькулятора – на рис. 4.6.

Листинг 4.5. Простейший калькулятор.

```
import Java.math.*;
class Calc{
public static void main(String[] args){
if (args.length < 3){
System.err.println("Usage: Java Calc operand operator operand");
return;
}
BigDecimal a = new BigDecimal(args[0]);
BigDecimal b = new BigDecimal(args[2]);
switch (args[l].charAt(0)){
case '+': System.out.println(a.add(b)); break;
case '-': System.out.println(a.subtract(b)); break;
case '*': System.out.println(a.multiply(b)); break;
case '/': System.out.println(a.divide(b,
BigDecimal.ROUND_HALF_EVEN)); break;
default: System.out.println("Invalid operator");
}
}
}
```

Почему символ умножения – звездочка – заключен на рис. 4.6 в кавычки? "Юниксоидам" это понятно, а для других дадим краткое пояснение.

Это особенность операционной системы, а не языка Java. Введенную с клавиатуры строку вначале просматривает командная оболочка (**shell**) операционной системы, а звездочка для нее – указание подставить на это место все имена файлов из текущего каталога. Оболочка сделает это, и интерпретатор Java получит от нее длинную строку, в которой вместо звездочки стоят имена файлов через пробел.

Звездочка в кавычках понимается командной оболочкой как обычный символ. Командная оболочка снимает кавычки и передает интерпретатору Java то, что надо.

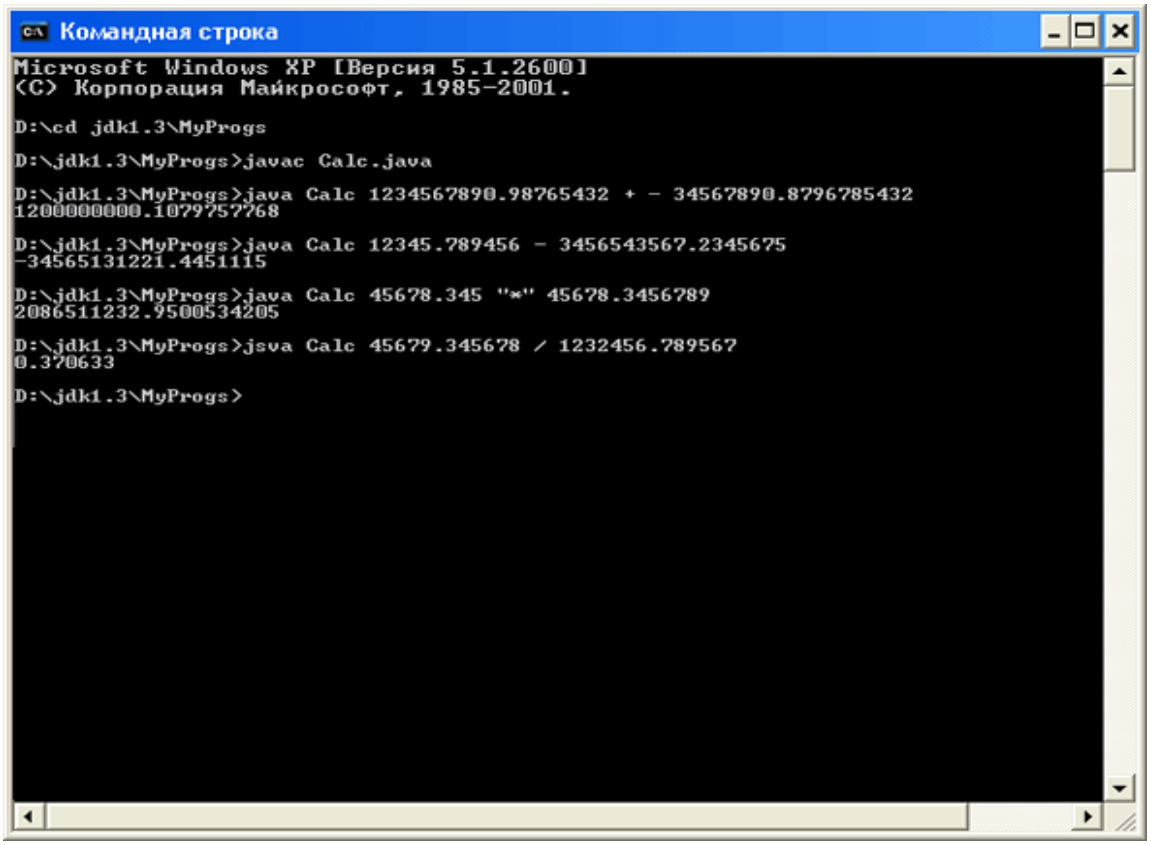

Рис. 4.6. Результаты работы калькулятора

Класс Class

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

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

В классе class нет конструкторов, экземпляр этого класса создается исполняющей системой Java во время загрузки класса и предоставляется методом **getciass()** класса object, например:

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

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

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

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

```
Class c2 = Java.lang.String.class;
```

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

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

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

Метод **getsuperciass()** возвращает суперкласс объекта ссылочного типа, **getPackage()** – пакет, **getModifiers()** – модификаторы класса В битовой форме. Модификаторы можно затем расшифровать методами класса Modifier из пакета Java.lang.reflect. Листинг 4.6 показывает применение этих методов, а рис. 4.7 – вывод результатов.

Листинг 4.6 tМетоды класса Class в программе ClassTest.

```
import java.lang.reflect.*;
class ClassTest{
public static void main(String[] args)(
Class c = null, c1 = null, c2 = null;
Field[] fld = null;
String s = "Some string";
c = s.getClass();
try{
cl = Class.forName("Java.lang.String"); // Старый стиль
c2 = Java.lang.String.class; // Новый стиль
if (!c1.isPrimitive())
fid = cl.getDeclaredFields(); // Все поля класса String
}catch(Exception e){}
System.out.println("Class c: " + c);
System.out.println("Class c1: " + c1);
System,out.println("Class c2: " + c2);
System.out.printlnt"Superclass c: " + c.getSuperclass());
System.out.println("Package c: " + c.getPackageO);
System.out.printlnf"Modifiers c: " + c.getModifiers());
for(int i = 0; i < fid.length; i++)
System.out.println(fld[i]);
}
}
```

Методы, возвращающие свойства классов, вызывают исключительные ситуации, требующие обработки. Поэтому в программу введен блок **try{} catch() {}**. Рассмотрение обработки исключительных ситуаций мы откладываем до главы 16.

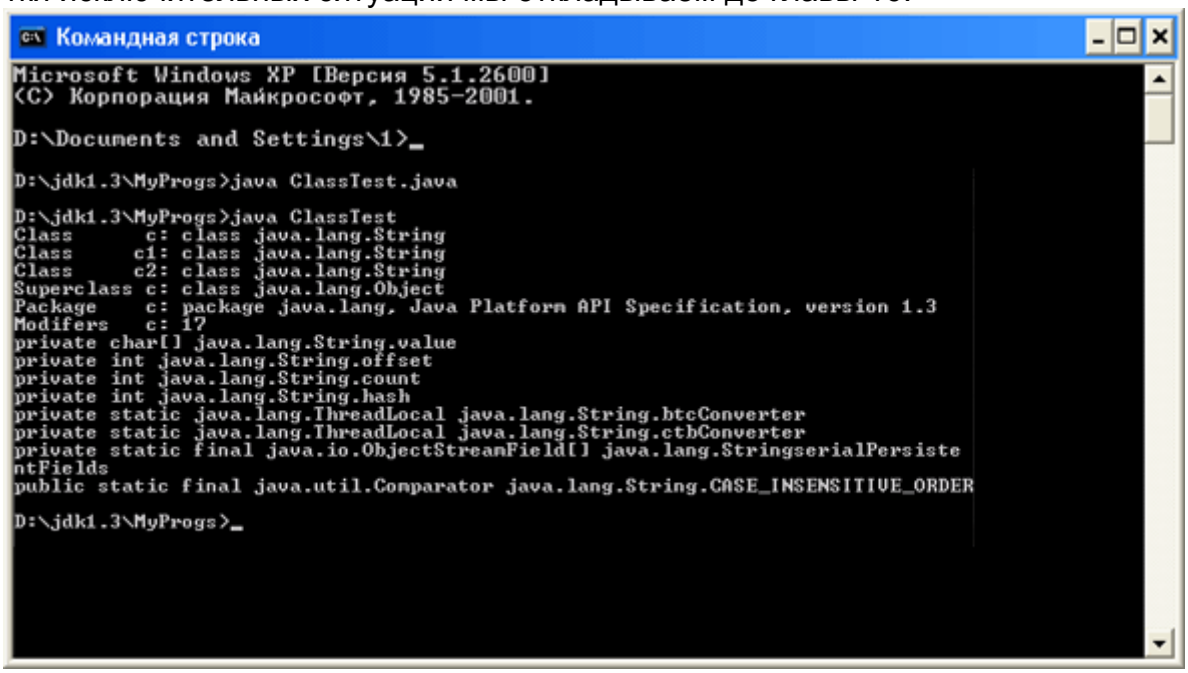

Рис. 4.7. Методы класса Class в программе ClassTest

Работа со строками

- ### Класс String
  Очень большое место в обработке информации занимает работа с текстами. Как и многое другое, текстовые строки в языке Java являются объектами. Они представляются экземплярами класса string или класса stringBuffer.
- ### Как создать строку. Сцепление строк.
  Самый простой способ создать строку – это организовать ссылку типа string на строку-константу: | String si = "Это строка."; | Если константа длинная, можно записать ее в нескольких строках текстового редактора, связывая их операцией сцепления: | String s2 = "Это длинная строка, " + | "записанная в двух строках исходного текста";
- ### Манипуляции строками. Как узнать длину строки. Как выбрать подстроку.
  В классе string есть множество методов для работы со строками. Посмотрим, что они позволяют делать. | Как узнать длину строки | Для того чтобы узнать длину строки, т. е. количество символов в ней, надо обратиться к методу length(): | String s = "Write once, run anywhere."; | int len = s.length{};
- ### Как выбрать символы из строки
  Выбрать символ с индексом ind (индекс первого символа равен нулю) можно методом charAt(int ind). Если индекс ind отрицателен или не меньше чем длина строки, возникает исключительная ситуация. Например, после определения: | char ch = s.charAt(3); | …переменная ch будет иметь значение 't'.
- ### Как сравнить строки
  Операция сравнения == сопоставляет только ссылки на строки. Она выясняет, указывают ли ссылки на одну и ту же строку. Например, для строк: | String s1 = "Какая-то строка"; | String s2 = "Другая-строка"; | …сравнение s1 == s2 дает в результате false.
- ### Как найти символ в строке
  Поиск всегда ведется с учетом регистра букв. | Первое появление символа ch в данной строке this можно отследить методом indexOf(int ch), возвращающим индекс этого символа в строке или -1, если символа ch в строке this нет. | Например, "Молоко", indexOf('0') выдаст в результате 1.
- ### Как найти подстроку
  Поиск всегда ведется с учетом регистра букв. | Первое вхождение подстроки sub в данную строку this отыскивает метод indexof (String sub). Он возвращает индекс первого символа первого вхождения подстроки sub в строку или -1, если подстрока sub не входит в строку this.
- ### Как изменить регистр букв. Как заменить отдельный символ. Как убрать пробелы в начале и конце строки.

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

- **Как преобразовать данные другого типа в строку**
  В языке Java принято соглашение – каждый класс отвечает за преобразование других типов в тип этого класса и должен содержать нужные для этого методы. | Класс string содержит восемь статических методов valueof (type elem) преобразования в строку примитивных типов boolean, char, int, long, float, double, массива char[], и просто объекта типа object.
- **Класс StringBuffer**
  Объекты класса StringBuffer – это строки переменной длины. Только что созданный объект имеет буфер определенной емкости (capacity), по умолчанию достаточной для хранения 16 символов. Емкость можно задать в конструкторе объекта.
- **Класс StringTokenizer**
  Класс StringTokenizer из пакета java.utii небольшой, в нем три конструктора и шесть методов. | Первый конструктор StringTokenizer (String str) создает объект, готовый разбить строку str на слова, разделенные пробелами, символами табуляций '\t', перевода строки '\n' и возврата каретки '\r'.

Класс String

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

Поначалу это необычно и кажется слишком громоздким, но, привыкнув, вы оцените удобство работы с,классами, а не с массивами символов.

Конечно, возможно занести текст в массив символов типа char или даже в массив байтов типа byte, но тогда вы не сможете использовать готовые методы работы с текстовыми строками.

Зачем в язык введены два класса для хранения строк? В объектах класса string хранятся строки-константы неизменной длины и содержания, так сказать, отлитые в бронзе. Это значительно ускоряет обработку строк и позволяет экономить память, разделяя строку между объектами, использующими ее. Длину строк, хранящихся в объектах класса stringBuffer, можно менять, вставляя и добавляя строки и символы, удаляя подстроки или сцепляя несколько строк в одну строку. Во многих случаях, когда надо изменить длину строки типа string, компилятор Java неявно преобразует ее к типу stringBuffer, меняет длину, потом преобразует обратно в тип string. Например, следующее действие:

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

…компилятор выполнит так:

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

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

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

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

Как создать строку. Сцепление строк.

Самый простой способ создать строку – это организовать ссылку типа string на строку-константу:

```
String si = "Это строка.";
```

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

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

Замечание

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

Самый правильный способ создать объект с точки зрения ООП – это вызвать его конструктор в операции **new**. Класс string предоставляет вам девять конструкторов:

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

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

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

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

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

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

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

- В MS-DOS применяется кодировка CP866.
- В UNIX обычно применяется кодировка KOI8-R.
- На компьютерах Apple Macintosh используется кодировка MacCyrillic.

- Есть еще и международная кодировка кириллицы ISO8859-5;

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

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

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

Листинг 5.1 показывает различные случаи записи кириллического текста. В нем создаются три массива байто'в, содержащих слово "Россия" в трех кодировках.

- Массив byteCP1251 содержит слово "Россия" в кодировке CP1251.
- Массив byteCP866 содержит слово "Россия" в кодировке CP866.
- Массив byteKOI8R содержит слово "Россия" в кодировке KOI8-R.

Из каждого массива создаются по три строки с использованием трех кодовых таблиц.

Кроме того, из массива символов c[] создается строка s1, из массива бай-тов, записанного в кодировке CP866, создается строка s2. Наконец, создается ссылка зз на строку-константу.

Листинг 5.1. Создание кириллических строк.

```
class StringTest{
public static void main(String[] args){
String winLikeWin = null, winLikeDOS = null, winLikeUNIX = null;
String dosLikeWin = null, dosLikeDOS = null, dosLikeUNIX = null;
String unixLikeWin = null, unixLikeDOS = null, unixLikeUNIX = null;
String msg = null;
byte[] byteCp!251 = {
(byte)0xD0, (byte)0xEE, (byte)0xFl,
(byte)0xFl, (byte)0xES, (byte)0xFF
};
byte[] byteCp866 = {
(byte)0x90, (byte)0xAE, (byte)0xE1,
(byte)0xEl, (byte)0xA8, (byte)0xEF
};
byte[] byteKOISR = (
(byte)0xF2, (byte)0xCF, (byte)0xD3,
(byte)0xD3, (byte)0xC9, (byte)0xDl
};
char[] c = {'P', 'o', 'c', 'c', 'и', 'я'};
String s1 = new String(c);
String s2 = new String(byteCp866); // Для консоли MS Windows
String s3 = "Россия";
System.out.println();
try{
// Сообщение в Cp866 для вывода на консоль MS Windows.
msg = new String("\"Россия\" в ".getBytes("Cp866"), "Cpl251");
winLikeWin = new String(byteCp1251, "Cpl251"); //Правильно
winLikeDOS = new String(byteCpl251,: "Cp866");
winLikeUNIX - new String(byteCp1251, "KOI8-R");
dosLikeWin = new String(byteCp866, "Cpl251"); // Для консоли
dosLikeDOS = new String(byteCp866, "Cp866"); // Правильно
dosLikeUNIX = new String(byteCp866, "KOI8-R");
unixLikeWin = new String(byteKOISR, "Cpl251");
unixLikeDOS = new String(byteKOISR, "Cp866");
unixLikeUNIX = new String(byteKOISR, "KOI8-R"); // Правильно
System.out.print(msg + "Cpl251: ");
System.out.write(byteCp1251);
System.out.println();
System.out.print(msg + "Cp866: ");
System, out.write (byteCp866};
```

```
System.out.println();
System.out.print(msg + "KOI8-R: ");
System.out.write(byteKOI8R);
{catch(Exception e)(
e.printStackTrace();
}
System.out.println();
System.out.println();
System.out.println(msg + "char array: " + s1);
System.out.println(msg + "default encoding: " + s2);
System.out.println(msg + "string constant: " + s3);
System.out.println();
System.out.println(msg + "Cp1251 › Cp1251: " + winLikeWin);
System.out.println(msg + "Cp1251 › Cp866: " + winLikeDOS);
System.out.println(msg + "Cp1251 › KOI8-R: " + winLikeUNIX);
System.out.println(msg + "Cp866 › Cp1251: " + dosLikeWin);
System.out.println(msg + "Cp866 › Cp866: " + dosLikeDOS);
System.out.println(msg + "Cp866 › KOI8-R: " + dosLikeUNIX);
System.out.println(msg + "KOI8-R › Cp1251: " + unixLikeWin);
System.out.println(msg + "KOI8-R › Cp866: " + unixLikeDOS);
System.out.println(msg + "KOI8-R › KOI8-R: " + unixLikeUNIX);
}
}
```

Все эти данные выводятся на консоль MS Windows 2000, как показано на рис. 5.1.

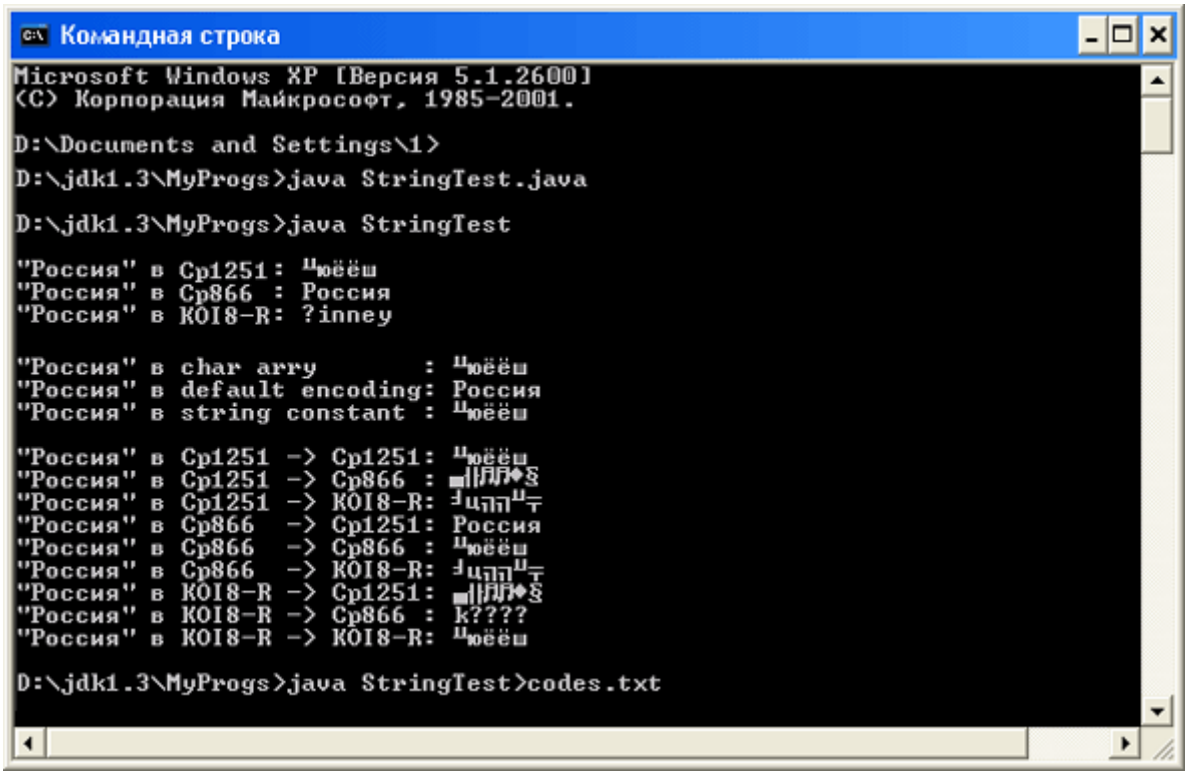

Рис. 5.1. Вывод кириллической строки на консоль MS Windows 2000

В первые три строки консоли выводятся массивы байтов byteCP1251, byteCP866 и byteKOI8R без преобразования в Unicode. Это выполняется методом write() класса FilterOutputStream из пакета java.io.

В следующие три строки консоли выведены строки Java, полученные из массива символов c[], массива byteCP866 и строки-константы.

Следующие строки консоли содержат преобразованные массивы.

Вы видите, что на консоль правильно выводится только массив в кодировке CP866, записанный в строку с использованием кодовой таблицы CP1251.

В чем дело? Здесь свой вклад в проблему русификации вносит вывод потока символов на консоль или в файл.

Как уже упоминалось в главе 1, в консольное окно **Command Prompt** операционной системы MS Windows текст выводится в кодировке CP866.

Для того чтобы учесть это, слова "\"Россия\" в" преобразованы в массив байтов, содержащий символы в кодировке CP866, а затем переведены в строку msg.

В предпоследней строке рис. 5.1 сделано перенаправление вывода программы в файл codes.txt. В MS Windows 2000 вывод текста в файл происходит в кодировке CP1251. На рис. 5.2 показано содержимое файла codes.txt в окне программы Notepad.

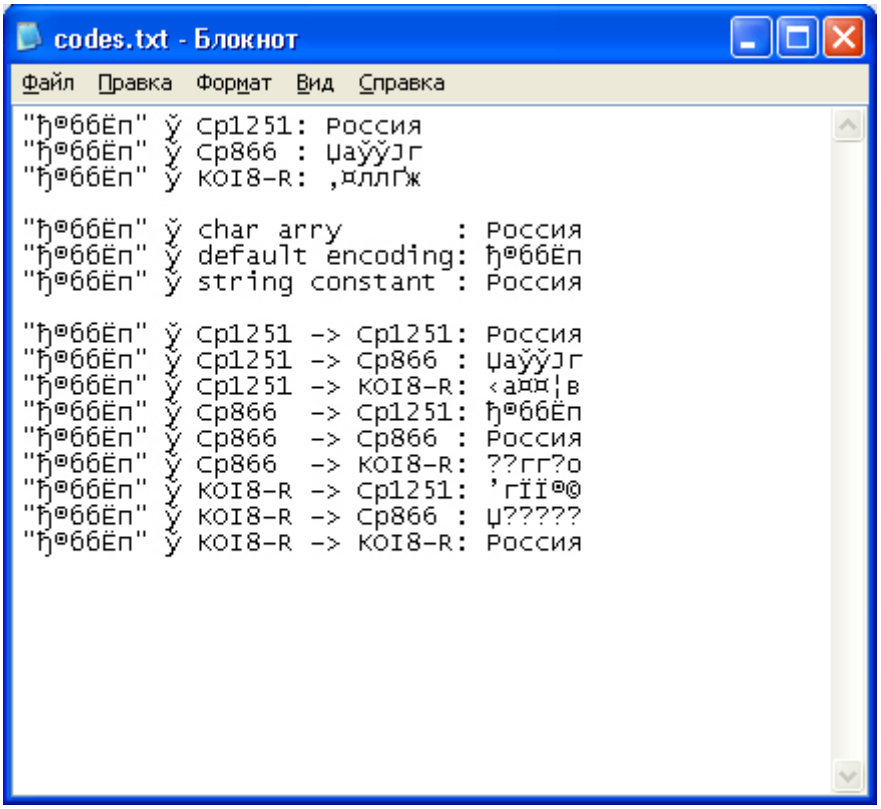

Рис. 5.2. Вывод кириллической строки в файл

Как видите, кириллица выглядит совсем по-другому. Правильные символы Unicode кириллицы получаются, если использовать ту же кодовую таблицу, в которой записан исходный массив байтов.

Вопросы русификации мы еще будем обсуждать в главах 9 и 18, а пока заметьте, что при создании строки из массива байтов лучше указывать ту же самую кириллическую кодировку, в которой записан массив. Тогда вы получите строку Java с правильными символами Unicode.

При выводе же строки на консоль, в окно, в файл или при передаче по сети лучше преобразовать строку Java с символами Unicode по правилам вывода в нужное место.

Еще один способ создать строку – это использовать два статических метода:

```
copyValueOf(chart] charArray) и
 copyValueOf(char[] charArray, int offset, int length).
```

Они создают строку по заданному массиву символов и возвращают ее в качестве результата своей работы. Например, после выполнения следующего фрагмента программы:

```
chart] c = ('С', 'и', 'м', 'в', 'о', 'л', 'ь', 'н', 'ы', 'й'};
String s1 = String.copyValueOf(c);
String s2 = String.copyValueOf(c, 3, 7);
```

…получим в объекте s1 строку " Символьный ", а в объекте s2 – строку " вольный ".

Сцепление строк

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

```
String attention = "Внимание: ";
String s = attention + "неизвестный символ";
Вторая операция – присваивание += – применяется к переменным в левой части:
attention += s;
```

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

Манипуляции строками. Как узнать длину строки. Как выбрать подстроку.

В классе string есть множество методов для работы со строками. Посмотрим, что они позволяют делать.

Как узнать длину строки

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

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

…или еще проще:

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

…поскольку строка-константа – полноценный объект класса string. Заметьте, что строка – это не массив, у нее нет поля length.

Внимательный читатель, изучивший рис. 4.7, готов со мной не согласиться. Ну, что же, действительно, символы хранятся в массиве, но он закрыт, как и все поля класса string.

Как выбрать подстроку

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

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

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

Например, после выполнения:

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

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

Как выбрать символы из строки

Выбрать символ с индексом ind (индекс первого символа равен нулю) можно методом**charAt(int ind)**. Если индекс ind отрицателен или не меньше чем длина строки, возникает исключительная ситуация. Например, после определения:

```
char ch = s.charAt(3);
```

…переменная ch будет иметь значение 't'.

Все символы строки в виде массива символов можно получить методом **toCharArray()**, возвращающим массив символов.

Если же надо включить в массив символов dst, начиная с индекса ind массива подстроку от индекса begin включительно до индекса end исключительно, то используйте метод**getChars(int begin, int end, char[] dst, int ind)** типа void.

В массив будет записано end – begin символов, которые займут элементы массива, начиная с индекса ind до индекса ind + (end -begin) -1.

Этот метод создает исключительную ситуацию в следующих случаях:

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

Например, после выполнения:

```
char[] ch = ('К', 'о', 'р', 'о', 'л', 'ь', ' ', 'л', 'е', 'т', 'а'};
"Пароль легко найти".getChars(2, 8, ch, 2);
```

…результат будет таков:

```
ch = ('К', 'о', 'р', 'о', 'л', 'ь ', ' ', 'л', 'е', 'т', 'а'};
```

Если надо получить массив байтов, содержащий все символы строки в байтовой кодировке ASCII, то используйте метод **getBytes()**.

Этот метод при переводе символов из Unicode в ASCII использует локальную кодовую таблицу.

Если же надо получить массив байтов не в локальной кодировке, а в какой-то другой, используйте метод **getBytes(String encoding)**.

Так сделано в листинге 5.1 при создании объекта msg. Строка "\'Тоссия в\"" перекодировалась в массив CP866-байтов для правильного вывода кириллицы в консольное окно **Command Prompt** операционной системы Windows 2000.

Как сравнить строки

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

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

…сравнение s1 == s2 дает в результате false.

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

Интересно, что если мы определим s2 так:

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

…то сравнение s1 == s2 даст в результате true, потому что компилятор создаст только один экземпляр константы "Какая-то строка" и направит на него все ссылки.

Вы, разумеется, хотите сравнивать не ссылки, а содержимое строк. Для этого есть несколько методов.

Логический метод **equals (object obj)**, переопределенный из класса object, возвращает true, если аргумент obj не равен null, является объектом класса string, и строка, содержащаяся в нем, полностью идентична данной строке вплоть до совпадения регистра букв. В остальных случаях возвращается значение false.

Логический метод **equalsIgnoreCase(object obj)** работает так же, но одинаковые буквы, записанные в разных регистрах, считаются совпадающими.

Например, s2.equals("другая строка") даст в результате false, а s2.equalsIgnoreCase("другая строка") возвратит true.

Метод compareTo(string str) возвращает целое число типа int, вычисленное по следующим правилам:

1. Сравниваются символы данной строки this и строки str с одинаковым индексом, пока не встретятся различные символы с индексом, допустим k, или пока одна из строк не закончится.
2. В первом случае возвращается значение this.charAt(k) – str.charAt(k), т. е. разность кодировок Unicode первйх несовпадающих символов.
3. Во втором случае возвращается значение this.length() – str.length(), т. е. разность длин строк.
4. Если строки совпадают, возвращается 0.

Если значение str равно null, возникает исключительная ситуация.

Нуль возвращается в той же ситуации, в которой метод **equals()** возвращает true.

Метод **compareToignoreCase(string str)** производит сравнение без учета регистра букв, точнее говоря, выполняется метод:

```
this.toUpperCase().toLowerCase().compareTo(
str.toUpperCase().toLowerCase());
```

Еще один метод– **compareTo (Object obj)** создает исключительную ситуацию, если obj не является строкой. В остальном он работает как метод **compareTo(String str)**.

Эти методы не учитывают алфавитное расположение символов в локальной кодировке.

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

Если вас такое расположение не устраивает, задайте свое размещение букв с помощью класса **RuleBasedCollator** из пакета java.text.

Сравнить подстроку данной строки this с подстрокой той же длины len другой строки str можно логическим методом:

```
regionMatches(int indl, String str, int ind2, int len)
```

Здесь ind1 – индекс начала подстроки данной строки this, ind2 – индекс начала подстроки другой строки str. Результат false получается в следующих случаях:

- хотя бы один из индексов ind1 или ind2 отрицателен;
- хотя бы одно из ind1 + len или ind2 + len больше длины соответствующей строки;
- хотя бы одна пара символов не совпадает.

Этот метод различает символы, записанные в разных регистрах. Если надо сравнивать подстроки без учета регистров букв, то используйте логический метод:

```
regionMatches(boolean flag, int indl, String str, int ind2, int len)
```

Если первый параметр flag равен true, то регистр букв при сравнении подстрок не учитывается, если false – учитывается.

Как найти символ в строке

Поиск всегда ведется с учетом регистра букв.

Первое появление символа ch в данной строке this можно отследить методом **indexOf(int ch)**, возвращающим индекс этого символа в строке или -1, если символа ch в строке this нет.

Например, "**Молоко**"**, indexOf('0')** выдаст в результате 1.

Конечно, этот метод выполняет в цикле последовательные сравнения **this.charAt(k++> == ch**, пока не получит значение true.

Второе и следующие появления символа ch в данной строке this можно отследить методом **indexOf(int ch, int ind)**.

Этот метод начинает поиск символа ch с индекса ind. Если ind < 0, то поиск идет с начала строки, если ind больше длины строки, то символ не ищется, т. е. возвращается -1.

Например, "**молоко**".**indexof('0', indexof ('0') + 1)** даст в результате 3.

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

Например, "**Молоко**".**lastindexof('0')** даст в результате 5.

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

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

Как найти подстроку

Поиск всегда ведется с учетом регистра букв.

Первое вхождение подстроки sub в данную строку this отыскивает метод **indexof (String sub)**. Он возвращает индекс первого символа первого вхождения подстроки sub в строку или -1, если подстрока sub не входит в строку this. Например, " Раскраска ".indexof ("рас") даст в результате 4.

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

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

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

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

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

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

Например:

```
if (fileName.endsWith(". Java"))
```

…отследит имена файлов с исходными текстами Java.

Перечисленные выше методы создают исключительную ситуацию, если sub == null.

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

Как изменить регистр букв. Как заменить отдельный символ. Как убрать пробелы в начале и конце строки.

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

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

При этом используется локальная кодовая таблица по умолчанию. Если нужна другая локаль, то применяются методы **toLowerCase(Locale loc)** и **toUpperCase(Locale loc)**.

Как заменить отдельный символ

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

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

Регистр букв при замене учитывается.

Как убрать пробелы в начале и конце строки

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

Как преобразовать данные другого типа в строку

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

Класс string содержит восемь статических методов **valueof (type elem)** преобразования в строку примитивных типов boolean, char, int, long, float, double, массива char[], и просто объекта типа object.

Девятый метод **valueof(char[] ch, int offset, int len)** преобразует в строку подмассив массива ch, начинающийся с индекса offset и имеющий len элементов.

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

Еще один простой способ – сцепить значение elem какого-либо типа с пустой строкой: **""** + elem. При этом неявно вызывается метод **elem .toString ()**.

Класс StringBuffer

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

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

В любое время емкость буфера можно увеличить, обратившись к методу**ensureCapacity(int minCapacity)**.

Этот метод изменит емкость, только если minCapacity будет больше длины хранящейся в объекте строки. Емкость будет увеличена по следующему правилу. Пусть емкость буфера равна N. Тогда новая емкость будет равна:

```
Max(2 * N + 2, minCapacity)
```

Таким образом, емкость буфера нельзя увеличить менее чем вдвое.

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

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

Если число newLength окажется отрицательным, возникнет исключительная ситуация.

Совет

Будьте осторожны, устанавливая новую длину объекта.

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

Создать объект класса **stringBuffer** можно только конструкторами.

Конструкторы

В классе stringBuffer три конструктора:

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

Как добавить подстроку

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

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

Шесть методов **append (type elem)** добавляют примитивные типы boolean, char, int, long, float, double, преобразованные в строку.

Два метода присоединяют к строке массив str и подмассив sub символов, преобразованные в строку:

```
append (char [] str) и
append (char [.], sub, int offset, int len).
```

Десятый метод добавляет просто объект **append (Object obj)**. Перед этим объект obj преобразуется в строку своим методом **tostring ()**.

Как вставить подстроку

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

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

Например, после выполнения:

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

…получим:

```
s== "Это небольшая строка".
```

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

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

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

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

Десятый метод вставляет просто объект:

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

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

Как удалить подстроку

Метод **delete (int begin, int end)** удаляет из строки символы, начиная с индекса begin включительно до индекса end исключительно, если end больше длины строки, то до конца строки.

Например, после выполнения:

```
String s = new StringBuffer("Это небольшая строка"),
delete(4, 6).toString();
```

…получим:

```
s == "Это большая строка".
```

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

Если begin == end, удаление не происходит.

Как удалить символ

Метод **deleteCharAt (int ind)** удаляет символ с указанным индексом ind. Длина строки уменьшается на единицу.

Если индекс ind отрицателен или больше длины строки, возникает исключительная ситуация.

Как заменить подстроку

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

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

Разумеется, метод **replace ()** – это последовательное выполнение методов **delete ()** и**insert ()**.

Как перевернуть строку

Метод **reverse ()** меняет порядок расположения символов в строке на обратный порядок.

Например, после выполнения:

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

…получим:

```
s == "акортс яашьлобен отЭ".
```

Класс StringTokenizer

Класс StringTokenizer из пакета java.utii небольшой, в нем три конструктора и шесть методов.

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

Второй конструктор **StringTokenizer (String str. String delimeters)** задает разделители вторым параметром deiimeters, например:

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

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

Третий конструктор позволяет включить разделители в число слов:

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

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

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

В разборе строки на слова активно участвуют два метода:

- метод **nextToken ()** возвращает в виде строки следующее слово;
- логический метод **hasMoreTokens ()** возвращает true, если в строке еще есть слова, и false, если слов больше нет.

Третий метод **countTokens ()** возвращает число оставшихся слов.

Четвертый метод **nextToken(string newDeiimeters)** позволяет "на ходу" менять разделители. Следующее слово будет выделено по новым разделителям newDeiimeters; новые разделители действуют далее вместо старых разделителей, определенных в конструкторе или предыдущем методе nextToken ().

Оставшиеся два метода **nextEiement ()** и **hasMoreEiements ()** реализуют интерфейс Enumeration. Они просто обращаются к методам **nextToken ()** и **hasMoreTokens()**.

Схема очень проста (листинг 5.2).

Листинг 5.2. Разбиение строки на слова.

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

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

Заключение

Все методы представленных в этой главе классов написаны на языке Java. Их исходные тексты можно посмотреть, они входят в состав JDK. Эти очень полезное занятие. Просмотрев исходный текст, вы получаете полное представление о том, как работает метод.

В последних версиях JDK исходные тексты хранятся в упакованном архиватором jar файле src.jar, лежащем в корневом каталоге JDK, например, в каталоге D:\jdk 1.3.

Чтобы распаковать их, перейдите в каталог jdk 1.3:

```
D: > cd jdk1.3
```

…и вызовите архиватор jar следующим образом:

```
D:\jdk1.3 > jar – xf src.jar
```

В каталоге jdk1.3 появится подкаталог src, а в нем подкаталоги, соответствующие пакетам и подпакетам JDK, с исходными файлами.

Классы-коллекции

- ## Класс Vector
  В листинге 5.2 мы разобрали строку на слова. Как их сохранить для дальнейшей обработки? | До сих пор мы пользовались массивами. Они удобны, если необходимо быстро обработать однотипные элементы, например, просуммировать числа, найти наибольшее и наименьшее значение, отсортировать элементы.
- ## Класс Stack
  Класс stack из пакета java.utii. объединяет элементы в стек. | Стек (stack) реализует порядок работы с элементами подобно магазину винтовки– первым выстрелит патрон, положенный в магазин последним,– или подобно железнодорожному тупику – первым из тупика выйдет вагон, загнанный туда последним.
- ## Класс Hashtable
  Класс Hashtable расширяет абстрактный класс Dictionary. В объектах этого класса хранятся пары "ключ – значение". | Из таких пар "Фамилия И. О. – номер" состоит, например, телефонный справочник. | Еще один пример – анкета.
- ## Класс Properties
  Класс Properties расширяет класс Hashtabie. Он предназначен в основном для ввода и вывода пар свойств системы и их значений. Пары хранятся в виде строк типа string. В классе Properties два конструктора: | Properties () – создает пустой объект;
- ## Интерфейс Collection
  Интерфейс collection из пакета java.util описывает общие свойства коллекций List и set. Он содержит методы добавления и удаления элементов, проверки и преобразования элементов: | boolean add (Object obj) – добавляет элемент obj в конец коллекции;
- ## Интерфейс List
  Интерфейс List из пакета java.utii, расширяющий интерфейс collection, описывает методы работы с упорядоченными коллекциями. Иногда их называют последовательностями (sequence). Элементы такой коллекции пронумерованы, начиная от нуля, к ним можно обратиться по индексу.
- ## Интерфейс Set. Интерфейс SortedSet.
  Интерфейс set из пакета java.utii, расширяющий интерфейс Collection, описывает неупорядоченную коллекцию, не содержащую повторяющихся элементов. Это соответствует математическому понятию множества (set).
- ## Интерфейс Map
  Интерфейс Map из пакета java.utii описывает коллекцию, состоящую из пар "ключ – значение". У каждого ключа только одно значение, что соответствует математическому понятию однозначной функции или отображения (тар).
- ## Вложенный интерфейс Map.Entry. Интерфейс SortedMap.
  Этот интерфейс описывает методы работы с парами, полученными методом entrySet(): | методы getKey() и getvalue() позволяют получить ключ и значение пары; | метод setvaiue (object value) меняет значение в данной паре.
- ## Абстрактные классы-коллекции
  Эти классы лежат в пакете java.util. | Абстрактный класс AbstractGollection реализует интерфейс Collection, но оставляет нереализованными методы iterator (), size (). | Абстрактный класс AbstractList реализует интерфейс List, но оставляет нереализованным метод get(mt) и унаследованный метод size().
- ## Интерфейс Iterator
  В 70-80-х годах прошлого столетия, после того как была осознана важность правильной организации данных в определенную структуру, большое внимание уделялось изучению и построению различных структур данных: связанных списков, очередей, деков, стеков, деревьев, сетей.
- ## Интерфейс ListIterator
  Интерфейс Listiterator расширяет интерфейс iterator, обеспечивая перемещение по коллекции как в прямом, так и в обратном направлении. Он может быть реализован только в тех коллекциях, в которых есть понятия следующего и предыдущего элемента и где элементы пронумерованы.
- ## Классы, создающие списки. Двунаправленный список.
  Класс ArrayList полностью реализует интерфейс List и итератор типа iterator. Класс ArrayList очень похож на класс Vector,имеет тот же набор методов и может использоваться в тех же ситуациях. | В классе ArrayList три конструктора: | ArrayList () – создает пустой объект;
- ## Классы, создающие отображения. Упорядоченные отображения.
  Класс например полностью реализует интерфейс Map, а также итератор типа iterator. Класс HashMap очень похож на класс Hashtabie и может использоваться в тех же ситуациях. Он имеет тот же набор функций и такие же конструкторы: | HashMap () – создает пустой объект с показателем загруженности 0.75;
- ## Сравнение элементов коллекций
  Интерфейс Comparator описывает два метода сравнения: | int compare (Object obji, object obj2) – возвращает отрицательное число, если obj1 в каком-то смысле меньше obj2; нуль, если они считаются равными; положительное число, если obj1 больше obj2.
- ## Классы, создающие множества. Упорядоченные множества.
  Класс HashSet полностью реализует интерфейс set и итератор типа iterator. Класс Hashset используется в тех случаях, когда надо хранить только одну копию каждого элемента. | В классе HashSet четыре конструктора: | Hashset () – создает пустой объект с показателем загруженности 0.75;

- **Действия с коллекциями. Методы класса Collections.**
  Коллекции предназначены для хранения элементов в удобном для дальнейшей обработки виде. Очень часто обработка заключается в сортировке элементов и поиске нужного элемента. Эти и другие методы обработки собраны В Класс Collections.

Класс Vector

В листинге 5.2 мы разобрали строку на слова. Как их сохранить для дальнейшей обработки?

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

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

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

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

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

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

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

Как создать вектор

В классе четыре конструктора:

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

Как добавить элемент в вектор

Метод **add (Object element)** позволяет добавить элемент в конец вектора (то же делает старый метод **addElement (Object element)**.

Методом **add (int index, Object element)** или старым методом **insertElementAt (Object element, int index)** можно вставить элемент В указанное место index. Элемент, находившийся на этом месте, и все последующие элементы сдвигаются, их индексы увеличиваются на единицу.

Метод **addAll (Collection coll)** позволяет добавить в конец вектора все элементы коллекции coll.

Методом **addAll(int index, Collection coll)** возможно вставить в позицию index все элементы коллекции coll.

Как заменить элемент

Метод **set (int index, object element)** заменяет элемент, стоявший в векторе в позиции index, на элемент **element** (то же позволяет выполнить старый метод **setElementAt (Object element, int index))**.

Как узнать размер вектора

Количество элементов в векторе всегда можно узнать методом **size ()**. Метод **capacity ()**возвращает емкость вектора.

Логический метод **isEmpty ()** возвращает true, если в векторе нет ни одного элемента.

Как обратиться к элементу вектора

Обратиться к первому элементу вектора можно методом **firstElement ()**, к последнему – методом **lastEiement ()**, к любому элементу – методом **get (int index)** или старым методом**elementAt (int index).**

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

Получить все элементы вектора в виде массива типа **object[]** можно методами **toArray()** и**toArray (Object [] a)**. Второй метод заносит все элементы вектора в массив а, если в нем достаточно места.

Как узнать, есть ли элемент в векторе

Логический метод **contains (object element)** возвращает true, если элемент element находится в векторе.

Логический метод **containsAll (Collectionc)** возвращает true, если вектор содержит все элементы указанной коллекции.

Как узнать индекс элемента

Четыре метода позволяют отыскать позицию указанного элемента element:

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

Если элемент не найден, возвращается -1.

Как удалить элементы

Логический метод **remove (Object element)** удаляет из вектора первое вхождение указанного элемента element. Метод возвращает true, если элемент найден и удаление произведено.

Метод **remove (int index)** удаляет элемент из позиции index и возвращает его в качестве своего результата типа object.

Аналогичные действия позволяют выполнить старые методы типа void:

```
removeElement (Object element) и
removeElementAt (int index)
```

…не возвращающие результата.

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

Удалить из данного вектора все элементы коллекции coil возможно логическим Методом**removeAll(Collection coll)**.

Удалить последние элементы можно, просто урезав вектор методом **setSize(int newSize)**.

Удалить все элементы, кроме входящих в указанную коллекцию coil, разрешает логический метод **retainAll(Collection coll)**.

Удалить все элементы вектора можно методом **clear ()** или старым методом**removeAllElements ()** или обнулив размер вектора методом **setSize(O)**.

Листинг 6.1 расширяет листинг 5.2, обрабатывая выделенные из строки слова с помощью вектора.

Листинг 6.1. Работа с вектором.

```
Vector v = new Vector();
String s = "Строка, которую мы хотим разобрать на слова.";
StringTokenizer st = new StringTokenizer(s, " \t\n\r,.");
while (st.hasMoreTokens()){
// Получаем слово и заносим в вектор
v.add(st.nextToken()); // Добавляем в конец вектора
}
System.out.println(v.firstElement()); // Первый элемент
System.out.println(v.lastElement()); // Последний элемент
v.setSize(4); // Уменьшаем число элементов
v.add("собрать."); // Добавляем в конец
// укороченного вектора
v.set(3, "опять"); // Ставим в позицию 3
for (int i = 0; i < v.sizeO; i++) // Перебираем весь вектор
System.out.print(v.get(i) + " ");
System.out.println();
```

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

Второй пример коллекции – класс stack – расширяет кладе vector.

Класс Stack

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

Стек (stack) реализует порядок работы с элементами подобно магазину винтовки– первым выстрелит патрон, положенный в магазин последним,– или подобно железнодорожному тупику – первым из тупика выйдет вагон, загнанный туда последним. Такой порядок обработки называется **LIFO** (Last In – First Out).

Перед работой создается пустой стек конструктором **stack ()**.

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

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

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

Листинг 6.2 показывает, как можно использовать стек для проверки парности символов.

Листинг 6.2. Проверка парности скобок.

```
import java.utii.*;
class StackTesti
static boolean checkParity(String expression,
String open, String close){
Stack stack = new Stack ();
StringTokenizer st = new StringTokenizer(expression,
" \t\n\r+*/-(){}", true);
while (st..hasMoreTokens ()) {
String tmp = st.nextToken();
if (tmp.equals(open)), stack.push(open);
i f (tmp.equals(close)) stack.pop();
}
if (stack.isEmpty ()) return true/return fals e;
}
public static void main(String[] args){
System.out.println(
checkParityC'a - (b - (c - a) / (b + c) - 2), "(", ")));
}
}
```

Как видите, коллекции значительно облегчают обработку наборов данных.

Еще один пример коллекции совсем другого рода – таблицы – предоставляет класс Hashtable.

Класс Hashtable

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

Из таких пар "Фамилия И. О. – номер" состоит, например, телефонный справочник.

Еще один пример – анкета. Ее можно представить как совокупность пар "Фамилия – Иванов", "Имя – Петр", "Отчество – Сидорович", "Год рождения – 1975" и т. д.

Подобных примеров можно привести множество.

Каждый объект класса Hashtable кроме **размера** (size) – количества пар, имеет еще две характеристики: **емкость** (capacity) – размер буфера, и **показатель загруженности** (load factor) – процент заполненности буфера, по достижении которого увеличивается его размер.

Как создать таблицу

Для создания объектов класс Hashtable предоставляет четыре конструктора:

- **Hashtable ()** – создает пустой объект с начальной емкостью в 101 элемент и показателем загруженности 0.75;
- **Hashtable (int capacity)** – создает пустой объект с начальной емкостью capacity и показателем загруженности 0.75;
- **Hashtable(int capacity, float loadFactor)** – создает пустой Объект с начальной емкостью capacity и показателем загруженности loadFactor;
- **Hashtable (Map f)** – создает объект класса Hashtable, содержащий все элементы отображения f, с емкостью, равной удвоенному числу элементов отображения f, но не менее 11, и показателем загруженности 0.75.

Как заполнить таблицу

Для заполнения объекта класса Hashtable используются два метода:

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

Как получить значение по ключу

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

Как узнать наличие ключа или значения

Логический метод **containsKey(object key)** возвращает true, если в таблице есть ключ key.

Логический метод **containsvalue (Object value)** или старый метод **contains (object value)**возвращают true, если в таблице есть ключи со значением value.

Логический метод **isEmpty ()** возвращает true, если в таблице нет элементов.

Как получить все элементы таблицы

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

Метод **keyset ()** предоставляет все ключи key таблицы в виде интерфейса set. Все изменения в объекте set корректируют таблицу, и наоборот.

Метод **entrySet()** представляет все пары " key– value " таблицы в виде интерфейса Set. Все модификации в объекте set изменяют таблицу, и наоборот.

Метод **tostring ()** возвращает строку, содержащую все пары.

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

Как удалить элементы

Метод **remove (Object key)** удаляет пару с ключом key, возвращая значение этого ключа, если оно есть, и null, если пара с ключом key не найдена.

Метод **clear ()** удаляет все элементы, очищая таблицу.

В листинге 6.3 показано, как можно использовать класс Hashtabie для создания телефонного справочника, а на рис. 6.1 – вывод этой программы.

Листинг 6.3. Телефонный справочник.

```
import java.util.*;
class PhoneBook{
public static void main(String[] args){
Hashtabie yp = new Hashtabie();
String name = null;
yp.put("John", "123-45-67");
yp.put ("Lemon", "567-34-12");
yp.put("Bill", "342-65-87");
yp.put("Gates", "423-83-49");
yp.put("Batman", "532-25-08");
try{
name = args[0];
(catch(Exception e){
System.out.println("Usage: Java PhoneBook Name");
return;
}
if (yp.containsKey(name))
System.out.println(name + "'s phone = " + yp.get(name));
else
System.out.println("Sorry, no such name");
)
}
```

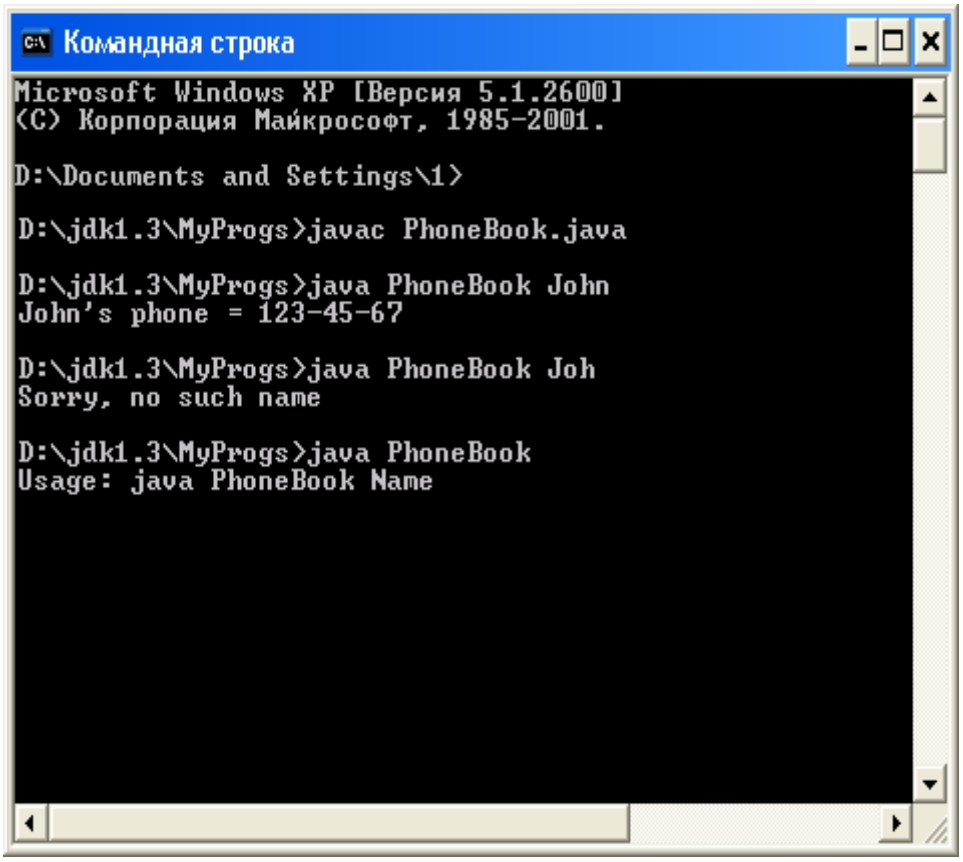

Рис. 6.1. Работа с телефонной книгой

Класс Properties

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

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

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

Два метода, возвращающих значение ключа-строки в виде строки:

- **string getProperty (string key)** – возвращает значение по ключу key;
- **String getProperty(String.key, String defaultValue)** – возвращает значение по ключу key, если такого ключа нет, возвращается defaultValue.

Метод **setProperty(String key, String value)** добавляет новую пару, если ключа key нет, и меняет значение, если ключ key есть.

Метод **load(Inputstream in)** загружает свойства из входного потока in.

Методы **list(PrintStream out)** и **list (PrintWriter out)** выводят свойства в выходной поток out.

Метод **store (OutputStream out, String header)** выводит свойства в выходной поток out с заголовком header.

Очень простой листинг 6.4 и рис. 6.2 демонстрируют вывод всех системных свойств Java.

Листинг 6.4. Вывод системных свойств.

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

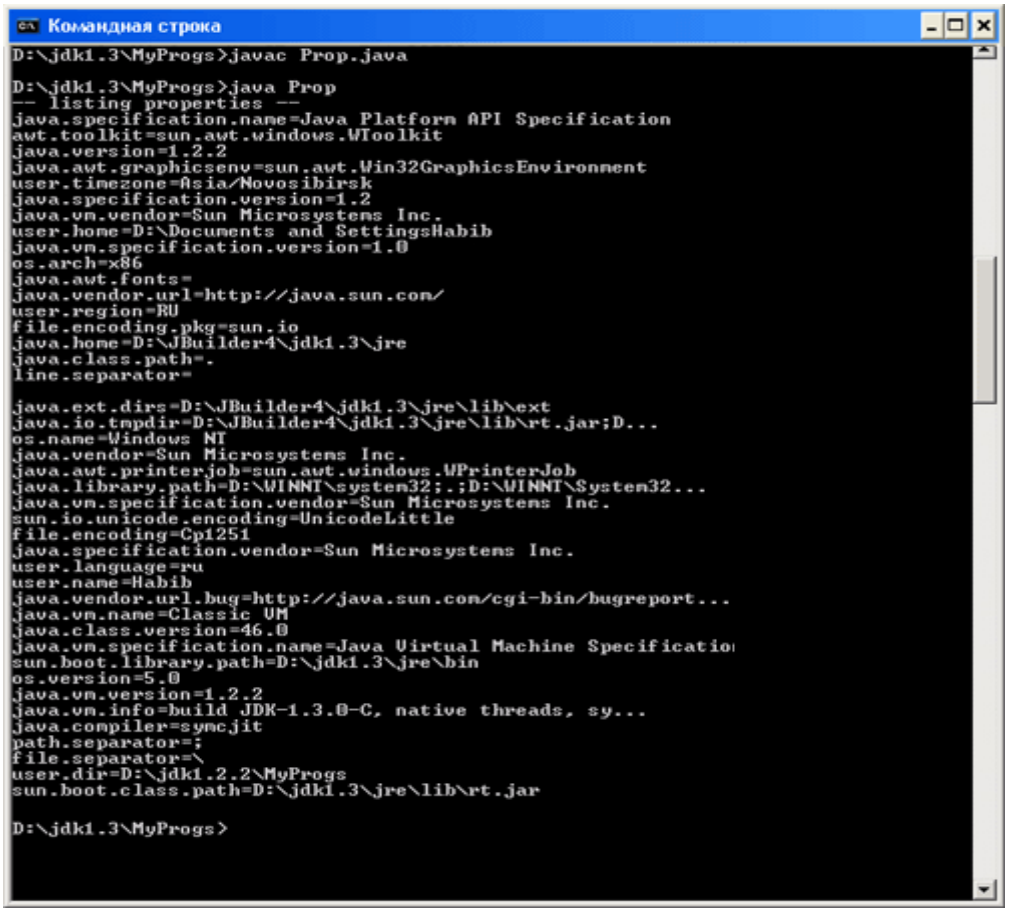

Рис. 6.2. Системные свойства

Примеры классов **Vector**, **Stack**, **Hashtabie**, **Properties** показывают удобство классов-коллекций. Поэтому в Java 2 разработана целая иерархия коллекций. Она показана на рис. 6.3. Курсивом записаны имена интерфейсов. Пунктирные линии указывают классы, реализующие эти интерфейсы. Все коллекции разбиты на три группы, описанные в интерфейсах **List**, **Set** и **Map**.

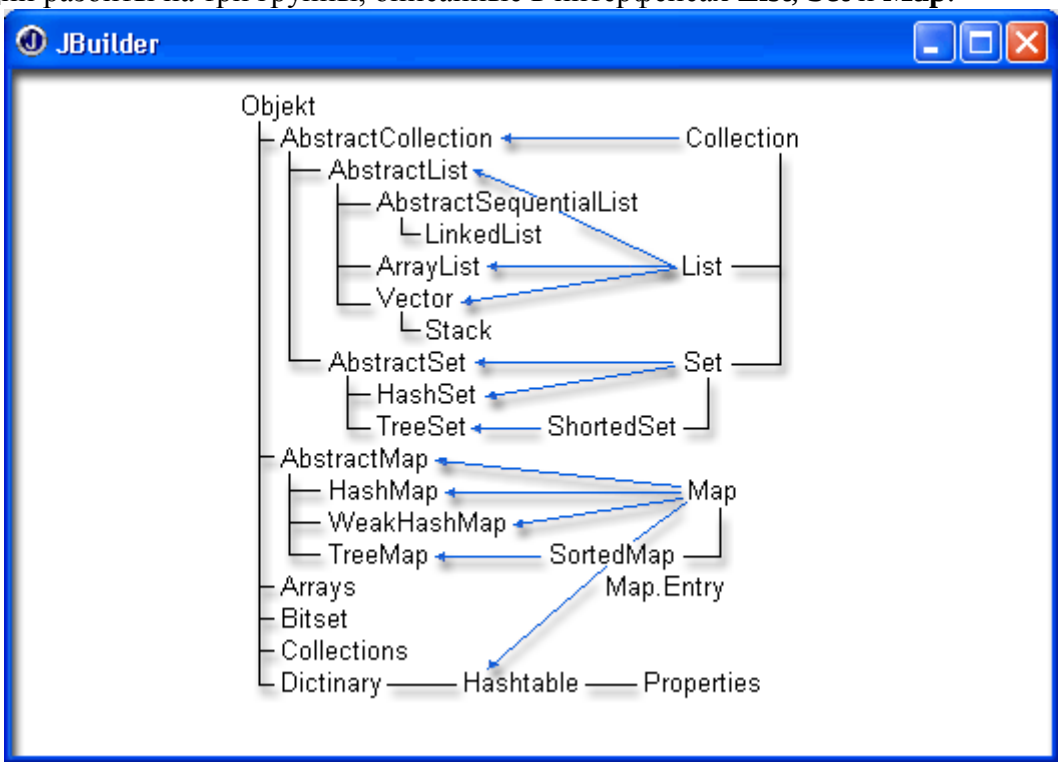

Рис. 6.3. Иерархия классов и интерфейсов-коллекций

Примером реализации интерфейса List может служить класс Vector, примером реализации интерфейса map – класс Hashtabie.

Коллекции **List** и **set** имеют много общего, поэтому их общие методы объединены и вынесены в суперинтерфейс Collection.

Посмотрим, что, по мнению разработчиков Java API, должно содержаться в этих коллекциях.

Интерфейс Collection

Интерфейс **collection** из пакета java.util описывает общие свойства коллекций List и set. Он содержит методы добавления и удаления элементов, проверки и преобразования элементов:

- **boolean add (Object obj)** – добавляет элемент obj в конец коллекции; возвращает false, если такой элемент в коллекции уже есть, а коллекция не допускает повторяющиеся элементы; возвращает true, если добавление прошло удачно;
- **boolean addAll (Collection coll)** – добавляет все элементы коллекции coll в конец данной коллекции;
- **void clear ()** – удаляет все элементы коллекции;
- **boolean contains (Object obj)** – проверяет наличие элемента obj в коллекции;
- **boolean containsAll (Collection coll)** – проверяет наличие всех элементов коллекции coll в данной коллекции;
- **boolean isEmpty()** – проверяет, пуста ли коллекция;
- **iterator iterator ()** – возвращает итератор данной коллекции;
- **boolean remove (object obj)** – удаляет указанный элемент из коллекции; возвращает false, если элемент не найден, true, если удаление прошло успешно;
- **boolean removeAll (Collection coll)** – удаляет элементы указанной коллекции, лежащие в данной коллекции;

- **boolean retainAll (Collection coll)** – удаляет все элементы данной коллекции, кроме элементов коллекции coll;
- **int size ()** – возвращает количество элементов в коллекции;
- **object [] toArray ()** – возвращает все элементы коллекции в виде массива;
- **Objectn toArray(object[] a)** – записывает все элементы коллекции в массив а, если в нем достаточно места.

Интерфейс List

Интерфейс **List** из пакета java.utii, расширяющий интерфейс **collection**, описывает методы работы с упорядоченными коллекциями. Иногда их называют**последовательностями** (sequence). Элементы такой коллекции пронумерованы, начиная от нуля, к ним можно обратиться по индексу.

В отличие от коллекции Set элементы коллекции List могут повторяться.

Класс **vector** – одна из реализаций интерфейса List.

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

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

Интерфейс Set. Интерфейс SortedSet.

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

Этот интерфейс расширен интерфейсом sortedset.

Интерфейс SortedSet

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

Элементы не нумеруются, но есть понятие первого, последнего, большего и меньшего элемента.

Дополнительные методы интерфейса отражают эти понятия:

- **comparator comparator ()** – возвращает способ упорядочения коллекции;
- **object first ()**– возвращает первый, меньший элемент коллекции;
- **SortedSet headset (Object toEiement)** – возвращает начальные, меньшие элементы до элемента toEiement исключительно;

- **object last ()** – возвращает последний, больший элемент коллекции;
- **SortedSet subset (Object fromElement, Object toEiement)** – Возвращает подмножество коллекции от элемента fromElement включительно до элемента toEiement исключительно;
- **SortedSet tailSet (Object fromElement)** – возвращает последние, большие элементы коллекции от элемента fromElement включительно.

Интерфейс Map

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

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

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

Класс **HashTable** – одна из реализаций интерфейса Map.

Интерфейс Map содержит методы, работающие с ключами и значениями:

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

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

Вложенный интерфейс Map.Entry. Интерфейс SortedMap.

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

- методы **getKey()** и **getvalue()** позволяют получить ключ и значение пары;
- метод **setvaiue (object value)** меняет значение в данной паре.

Интерфейс SortedMap

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

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

- **comparator comparator ()** – возвращает способ упорядочения коллекции;
- **object firstKey()** – возвращает первый, меньший элемент коллекции;
- **SortedMap headMap(Object toKey)** – возвращает начало коллекции до элемента с ключом toKey исключительно;
- **object lastKey()** – возвращает последний, больший ключ коллекции;

- **SprtedMap subMap (Object fromKey, Object toKey) –** возвращает часть коллекции от элемента с ключом fromKey включительно до элемента с ключом toKey исключительно;
- **SortedMap tailMap (object fromKey)** – возвращает остаток коллекции от элемента fromKey включительно.

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

Абстрактные классы-коллекции

Эти классы лежат в пакете **java.util**.

Абстрактный класс **AbstractGollection** реализует интерфейс **Collection**, но оставляет нереализованными методы **iterator ()**, **size ()**.

Абстрактный класс **AbstractList** реализует интерфейс List, но оставляет нереализованным метод **get(mt)** и унаследованный метод **size()**. Этот класс позволяет реализовать коллекцию с прямым доступом к элементам, подобно массиву.

Абстрактный класс**AbstractSequantaaiList** реализует интерфейс List, но оставляет нереализованным метод **listiteratordn(index)** и унаследованный метод **size ()**. Данный класс позволяет реализовать коллекции с последовательным доступом к элементам с помощью итератора **Listiterator**.

Абстрактный класс Abstractset реализует интерфейс Set, но оставляет нереализованными методы, унаследованные от**Abstractcollection**.

Абстрактный класс **AbstractMap** реализует интерфейс Map, но оставляет нереализованным метод **entrySet ()**.

Наконец, в составе Java API есть полностью реализованные классы-коллекции помимо уже рассмотренных классов **Vectdr**, **Stack**, **Hashtable** и **Properties**.

Это классы **ArrayList**, **LinkedList**, **HashSet**, **TreeSet**, **HashMap**, **TreeMap**, **WeakHashMap**.

Для работы с этими классами разработаны интерфейсы **iterator**, **Listiterator**, **Comparator**и классы **Arrays** и **Collections**.

Перед тем как рассмотреть использование данных классов, обсудим понятие итератора.

Интерфейс Iterator

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

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

Этим вопросам посвящена обширная литература.

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

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

В интерфейсе iterator описаны всего три метода:

- логический метод **hasNext ()** возвращает true, если обход еще не завершен;
- метод **next ()** делает текущим следующий элемент коллекции и возвращает его в виде объекта класса object;
- метод **remove ()** удаляет текущий элемент коллекции.

Можно представить себе дело так, что итератор – это указатель на элемент коллекции. При создании итератора указатель устанавливается перед первым элементом, метод next () перемещает указатель на первый элемент и показывает его. Следующее применение метода next () перемещает указатель на второй элемент коллекции и показывает его. Последнее применение метода **next ()** выводит указатель за последний элемент коллекции.

Метод **remove ()**, пожалуй, излишен, он уже не относится к задаче обхода коллекции, но позволяет при просмотре коллекции удалять из нее ненужные элементы.

В листинге 6.5 к тексту листинга 6.1 добавлена работа с итератором.

Листинг 6.5. Использование итератора вектора.

```
Vector v = new Vector();
String s = "Строка, которую мы хотим разобрать на слова.";
StringTokenizer st = new StringTokenizer(s, " \t\n\r,.");
while (st.hasMoreTokens()){
// Получаем слово и заносим в вектор.
v.add(st.nextToken()); // Добавляем в конец вектора }
System.out.print*Ln(v.firstElement(}); // Первый элемент
System.out.println(v.lastElement()); // Последний элемент
v.setSize(4); // Уменьшаем число элементов
v.add("собрать."); // Добавляем в конец укороченного вектора
v.set(3, "опять"); // Ставим в позицию 3
for (int i = 0; i < v.sizeO; i++) // Перебираем весь вектор
System.out.print(v.get(i) + ".");
System.out.println(};
Iterator it = v.iterator (); // Получаем итератор вектора
try{
while(it.hasNext()) // Пока в векторе есть элементы,
System.out.println(it.next()); // выводим текущий элемент
}catch(Exception e){}
```

Интерфейс Listlterator

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

В интерфейс Listiterator добавлены следующие методы:

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

Как видите, итераторы могут изменять коллекцию, в которой они работают, добавляя, удаляя и заменяя элементы. Чтобы это не приводило к конфликтам, предусмотрена исключительная ситуация, возникающая при попытке использования итераторов параллельно "родным" методам коллекции. Именно поэтому в листинге 6.5 действия с итератором заключены в блок tryUcatch(){}.

Изменим окончание листинга 6.5 с использованием итератора Listiterator.

```
// Текст листинга 6.1…
//…
Listiterator lit = v.listlterator(); // Получаем итератор вектора
// Указатель сейчас находится перед началом вектора
try{
while(lit.hasNext()) // Пока в векторе есть элементы
System.out.println(lit.next()); // Переходим к следующему
// элементу и выводим его
// Теперь указатель за концом вектора. Пройдем к началу
while (lit .hasPrevious ())
System, out .printlnf lit .previblis ());:
}catch (Exception e) ()
```

Интересно, что повторное применение методов **next ()** и **previous ()** друг за другом будет выдавать один и тот же текущий элемент. Посмотрим теперь, какие возможности предоставляют классы-коллекции Java 2.

Классы, создающие списки. Двунаправленный список.

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

В классе ArrayList три конструктора:

- **ArrayList ()** – создает пустой объект;
- **ArrayList (Collection coil)** – создает объект, содержащий все элементы коллекции coll;
- **ArrayList (int initCapacity)** – создает пустой Объект емкости initCapacity.

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

Эти вопросы мы рассмотрим в главе 17.

Двунаправленный список

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

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

В классе LinkedList два конструктора:

- **LinkedList** – создает пустой объект
- **LinkedList (Collection coil)** – создает объект, содержащий все элементы коллекции coll.

Классы, создающие отображения. Упорядоченные отображения.

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

- **HashMap ()** – создает пустой объект с показателем загруженности 0.75;
- **HashMap(int.capacity)** – создает пустой объект с начальной емкостью capacity и показателем загруженности 0.75;
- **HashMap (int capacity, float loadFactor)** – создает пустой объект с начальной емкостью capacity и показателем загруженности loadFactor;

- **HashMap (Map f)** – создает объект класса HashMap, содержащий все элементы отображения f, с емкостью, равной удвоенному числу элементов отображения f, но не менее 11, и показателем загруженности 0.75.

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

Упорядоченные отображения

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

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

В этом классе четыре конструктора:

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

Здесь надо пояснить, каким образом можно задать упорядоченность элементов коллекции.

Сравнение элементов коллекций

Интерфейс Comparator описывает два метода сравнения:

- **int compare (Object obji, object obj2)** – возвращает отрицательное число, если obj1 в каком-то смысле меньше obj2; нуль, если они считаются равными; положительное число, если obj1 больше obj2. Для читателей, знакомых с теорией множеств, скажем, что этот метод сравнения обладает свойствами тождества, антисимметричности и транзитивности;
- **boolean equals (Object obj)** – сравнивает данный объект с объектом obj, возвращая true, если объекты совпадают в каком-либо смысле, заданном этим методом.

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

Листинг 6.6 показывает один из возможных способов упорядочения комплексных чисел – объектов класса complex из листинга 2.4. Здесь описывается класс **ComplexCompare**, реализующий интерфейс Comparator, В листинге **6.7** он применяется для упорядоченного хранения множества комплексных чисел.

Листинг 6.6. Сравнение комплексных чисел.

```
import java.util.*;
class ComplexCompare implements Comparator{
public int compare(Object objl, Object obj2){
Complex zl = (Complex)objl, z2 = (Complex)obj2;
double rel = zl.getReO, iml = zl.getlm();
double re2 = z2.getRe(), im2 = z2.getlm();
if (rel!= re2) return (int)(rel – re2);
else if (iml!= im2) return (int)(iml – im2);
else return 0;
}
public boolean equals(Object z) {
return compare(this, z) == 0;
}
}
```

Классы, создающие множества. Упорядоченные множества.

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

В классе HashSet четыре конструктора:

- **Hashset ()** – создает пустой объект с показателем загруженности 0.75;
- **HashSet (int capacity)** – создает пустой объект с начальной емкостью capacity и показателем загруженности 0.75;
- **HashSet (int capacity, float loadFactor)** – создает пустой объект с начальной емкостью capacity и показателем загруженности loadFactor;
- **HashSet (Collection coll)** – создает объект, содержащий все элементы коллекции coll, с емкостью, равной удвоенному числу элементов коллекции coll, но не менее 11, и показателем загруженности 0.75.

Упорядоченные множества

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

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

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

В классе TreeSet четыре конструктора:

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

В листинге 6.7 показано, как можно хранить комплексные числа в упорядоченном виде. Порядок задается объектом класса **ComplexCompare**, определенного в листинге 6.6.

Листинг 6.7. Хранение комплексных чисел в упорядоченном виде.

```
TreeSet ts = new TreeSet (new ComptexCompare());
ts.add(new Complex(1.2, 3.4));
ts .add (new Complex (-1.25, 33.4";
ts.add(new Complex(1.23, – 3.45));
ts.add(new Complex(16.2, 23.4));
Iterator it = ts.iterator();
while(it.hasNext()), ((Complex)it.next()).pr();
```

Действия с коллекциями. Методы класса Collections.

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

Все методы класса collections статические, ими можно пользоваться, не создавая экземпляры классу **Collections**.

Как обычно в статических методах, коллекция, с которой работает метод, задается его аргументом.

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

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

После сортировки можно осуществить бинарный поиск в коллекции:

- **static int binarySearch(List coll, Object element)** – отыскивает элемент element в отсортированной в естественном порядке возрастания коллекции coll и возвращает индекс элемента или отрицательное число, если элемент не найден; отрицательное число показывает индекс, с которым элемент element был бы вставлен в коллекцию, с обратным знаком;
- **static int binarySearch(List coll, Object element, Comparator c)** – то же, но коллекция отсортирована в порядке, определенном объектом с.

Четыре метода находят наибольший и наименьший элементы в упорядочиваемой коллекции:

- **static object max (Collection coll)** – возвращает наибольший в естественном порядке элемент коллекции coll;
- **static Object max (Collection coll, Comparator c)** – то же в порядке, заданном объектом с;
- **static object mm (Collection coll)** – возвращает наименьший в естественном порядке элемент коллекции сои;
- **static Object min(Collection coll, Comparator c)** – то же в порядке, заданном объектом с.

Два метода "перемешивают" элементы коллекции в случайном порядке:

- **static void shuffle (List coll)** – случайные числа задаются по умолчанию;
- **static void shuffle (List coll, Random r)** – случайные числа определяются объектом r.

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

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

Метод **fill (List coll, object element)** заменяет все элементы существующей коллекции coll элементом element.

С остальными методами познакомимся по мере надобности.

Заключение

Итак, в данной главе мы выяснили, что язык Java предоставляет множество средств для работы с большими объемами информации. В большинстве случаев достаточно добавить в программу три-пять операторов, чтобы можно было проделать нетривиальную обработку информации.

В следующей главе мы рассмотрим аналогичные средства для работы с массивами, датами, для получения случайных чисел и прочих необходимых средств программирования.

Классы-утилиты

- ## Работа с массивами
  В этой главе описаны средства, полезные для создания программ: работа с массивами, датами, случайными числами. | В классе Arrays из пакета java.utii собрано множество методов для работы с массивами. Их можно разделить на четыре группы.
- ## Локальные установки
  Некоторые данные – даты, время – традиционно представляются в разных местностях по-разному. Например, дата в России выводится в формате число, месяц, год через точку: 27.06.01. В США принята запись месяц/число/год через наклонную черту: 06/27/01.
- ## Работа с датами и временем
  Методы работы с датами и показаниями времени собраны в два класса: Calendar и Date из пакета java.utii. | Объект класса Date хранит число миллисекунд, прошедших с 1 января 1970 г. 00:00:00 по Гринвичу. Это "день рождения" UNIX, он называется " Epoch ".
- ## Часовой пояс и летнее время. Класс Calendar.
  Методы установки и изменения часового пояса (time zone), а также летнего времени DST (Daylight Savings Time), собраны в абстрактном классе Timezone из пакета java.utii. В этом же пакете есть его реализация – подкласс SimpleTimeZone.
- ## Подкласс GregorianCalendar
  В григорианском календаре две целочисленные константы определяют эры: BC (before Christ) и AD (Anno Domini). | Семь конструкторов определяют календарь по времени, часовому поясу и/или локали: | GregorianCalendar() | GregorianCalendar(int year, int month, int date)
- ## Представление даты и времени
  Различные способы представления дат и показаний времени можно осуществить методами, собранными в абстрактный класс DateFormat и его подкласс SimpleDateFormat из пакета Java.text. | Класс DateFormat предлагает четыре стиля представления даты и времени:
- ## Получение случайных чисел. Копирование массивов.
  Получить случайное неотрицательное число, строго меньшее единицы, в виде типа double можно статическим методом random () из класса java.lang.Math. | При первом обращении к этому методу создается генератор псевдослучайных чисел, который используется потом при получении следующих случайных чисел.
- ## Взаимодействие с системой
  Класс System позволяет осуществить и некоторое взаимодействие с системой во время выполнения программы (run time). Но кроме него для этого есть специальный класс Runtime. | Класс Runtime содержит некоторые методы взаимодействия с JVM во время выполнения программы.

Работа с массивами

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

В классе Arrays из пакета java.utii собрано множество методов для работы с массивами. Их можно разделить на четыре группы.

Восемнадцать статических методов сортируют массивы с разными типами числовых элементов в порядке возрастания чисел или просто объекты в их естественном порядке.

Восемь из них имеют простой вид:

```
static void sort(type[] a)
```

Где type может быть один из семи примитивных типов **byte**, **short**, **int**, **long**, **char**, **float**,**double** или тип **Object**.

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

```
static void sort(type[] a, int from, int to)
```

Оставшиеся два метода сортировки упорядочивают массив или его часть с элементами типа object по правилу, заданному объектом с, реализующим интерфейс Comparator:

```
static void sort(Object[] a, Comparator c)
static void sort(Object[] a, int from, int to, Comparator c)
```

После сортировки можно организовать бинарный поиск в массиве одним из девяти статических методов поиска. Восемь методов имеют вид:

```
static int binarySearch(type[] a, type element)
```

Где type – один из тех же восьми типов.

Девятый метод поиска имеет вид:

```
static int binarySearch(Object[] a, Object element, Comparator c).
```

Он отыскивает элемент element в массиве, отсортированном в порядке, заданном объектом с.

Методы поиска возвращают индекс найденного элемента массива. Если элемент не найден, то возвращается отрицательное число, означающее индекс, с которым элемент был бы вставлен в массив в заданном порядке, с обратным знаком.

Восемнадцать статических методов заполняют массив или часть массива указанным значением value:

```
static void fill(type[], type value)
static void fill(type[], int from, int to, type value)
```

Где type – один из восьми примитивных типов или тип object. Наконец, девять статических логических методов сравнивают массивы:

```
static boolean equals(type[] al, type[] a2)
```

Где type – один из восьми примитивных типов или тип Object.

Массивы считаются равными, и возвращается true, если они имеют одинаковую длину и равны элементы массивов с одинаковыми индексами.

В листинге 7.1 приведен простой пример работы с этими методами.

Листинг 7.1. Применение методов класса Arrays.

```
import java.utii.*;
class ArraysTest{
public static void main(String[] args){
int[] a = {34, -45, 12, 67, -24, 45, 36, -56};
Arrays.sort(a);
for (int i = 0; i < a.length; i++)
System.out.print (a[i]. + " ");
System.out.println();
Arrays.fill(a, Arrays.binarySearch(a, 12), a.length, 0);
for (int i = 6; i < a.length; i++)
System.out.print(a[i] + " ");
System.out.println();
}
}
```

Локальные установки

Некоторые данные – даты, время – традиционно представляются в разных местностях по-разному. Например, дата в России выводится в формате число, месяц, год через точку: 27.06.01. В США принята запись месяц/число/год через наклонную черту: 06/27/01.

Совокупность таких форматов для данной местности, как говорят на жаргоне "локаль", хранится в объекте класса **Locale** из пакета java.utii. Для создания такого объекта достаточно знать язык language и местность country. Иногда требуется третья характеристика – вариант variant, определяющая программный продукт, например, "WIN", "MAC", "POSIX".

По умолчанию местные установки определяются операционной системой и читаются из системных свойств. Посмотрите на строки (см. рис. 6.2):

```
user.language = ru // Язык – русский
user.region = RU // Местность – Россия
file.encoding = Cpl251 // Байтовая кодировка – CP1251
```

Они определяют русскую локаль и локальную кодировку байтовых символов. Локаль, установленную по умолчанию на той машине, где выполняется программа, можно выяснить статическим методом **Locale.getoefault()**.

Чтобы работать с другой локалью, ее надо прежде всего создать. Для этого в классе**Locale** есть два конструктора:

```
Locale(String language, String country)
Locale(String language, String country. String variant)
```

Параметр **language** – это строка из двух строчных букв, определенная стандартом ISO639, например, "ru", "fr", "en". Параметр **country** – строка из двух прописных букв, определенная стандартом ISO3166, например, "RU", "us", "ев". Параметр **variant** не определяется стандартом, это может быть, например, строка " Traditional ".

Локаль часто указывают одной строкой "ru_RU", "en_GB", "en_us", "en_CA " и т. д.

После создания локали можно сделать ее локалью по умолчанию статическим методом:

```
Locale.setDefault(Locale newLocale);
```

Несколько статических методов класса Locale позволяют получить параметры локали по умолчанию, или локали, заданной параметром locale:

- **string getcountry()** – стандартный код страны из двух букв;
- **string getDispiayCountry()** – страна записывается словом, обычно выводящимся на экран;

- **String getDisplayCountry (Locale locale) –** то же для указанной локали.

Такие же методы есть для языка и варианта.

Можно просмотреть список всех локалей, определенных для данной JVM, и их параметров, выводимый в стандартном виде:

```
Locale[] getAvailableLocales()
String[] getlSOCountries()
String[] getlSOLanguages()
```

Установленная локаль в дальнейшем используется при выводе данных в местном формате.

Работа с датами и временем

Методы работы с датами и показаниями времени собраны в два класса: **Calendar** и **Date**из пакета java.utii.

Объект класса Date хранит число миллисекунд, прошедших с 1 января 1970 г. 00:00:00 по Гринвичу. Это "день рождения" UNIX, он называется " Epoch ".

Класс Date удобно использовать для отсчета промежутков времени в миллисекундах.

Получить текущее число миллисекунд, прошедших с момента Epoch на той машине, где выполняется программа, можно статическим методом:

```
System.currentTimeMillis()
```

В классе Date два конструктора. Конструктор **Date ()** заносит в создаваемый объект текущее время машины, на которой выполняется программа, по системным часам, а конструктор **Date (long millisec)** – указанное число.

Получить значение, хранящееся в объекте, можно методом **long getTime ()**, установить новое значение – методом **setTime(long newTime).**

Три логических метода сравнивают отсчеты времени:

- **boolean after (long when)** – возвращает true, если время when больше данного;
- **boolean before (long when)** – возвращает true, если время when меньше данного;
- **boolean after (Object when)** – возвращает true, если when – объект класса Date и времена совпадают.

Еще два метода, сравнивая отсчеты времени, возвращают отрицательное число типа int, если данное время меньше аргумента when; нуль, если времена совпадают; положительное число, если данное время больше аргумента when:

- **int compareTo(Date when)**;
- **int compareTo(object when)** – если when не относится к объектам класса Date, создается исключительная ситуация.

Преобразование миллисекунд, хранящихся в объектах класса Date, в текущее время и дату производится методами класса calendar.

Часовой пояс и летнее время. Класс Calendar.

Методы установки и изменения часового пояса (**time zone**), а также летнего времени DST (**Daylight Savings Time**), собраны в абстрактном классе Timezone из пакета java.utii. В этом же пакете есть его реализация – подкласс **SimpleTimeZone**.

В классе simpieTimeZon е три конструктора, но чаще всего объект создается статическим методом **getoefault()**, возвращающим часовой пояс, установленный на машине, выполняющей программу.

В этих классах множество методов работы с часовыми поясами, но в большинстве случаев требуется только узнать часовой пояс на машине, выполняющей программу, статическим методом **getDefault ()**, проверить, осуществляется ли переход на летнее время, логическим методом **useDaylightTime ()**, и установить часовой пояс методом **setDef ault (TimeZone zone)**.

Класс Calendar

Класс **Calendar** – абстрактный, в нем собраны общие свойства календарей: юлианского, григорианского, лунного. В Java API пока есть только одна его реализация – подкласс**GregorianCalendar**.

Поскольку **calendar** – абстрактный класс, его экземпляры создаются четырьмя статическими методами по заданной локали и/или часовому поясу:

- **Calendar getInstance()**
- **Calendar getInstance(Locale loc)**
- **Calendar getInstance(TimeZone tz)**
- **Calendar getInstance(TimeZone tz, Locale loc)**

Для работы с месяцами определены целочисленные константы от JANUARY до DECEMBER, 3 для работы с днями недели – константы MONDAY до SUNDAY.

Первый день недели можно узнать методом **int getFirstDayOfweek()**, а установить – методом **setFirstDayOfWeek(int day)**, например:

```
setFirstDayOfWeek(Calendar.MONDAY)
```

Остальные методы позволяют просмотреть время и часовой пояс или установить их.

Подкласс GregorianCalendar

В григорианском календаре две целочисленные константы определяют эры: **BC** (before Christ) и **AD** (Anno Domini).

Семь конструкторов определяют календарь по времени, часовому поясу и/или локали:

- GregorianCalendar()
- GregorianCalendar(int year, int month, int date)
- GregorianCalendar(int year, int month, int date, int hour, int minute)
- GregorianCalendar(int year, int month, int date, int hour, int minute, int second)
- GregorianCalendar(Locale loc)
- GregorianCalendar(TimeZone tz)
- GregorianCalendar(TimeZone tz, Locale loc)

После создания объекта следует определить дату перехода **с** юлианского календаря на григорианский календарь методом **setGregorianChange(Date date)**. По умолчанию это 15 октября 1582 г. На территории России переход был осуществлен 14 февраля 1918 г., значит, создание объекта greg надо выполнить так:

```
GregorianCalendar greg = new GregorianCalendar();
greg.setGregorianChange(new GregorianCalendar(1918, Calendar.FEBRUARY, 14).getTime ());
```

Узнать, является ли год високосным в григорианском календаре, можно логическим методом **isLeapYear ()**.

Метод get (**int field**) возвращает элемент календаря, заданный аргументом field. Для этого аргумента в классе Calendar определены следующие статические целочисленные константы:

- ERA WEEK_OF_YEAR DAY_OF_WEEK SECOND
- YEAR WEEK_OF_MONTH DAY_OF_WEEK_IN_MONTH MILLISECOND
- MONTH DAY_OF_YEAR HOUR_OF_DAY ZONE_OFFSET
- DATE DAY_OF_MONTH MINUTE DST_OFFSET

Несколько методов **set ()**, использующих эти константы, устанавливают соответствующие значения.

Представление даты и времени

Различные способы представления дат и показаний времени можно осуществить методами, собранными в абстрактный класс DateFormat и его подкласс **SimpleDateFormat**из пакета Java.text.

Класс DateFormat предлагает четыре стиля представления даты и времени:

- стиль **SHORT** представляет дату и время в коротком числовом виде: 27.04.01 17:32; в локали США: 4/27/01 5:32 PM;
- стиль **MEDIUM** задает год четырьмя цифрами и показывает секунды: 27.04.2001 17:32:45; в локали США месяц представляется тремя буквами;
- стиль **LONG** представляет месяц словом и добавляет часовой пояс: 27 апрель 2001 г. 17:32:45 GMT+03.-00;
- стиль **FULL** в русской локали таков же, как и стиль LONG; в локали США добавляется еще день недели.

Есть еще стиль DEFAULT, совпадающий со стилем MEDIUM.

При создании объекта класса **simpieDateFormat** можно задать в конструкторе шаблон, определяющий какой-либо другой формат, например:

```
SimpieDateFormat sdf = new SimpieDateFormat("dd-MM-yyyy hh.iran");
System.out.println(sdf.format(new Date()));
```

Получим вывод в таком виде:

```
27-04-2001 17.32.
```

В шаблоне буква d означает цифру дня месяца, м – цифру месяца, у – цифру года, h – цифру часа, m – цифру минут. Остальные обозначения для шаблона указаны В Документации по классу **SimpieDateFormat.**

Эти буквенные обозначения можно изменить с помощью класса **DateFormatSymbols.**

Не во всех локалях можно создать объект класса SimpieDateFormat. В таких случаях используются статические методы **getinstance()** класса DateFormat, возвращающие объект класса DateFormat. Параметрами этих методов служат стиль представления даты и времени и, может быть, локаль.

После создания объекта метод **format ()** класса DateFormat возвращает строку с датой и временем, согласно заданному стилю. В качестве аргумента задается объект класса Date.

Например:

```
System.out.println("LONG: " + DateFormat.getDateTimelnstance(
DateFormat. LONG, DateFormat. LONG) .format (new Date ()));
```

…или:

```
System.out.println("FULL: " + DateFormat.getDateTimelnstance(
DateFormat.FULL,DateFormat.FULL, Locale.US).format(new Date()));
```

Получение случайных чисел. Копирование массивов.

Получить случайное неотрицательное число, строго меньшее единицы, в виде типа double можно статическим методом **random ()** из класса java.lang.Math.

При первом обращении к этому методу создается генератор псевдослучайных чисел, который используется потом при получении следующих случайных чисел.

Более серьезные действия со случайными числами можно организовать с помощью методов класса Random из пакета java.utii. В классе два конструктора:

- **Random (long seed)** – создает генератор псевдослучайных чисел, использующий для начала работы число seed;
- **Random()** – выбирает в качестве начального значения текущее время.

Создав генератор, можно получать случайные числа соответствующего типа методами**nextBoolean()**, **nextDouble()**, **nextFloat()**, **nextGau.ssian()**, **next int()**, **nextLong()**, **nextint(int max)** или записать сразу последовательность случайных чисел в заранее определенный массив байтов bytes методом **nextBytes(byte[] bytes)**.

Вещественные случайные числа равномерно располагаются в диапазоне от 0.0 включительно до 1.0 исключительно. Целые случайные числа равномерно распределяются по всему диапазону соответствующего типа за, одним исключением: если в аргументе указано целое число max, то диапазон случайных чисел будет от нуля включительно до max исключительно.

Копирование массивов

В классе System из пакета java.iang есть статический метод копирования массивов, который использует сама исполняющая система Java. Этот метод действует быстро и надежно, его удобно применять в программах. Синтаксис:

```
static void arraycopy(Object src, int src_ind, Object dest, int dest_ind, int count)
```

Из массива, на который указывает ссылка src, копируется count элементов, начиная с элемента с индексом **src_ind**, в массив, на который указывает ссылка dest, начиная с его элемента с индексом **dest_ind**.

Все индексы должны быть заданы так, чтобы элементы лежали в массивах, типы массивов должны быть совместимы, а примитивные типы обязаны полностью совпадать. Ссылки на массивы не должны быть равны null.

Ссылки src и dest могут совпадать, при этом для копирования создается промежуточный буфер. Метод можно использовать, например, для сдвига элементов в массиве. После выполнения:

```
int[] arr = {5, 6, 1, 8, 9, 1, 2, 3, 4, 5, -3, -7};
System.arraycopy(arr, 2, arr, 1, arr.length -2);
```

…получим:

```
(5, 7, 8, 9, 1, 2, 3, 4, 5, -3, -7, -7}.
```

Взаимодействие с системой

Класс **System** позволяет осуществить и некоторое взаимодействие с системой во время выполнения программы (**run time**). Но кроме него для этого есть специальный класс Runtime.

Класс **Runtime** содержит некоторые методы взаимодействия с JVM во время выполнения программы. Каждое приложение может получить только один экземпляр данного класса статическим методом **getRuntime ()**. Все вызовы этого метода возвращают ссылку на один и тот же объект.

Методы **freeMemory ()** и **totalMemory ()** возвращают количество свободной и всей памяти, находящейся в распоряжении JVM для размещения объектов, в байтах, в виде числа типа long. Не стоит твердо опираться на эти числа, поскольку количество памяти меняется динамически.

Метод **exit (int status)** запускает процесс останова JVM и передает операционной системе статус завершения status. По соглашению, ненулевой статус означает ненормальное завершение. Удобнее использовать аналогичный метод класса system, который является статическим.

Метод **hait (int status)** осуществляет немедленный останов JVM. Он не завершает запущенные процессы нормально и должен использоваться только в аварийных ситуациях.

Метод **loadbibrary(string libName)** позволяет подгрузить динамическую библиотеку во время выполнения по ее имени libName.

Метод **load (string fileName)** подгружает динамическую библиотеку по имени файла fileName, в котором она хранится.

Впрочем, вместо этих методов удобнее использовать статические методы класса system с теми же именами и аргументами.

Метод **gc()** запускает процесс освобождения ненужной оперативной памяти (**garbage collection**). Этот процесс периодически запускается самой виртуальной машиной Java и выполняется на фоне с небольшим приоритетом, но можно его запустить и из программы. Опять-таки удобнее использовать статический Метод **System.gc ()**.

Наконец, несколько методов **exec ()** запускают в отдельных процессах исполнимые файлы. Аргументом этих методов служит командная строка исполнимого файла.

Например, **Runtime.getRuntime ().exec ("notepad")** запускает программу Блокнот на платформе MS Windows.

Методы **exec ()** возвращают экземпляр класса process, позволяющего управлять запущенным процессом. Методом **destroy ()** можно остановить процесс, методом **exitValue()**получить его код завершения. метод **waitFor()** приостанавливает основной подпроцесс до тех пор, пока не закончится запущенный процесс. Три метода **getlnputStream()**,**getOutputStream()** и **getErrorStream()** возвращают входной, выходной поток и поток ошибок запущенного процесса (см. главу 18).