Лабораторная работа №1

Введение в язык программирования Java. Виртуальная машина Java. Базовые средства языка Java.

Цель лабораторной работы:	1. Познакомиться с работой виртуальной машины Java. 2. Познакомиться с JDK. 3. Научиться создавать, компилировать и выполнять приложения, написанные на Java. 4. Познакомиться с базовыми типами и возможностями языка Java
Время для выполнения:	[redacted]

Содержание работы.

1. Теоретические сведения

1.1 Java машина и JDK. Компиляция программы. [читать ☐]

1.2 Примитивные типы данных в языке Java

Описание простых типов языка Java читаем здесь

.

1.3 Управление потоком выполнения программы

Описание операторов языка Java читаем здесь

.

1.4 Работа со строками

Описание работы со строками в языке Java читаем здесь

.

1.5 Ввод-вывод информации с консоли[читать]

2. Задание на лабораторную работу

Задание 1

Для x изменяющегося от a до b с шагом n, где n - задано (см.условие задания 1), вычислить функцию f(x), используя ее разложение в степенной ряд в трех случаях:
1) для «точного» значения (по аналитической формуле).
2) для заданного k (запросить у пользователя);
3) для заданной точности e (запросить у пользователя);

Для сравнения найти относительную погрешность вычисления функции значение функции
o_погр = ABS((точ_знач – приближ_знач)/ точ_знач)
Результаты расчетов отпечатать в следующем виде:

```
      Вычисление функции  "написать какой"

   X      Y       Y1      Y2      погр1    погр2
 ......  ....    .....   .....     ....     ...
 ......  ....    .....   .....     ....     ...

 ......  ....    .....   .....     ....     ...
 ......  ....    .....   .....     ....     ...
```

Здесь X- значение параметра; Y1- значение суммы для заданного n; Y2- значение суммы для заданной точности; Y-точное значение функции; погр1, погр2 – относительные погрешности приближенных вычислений.

Значения a, b и f(x) согласно варианту можно посмотреть здесь

.
Математическое описание разложения функции в ряд описано здесь

.

пример и дополнительную информацию можно посмотреть здесь

Задание 2

Задана строка, состоящая из символов. Символы объединяются в слова. Слова друг от друга отделяются одним или несколькими пробелами или другими символами разделителями. Выполнить ввод строки с консоли и обработку ее в соответствии со своим вариантом.

Задания согласно варианту можно посмотреть здесь

.

Примеры можно посмотреть здесь html посмотреть и скачать здесь

работу со строками можно посмотреть здесь

3. Отчет

1. Постановка задач.
2. Вариант задания.
3. Математическая модель (формулы, по которым выполняются вычисления слагаемых ряда).
4. Программа.
5. Полученные результаты работы написанной программы.

Успехов!

Простые типы в Java

Простые типы в Java не являются объектно-ориентированными, они аналогичны простым типам большинства традиционных языков программирования. В Java имеется восемь простых типов: — byte, short, int, long, char, float, double и boolean. Их можно разделить на четыре группы:

1. Целые. К ним относятся типы byte, short, int и long. Эти типы предназначены для целых чисел со знаком.
2. Типы с плавающей точкой — float и double. Они служат для представления чисел, имеющих дробную часть.
3. Символьный тип char. Этот тип предназначен для представления элементов из таблицы символов, например, букв или цифр.
4. Логический тип boolean. Это специальный тип, используемый для представления логических величин.

В Java, в отличие от некоторых других языков, отсутствует автоматическое приведение типов. Несовпадение типов приводит не к предупреждению при трансляции, а к сообщению об ошибке. Для каждого типа строго определены наборы допустимых значений и разрешенных операций.

Целые числа

В языке Java понятие беззнаковых чисел отсутствует. Все числовые типы этого языка — знаковые. Например, если значение переменной типа byte равно в шестнадцатиричном виде 0x80, то это — число -1.

Отсутствие в Java беззнаковых чисел вдвое сокращает количество целых типов. В языке имеется 4 целых типа, занимающих 1, 2, 4 и 8 байтов в памяти. Для каждого типа — byte, short, int и long, есть свои естественные области применения.

byte

Тип byte — это знаковый 8-битовый тип. Его диапазон — от -128 до 127. Он лучше всего подходит для хранения произвольного потока байтов, загружаемого из сети или из файла.

```
byte b;

byte c = 0x55;
```

Если речь не идет о манипуляциях с битами, использования типа byte, как правило, следует избегать. Для нормальных целых чисел, используемых в качестве счетчиков и в арифметических выражениях, гораздо лучше подходит тип int.

short

short — это знаковый 16-битовый тип. Его диапазон — от -32768 до 32767. Это, вероятно, наиболее редко используемый в Java тип, поскольку он определен, как тип, в котором старший байт стоит первым.

```
short s;

short t = 0x55aa;
```

int

Тип int служит для представления 32-битных целых чисел со знаком. Диапазон допустимых для этого типа значений — от -2147483648 до 2147483647. Чаще всего этот тип данных используется для хранения обычных целых чисел со значениями, достигающими двух миллиардов. Этот тип прекрасно подходит для использования при обработке массивов и для счетчиков. В ближайшие годы этот тип будет прекрасно соответствовать машинным словам не только 32-битовых процессоров, но и 64-битовых с поддержкой быстрой конвейеризации для выполнения 32-битного кода в режиме совместимости. Всякий раз, когда в одном выражении фигурируют переменные типов byte, short, int и целые литералы, тип всего выражения перед завершением вычислений приводится к int.

```
int i;

int j = 0x55aa0000;
```

long

Тип long предназначен для представления 64-битовых чисел со знаком. Его диапазон допустимых значений достаточно велик даже для таких задач, как подсчет числа атомов во вселенной.

```
long m;

long n = Ox55aa000055aa0000;
```

Не надо отождествлять *разрядность* целочисленного типа с занимаемым им количеством памяти. Исполняющий код Java может использовать для ваших переменных то количество памяти, которое сочтет нужным, лишь бы только их поведение соответствовало *поведению* типов, заданных вами. Фактически, нынешняя реализация Java из соображений эффективности

хранит переменные типа byte и short в виде 32-битовых значений, поскольку этот размер соответствует машинному слову большинства современных компьютеров (СМ – 8 бит, 8086 – 16 бит, 80386/486 – 32 бит, Pentium – 64 бит).

Ниже приведена таблица разрядностей и допустимых диапазонов для различных типов целых чисел.

Имя	Разрядность	Диапазон
long	64	-9, 223, 372, 036, 854, 775, 808.. 9, 223, 372, 036, 854, 775, 807
Int	32	-2, 147, 483, 648.. 2, 147, 483, 647
Short	16	-32, 768.. 32, 767
byte	8	-128.. 127

Числа с плавающей точкой

Числа с плавающей точкой, часто называемые в других языках вещественными числами, используются при вычислениях, в которых требуется использование дробной части. В Java реализован стандартный (IEEE-754) набор типов для чисел с плавающей точкой — float и double и операторов для работы с ними. Характеристики этих типов приведены в таблице.

Имя	Разрядность	Диапазон
double	64	1. 7e-308.. 1. 7e+ 308
float	32	3. 4e-038.. 3. 4e+ 038

float

В переменных с обычной, или *одинарной точностью,* объявляемых с помощью ключевого слова float, для хранения вещественного значения используется 32 бита.

```
float f;
```

```
float f2 = 3. 14F; // обратите внимание на F, т.к. по умолчанию все литералы double
```

double

В случае *двойной точности,* задаваемой с помощью ключевого слова double, для хранения значений используется 64 бита. Все *трансцендентные* математические функции, такие, как sin, cos, sqrt, возвращают результат типа double.

```
double d;
```

```
double pi = 3. 141592653589793238 46;
```

Приведение типа

Иногда возникают ситуации, когда у вас есть величина какого-то определенного типа, а вам нужно ее присвоить переменной другого типа. Для некоторых типов это можно проделать и без приведения типа, в таких случаях говорят об автоматическом преобразовании типов. В Java автоматическое преобразование возможно только в том случае, когда точности представления чисел переменной-приемника достаточно для хранения исходного значения. Такое преобразование происходит, например, при занесении литеральной константы или значения переменной типа byte или short в переменную типа int. Это называется *расширением (widening)*

или *повышением (promotion),* поскольку тип меньшей разрядности расширяется (повышается) до большего совместимого типа. Размера типа int всегда достаточно для хранения чисел из диапазона, допустимого для типа byte, поэтому в подобных ситуациях оператора явного приведения типа не требуется. Обратное в большинстве случаев неверно, поэтому для занесения значения типа int в переменную типа byte необходимо использовать оператор приведения типа. Эту процедуру иногда называют *сужением (narrowing),* поскольку вы явно сообщаете транслятору, что величину необходимо преобразовать, чтобы она уместилась в переменную нужного вам типа. Для приведения величины к определенному типу перед ней нужно указать этот тип, заключенный в круглые скобки. В приведенном ниже фрагменте кода демонстрируется приведение типа источника (переменной типа int) к типу приемника (переменной типа byte). Если бы при такой операции целое значение выходило за границы допустимого для типа byte диапазона, оно было бы уменьшено путем деления по модулю на допустимый для byte диапазон (результат деления по модулю на число — это остаток от деления на это число).

```
int a = 100;

byte b = (byte) a;
```

Автоматическое преобразование типов в выражениях

Когда вы вычисляете значение выражения, точность, требуемая для хранения промежуточных результатов, зачастую должна быть выше, чем требуется для представления окончательного результата.

```
byte a = 40;

byte b = 50;
```

```
byte c = 100;

int d = a* b / c;
```

Результат промежуточного выражения (a* b) вполне может выйти за диапазон допустимых для типа byte значений. Именно поэтому Java автоматически повышает тип каждой части выражения до типа int, так что для промежуточного результата (a* b) хватает места.

Автоматическое преобразование типа иногда может оказаться причиной неожиданных сообщений транслятора об ошибках. Например, показанный ниже код, хотя и выглядит вполне корректным, приводит к сообщению об ошибке на фазе трансляции. В нем мы пытаемся записать значение 50* 2, которое должно прекрасно уместиться в тип byte, в байтовую переменную. Но из-за автоматического преобразования типа результата в int мы получаем сообщение об ошибке от транслятора — ведь при занесении int в byte может произойти потеря точности.

```
byte b = 50;

b = b* 2;

^ Incompatible type for =. Explicit cast needed to convert int to
byte.

(Несовместимый тип для =. Необходимо явное преобразование int в
byte)

Исправленный текст :
```

```
byte b = 50;

b = (byte) (b* 2);
```

что приводит к занесению в b правильного значения 100.

Если в выражении используются переменные типов byte, short и int, то во избежание переполнения тип всего выражения автоматически повышается до int. Если же в выражении тип хотя бы одной переменной — long, то и тип всего выражения тоже повышается до long. Не забывайте, что все целые литералы, в конце которых не стоит символ L (или l), имеют тип int.

Если выражение содержит операнды типа float, то и тип всего выражения автоматически повышается до float. Если же хотя бы один из операндов имеет тип double, то тип всего выражения повышается до double. По умолчанию Java рассматривает все литералы с плавающей точкой, как имеющие тип double. Приведенная ниже программа показывает, как повышается тип каждой величины в выражении для достижения соответствия со вторым операндом каждого бинарного оператора.

```
class Promote {

public static void main (String args []) { byte b = 42;

char c = 'a';

short s = 1024;

int i = 50000;

float f = 5.67f;
```

```
double d =.1234;
double result = (f* b) + (i/ c) - (d* s);
System. out. println ((f* b)+ "+ "+ (i / c)+ " - " + (d* s));
System. out. println ("result = "+ result);
}
}
```

Подвыражение f* b — это число типа float, умноженное на число типа byte. Поэтому его тип автоматически повышается до float. Тип следующего подвыражения i / c (int, деленный на char) повышается до int. Аналогично этому тип подвыражения d* s (double, умноженный на short) повышается до double. На следующем шаге вычислений мы имеем дело с тремя промежуточными результатами типов float, int и double. Сначала при сложении первых двух тип int повышается до float и получается результат типа float. При вычитании из него значения типа double тип результата повышается до double. Окончательный результат всего выражения — значение типа double.

Символы

Поскольку в Java для представления символов в строках используется кодировка Unicode, разрядность типа char в этом языке — 16 бит. В нем можно хранить десятки тысяч символов интернационального набора символов Unicode. Диапазон типа char — 0..65536. Unicode — это объединение десятков кодировок символов, он включает в себя латинский, греческий, арабский алфавиты, кириллицу и многие другие наборы символов.

```
char c;

char c2 = 0xf132;

char c3 = ' a';

char c4 = '\n';
```

Хотя величины типа char и не используются, как целые числа, вы можете оперировать с ними так, как если бы они были целыми. Это дает вам возможность сложить два символа вместе, или инкрементировать значение символьной переменной. В приведенном ниже фрагменте кода мы, располагая базовым символом, прибавляем к нему целое число, чтобы получить символьное представление нужной нам цифры.

```
int three = 3;

char one = '1';

char four = (char) (three+ one);
```

В результате выполнения этого кода в переменную four заносится символьное представление нужной нам цифры — '4'. Обратите внимание — тип переменной one в приведенном выше выражении повышается до типа int, так что перед занесением результата в переменную four приходится использовать оператор явного приведения типа.

Тип boolean

В языке Java имеется простой тип boolean, используемый для хранения логических значений. Переменные этого типа могут принимать всего два значения — true (истина) и false (ложь).

Значения типа boolean возвращаются в качестве результата всеми операторами сравнения, например (a < b) — boolean — это тип, *требуемый* всеми условными операторами управления — такими, как if, while, do.

```
boolean done = false;
```

В приведенном ниже примере создаются переменные каждого из простых типов и выводятся значения этих переменных.

```
class SimpleTypes {

public static void main(String args []) {

byte b = 0x55;

short s = 0x55ff;

int i = 1000000;

long l = 0xffffffffL;

char c = ' a' ;

float f = .25f;

double d = .00001234;

boolean bool = true;
```

```
System.out.println("byte b = " + b);

System.out.println("short s = " +s);

System.out.println("int i = " + i);

System.out.println("long l = " + l);

System.out.println("char c = " + c);

System.out.println("float f = " + f);

System.out.println("double d = " + d);

System.out.println("boolean bool = " + bool);

} }
```

Запустив эту программу, вы должны получить результат, показанный ниже:

```
C: \> java SimpleTypes

byte b = 85

short s = 22015

int i = 1000000

long l = 4294967295
```

char c = a

float f = 0.25

double d = 1.234e-005

boolean bool = true

Операторы

Операторы в языке Java — это специальные символы, которые сообщают транслятору о том, что вы хотите выполнить операцию с некоторыми операндами. Некоторые операторы требуют одного операнда, их называют унарными. Одни операторы ставятся перед операндами и называются префиксными, другие — после, их называют постфиксными операторами. Большинство же операторов ставят между двумя операндами, такие операторы называются инфиксными бинарными операторами. Существует тернарный оператор, работающий с тремя операндами.

В Java имеется 44 встроенных оператора. Их можно разбить на 4 класса - арифметические, битовые, операторы сравнения и логические.

Арифметические операторы

Арифметические операторы используются для вычислений так же как в алгебре (см. таблицу

со сводкой арифметических операторов ниже). Допустимые операнды должны иметь числовые типы. Например, использовать эти операторы для работы с логическими типами нельзя, а для работы с типом char можно, поскольку в Java тип char — это подмножество типа int.

Оператор	Результат	Оператор	Результат
+	Сложение	+ =	сложение с присваиванием
-	вычитание (также унарный минус)	-=	вычитание с присваиванием
*	Умножение	* =	умножение с присваиванием
/	Деление	/=	деление с присваиванием
%	деление по модулю	%=	деление по модулю с присваиванием
++	Инкремент	--	декремент

Четыре арифметических действия

Ниже, в качестве примера, приведена простая программа, демонстрирующая использование операторов. Обратите внимание на то, что операторы работают как с целыми литералами, так и с переменными.

class BasicMath { public static void int a = 1 + 1;

int b = a * 3;

```
main(String args[]) {

int c = b / 4;

int d = b - a;

int e = -d;

System.out.println("a = " +   a);

System.out.println("b = " +   b);

System.out.println("c = " +   c);

System.out.println("d = " +   d);

System.out.println("e = " +   e);

} }
```

Исполнив эту программу, вы должны получить приведенный ниже результат:

```
C: \> java BasicMath

a = 2

b = 6

c = 1
```

d = 4

e = -4

Оператор деления по модулю

Оператор деления по модулю, или оператор mod, обозначается символом %. Этот оператор возвращает остаток от деления первого операнда на второй. В отличие от C++, функция mod в Java работает не только с целыми, но и с вещественными типами. Приведенная ниже программа иллюстрирует работу этого оператора.

```
class Modulus {

public static void main (String args []) {

int x = 42;

double y = 42.3;

System.out.println("x mod 10 = " + x % 10);

System.out.println("y mod 10 = " + y % 10);

} }
```

Выполнив эту программу, вы получите следующий результат:

```
C:\> Modulus
```

x mod 10 = 2

y mod 10 = 2.3

Арифметические операторы присваивания

Для каждого из арифметических операторов есть форма, в которой одновременно с заданной операцией выполняется присваивание. Ниже приведен пример, который иллюстрирует использование подобной разновидности операторов.

```
class OpEquals {

public static void main(String args[]) {

int a = 1;

int b = 2;

int c = 3;

a += 5;

b *= 4;

c += a * b;

c %= 6;
```

System.out.println("a = " + a);

System.out.println("b = " + b);

System.out.println("c = " + c);

} }

А вот и результат, полученный при запуске этой программы:

C:> Java OpEquals

a = 6

b = 8

c = 3

Инкремент и декремент

В C существует 2 оператора, называемых операторами инкремента и декремента (++ и --) и являющихся сокращенным вариантом записи для сложения или вычитания из операнда единицы. Эти операторы уникальны в том плане, что могут использоваться как в префиксной, так и в постфиксной форме. Следующий пример иллюстрирует использование операторов инкремента и декремента.

class IncDec {

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

int a = 1;

int b = 2;

int c = ++b;

int d = a++;

c++;

System.out.println("a = " + a);

System.out.println("b = " + b);

System.out.println("c = " + c);

System.out.println("d = " + d);

} }
```

Результат выполнения данной программы будет таким:

```
C:\ java IncDec

a = 2

b = 3
```

c = 4

d = 1

Целочисленные битовые операторы

Для целых числовых типов данных — long, int, short, char и byte, определен дополнительный набор операторов, с помощью которых можно проверять и модифицировать состояние отдельных битов соответствующих значений. В таблице приведена сводка таких операторов. Операторы битовой арифметики работают с каждым битом как с самостоятельной величиной.

Оператор	Результат	Оператор	Результат
~	побитовое унарное отрицание (NOT)		
&	побитовое И (AND)	&=	побитовое И (AND) с присваиванием
|	побитовое ИЛИ (OR)	|=	побитовое ИЛИ (OR) с присваиванием
^	побитовое исключающее ИЛИ (XOR)	^=	побитовое исключающее ИЛИ (XOR) с присваиванием
>>	сдвиг вправо	>> =	сдвиг вправо с присваиванием
>>>	сдвиг вправо с заполнением	>>>=	сдвиг вправо с заполнением нулями

	нулями		с присваиванием
<<	сдвиг влево	<<=	сдвиг влево с присваиванием

Пример программы, манипулирующей с битами

В таблице, приведенной ниже, показано, как каждый из операторов битовой арифметики воздействует на возможные комбинации битов своих операндов. Приведенный после таблицы пример иллюстрирует использование этих операторов в программе на языке Java.

A	B	OR	AND	XOR	NOT A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

```
class Bitlogic {

public static void main(String args []) {

String binary[] = { "OOOO", "0001", "0010", "0011", "0100", "0101", "0110", "0111", "1000",
"1001",     "1010", "1011", "1100", "1101",

"1110", "1111" };

int a = 3;  //    0+2+1 или двоичное 0011
```

```
int b = 6;  //    4+2+0 или двоичное 0110

int c = a | b;

int d = a & b;

int e = a ^ b;

int f = (~a & b) | (a & ~b);

int g = ~a & 0x0f;

System.out.println(" a = " + binary[a]);

System.out.println(" b = " + binary[b]);

System.out.println(" ab = " + binary[c]);

System.out.println(" a&b = " + binary[d]);

System.out.println(" a^b = " + binary[e]);

System.out.println("~a&b|a^~Ь = " + binary[f]);

System.out.println(" ~a = " + binary[g]);

} }
```

Ниже приведен результат, полученный при выполнении этой программы:

C: \> Java BitLogic

a = 0011

b = 0110

a | b = 0111

a & b = 0010

a ^ b = 0101

~a & b | a & ~b = 0101

~a = 1100

Сдвиги влево и вправо

Оператор << выполняет сдвиг влево всех битов своего левого операнда на число позиций, заданное правым операндом. При этом часть битов в левых разрядах выходит за границы и теряется, а соответствующие правые позиции заполняются нулями. В предыдущей главе уже говорилось об автоматическом повышении типа всего выражения до int в том случае если в выражении присутствуют операнды типа int или целых типов меньшего размера. Если же хотя бы один из операндов в выражении имеет тип long, то и тип всего выражения повышается до long.

Оператор >> означает в языке Java сдвиг вправо. Он перемещает все биты своего левого

операнда вправо на число позиций, заданное правым операндом. Когда биты левого операнда выдвигаются за самую правую позицию слова, они теряются. При сдвиге вправо освобождающиеся старшие (левые) разряды сдвигаемого числа заполняются предыдущим содержимым знакового разряда. Такое поведение называют расширением знакового разряда.

В следующей программе байтовое значение преобразуется в строку, содержащую его шестнадцатиричное представление. Обратите внимание - сдвинутое значение приходится маскировать, то есть логически умножать на значение 0x0f, для того, чтобы очистить заполняемые в результате расширения знака биты и понизить значение до пределов, допустимых при индексировании массива шестнадцатиричных цифр.

```
class HexByte {

static public void main(String args[]) {

char hex[] = { '0', '1, '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f };

byte b = (byte) 0xf1;

System.out.println("b = 0x" + hex[(b >> 4) & 0x0f] +       hex[b & 0x0f]);

} }
```

Ниже приведен результат работы этой программы:

```
C:\> java HexByte

b = 0xf1
```

Беззнаковый сдвиг вправо

Часто требуется, чтобы при сдвиге вправо расширение знакового разряда не происходило, а освобождающиеся левые разряды просто заполнялись бы нулями.

```
class ByteUShift  {

static public void main(String args[]) {

char hex[] = { '0', '1', '2', '3', '4',  '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd',   'e', 'f' };

byte b = (byte) 0xf1;

byte c = (byte) (b >> 4);

byte d = (byte) (b >> 4);

byte e = (byte) ((b & 0xff) >> 4);

System.out.println(" b = 0x" + hex(b >> 4) & 0x0f] + hex[b & 0x0f]);

System.out.println(“ b >> 4 = 0x" + hex[(c >> 4) & 0x0f] + hex[c & 0x0f]);

System.out.println(“b >>> 4 = 0x" + hex[(d >> 4) & 0x0f] + hex[d & 0x0f]);

System.out.println(“(b & 0xff) >> 4 = 0x" + hex[(e >> 4) & 0x0f] + hex[e & 0x0f]);

} }
```

Для этого примера переменную b можно было бы инициализировать произвольным отрицательным числом, мы использовали число с шестнадцатиричным представлением 0xf1. Переменной c присваивается результат знакового сдвига b вправо на 4 разряда. Как и ожидалось, расширение знакового разряда приводит к тому, что 0xf1 превращается в 0xff. Затем в переменную d заносится результат беззнакового сдвига b вправо на 4 разряда. Можно было бы ожидать, что в результате d содержит 0x0f, однако на деле мы снова получаем 0xff. Это — результат расширения знакового разряда, выполненного при автоматическом повышении типа переменной b до int перед операцией сдвига вправо. Наконец, в выражении для переменной e нам удается добиться желаемого результата — значения 0x0f. Для этого нам пришлось перед сдвигом вправо логически умножить значение переменной b на маску 0xff, очистив таким образом старшие разряды, заполненные при автоматическом повышении типа. Обратите внимание, что при этом уже нет необходимости использовать беззнаковый сдвиг вправо, поскольку мы знаем состояние знакового бита после операции AND.

C: \> java ByteUShift

b = 0xf1

b >> 4 = 0xff

b >>> 4 = 0xff

b & 0xff) >> 4 = 0x0f

Битовые операторы присваивания

Так же, как и в случае арифметических операторов, у всех бинарных битовых операторов есть

родственная форма, позволяющая автоматически присваивать результат операции левому операнду. В следующем примере создаются несколько целых переменных, с которыми с помощью операторов, указанных выше, выполняются различные операции.

```
class OpBitEquals {

public static void main(String args[]) {

int a = 1;

int b = 2;

int c = 3;

a |= 4;

b >>= 1;

c <<= 1;

a ^= c;

System.out.println("a = " + a);

System.out.println("b = " + b);

System.out.println("c = " + c);

} }
```

Результаты исполнения программы таковы:

C:\> Java OpBitEquals

a = 3

b = 1

c = 6

Операторы отношения

Для того, чтобы можно было сравнивать два значения, в Java имеется набор операторов, описывающих отношение и равенство. Список таких операторов приведен в таблице.

Оператор	Результат
==	равно
!=	не равно
>	больше
<	меньше
>=	больше или равно
<=	меньше или равно

Значения любых типов, включая целые и вещественные числа, символы, логические значения и

ссылки, можно сравнивать, используя оператор проверки на равенство == и неравенство !=. Обратите внимание — в языке Java, так же, как в C и C++ проверка на равенство обозначается последовательностью (==). Один знак (=) — это оператор присваивания.

Булевы логические операторы

Булевы логические операторы, сводка которых приведена в таблице ниже, оперируют только с операндами типа boolean. Все бинарные логические операторы воспринимают в качестве операндов два значения типа boolean и возвращают результат того же типа.

Оператор	Результат	Оператор	Результат
&	логическое И (AND)	&=	И (AND) с присваиванием
|	логическое ИЛИ (OR)	=	ИЛИ (OR) с присваиванием
^	логическое исключающее ИЛИ (XOR)	^=	исключающее ИЛИ (XOR) с присваиванием
||	оператор OR быстрой оценки выражений (short circuit OR)	==	равно
&&	оператор AND быстрой оценки выражений (short circuit AND)	!=	не равно

!	логическое унарное отрицание (NOT)	?:	тернарный оператор if-then-else

Результаты воздействия логических операторов на различные комбинации значений операндов показаны в таблице.

A	B	OR	AND	XOR	NOT A
false	false	false	false	false	true
true	false	true	false	true	false
false	true	true	false	true	true
true	true	true	true	false	false

Программа, приведенная ниже, практически полностью повторяет уже знакомый вам пример BitLogic. Только но на этот раз мы работаем с булевыми логическими значениями.

```
class BoolLogic {

public static void main(String args[]) {

boolean a = true;

boolean b = false;

boolean c = a | b;

boolean d = a & b;
```

```
boolean e = a ^ b;
boolean f = (!a & b) | (a & !b);
boolean g = !a;
System.out.println(" a = " + a);
System.out.println(" b = " + b);
System.out.println(" a|b = " + c);
System.out.println(" a&b = " + d);
System.out.println(" a^b = " + e);
System.out.println("!a&b|a&!b = " + f);
System.out.println(" !a = " + g);
} }
C: \> Java BoolLogic
a = true
b = false
a|b = true
```

a&b = false

a^b = true

!a&b|a&!b = true

!a = false

Операторы быстрой оценки логических выражений (short circuit logical operators)

Существуют два интересных дополнения к набору логических операторов. Это — альтернативные версии операторов AND и OR, служащие для быстрой оценки логических выражений. Вы знаете, что если первый операнд оператора OR имеет значение true, то независимо от значения второго операнда результатом операции будет величина true. Аналогично в случае оператора AND, если первый операнд — false, то значение второго операнда на результат не влияет — он всегда будет равен false. Если вы в используете операторы && и || вместо обычных форм & и |, то Java не производит оценку правого операнда логического выражения, если ответ ясен из значения левого операнда. Общепринятой практикой является использование операторов && и || практически во всех случаях оценки булевых логических выражений. Версии этих операторов & и | применяются только в битовой арифметике.

Тернарный оператор if-then-else

Общая форма оператора if-then-use такова:

выражение1? выражение2: выражение3

В качестве первого операнда — «выражение1» — может быть использовано любое выражение, результатом которого является значение типа boolean. Если результат равен true, то выполняется оператор, заданный вторым операндом, то есть, «выражение2». Если же первый операнд равен false, то выполняется третий операнд — «выражение3». Второй и третий операнды, то есть «выражение2» и «выражение3», должны возвращать значения одного типа и не должны иметь тип void.

В приведенной ниже программе этот оператор используется для проверки делителя перед выполнением операции деления. В случае нулевого делителя возвращается значение 0.

```
class Ternary {

public static void main(String args[]) {

int a = 42;

int b = 2;

int c = 99;

int d = 0;

int e = (b == 0) ? 0 : (a / b);

int f = (d == 0) ? 0 : (c / d);

System.out.println("a = " + a);
```

```
System.out.println("b = " + b);

System.out.println("c = " + c);

System.out.println("d = " + d);

System.out.println("a / b = " + e);

System.out.println("c / d = " + f);

} }
```

При выполнении этой программы исключительной ситуации деления на нуль не возникает и выводятся следующие результаты:

```
C: \>java Ternary

a = 42

b = 2

c = 99

d = 0

a / b = 21

c / d  = 0
```

Приоритеты операторов

В Java действует определенный порядок, или приоритет, операций. В элементарной алгебре нас учили тому, что у умножения и деления более высокий приоритет, чем у сложения и вычитания. В программировании также приходится следить и за приоритетами операций. В таблице указаны в порядке убывания приоритеты всех операций языка Java.

Высший			
( )	[ ]	.	
~	!		
*	/	%	
+	-		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
Низший			

В первой строке таблицы приведены три необычных оператора, о которых мы пока не

говорили. Круглые скобки () используются для явной установки приоритета. Как вы узнали из предыдущей главы, квадратные скобки [] используются для индексирования переменной-массива. Оператор . (точка) используется для выделения элементов из ссылки на объект — об этом мы поговорим в главе 7. Все же остальные операторы уже обсуждались в этой главе.

Явные приоритеты

Поскольку высший приоритет имеют круглые скобки, вы всегда можете добавить в выражение несколько пар скобок, если у вас есть сомнения по поводу порядка вычислений или вам просто хочется сделать свои код более читабельным.

a >> b + 3

Какому из двух выражений, a >> (b + 3) или (a >> b) + 3, соответствует первая строка? Поскольку у оператора сложения более высокий приоритет, чем у оператора сдвига, правильный ответ — a>> (b + a). Так что если вам требуется выполнить операцию (a>>b)+ 3 без скобок не обойтись.

Операторы управления потоком выполнения Java программы

Как вы знаете, любой алгоритм, предназначенный для выполнения на компьютере, можно разработать, используя только линейные вычисления, разветвления и циклы.

Всякий язык программирования должен иметь средства записи алгоритмов. Они называются *операторами* (statements) языка. Минимальный набор операторов должен содержать оператор для записи линейных вычислений, условный оператор для записи разветвлении и оператор цикла.

Обычно состав операторов языка программирования шире: для удобства записи алгоритмов в язык включаются несколько операторов цикла, оператор варианта, операторы перехода, операторы описания объектов.

Набор операторов языка Java включает:

- операторы описания переменных и других объектов
- операторы-выражения;
- операторы присваивания;
- условный оператор if;
- три оператора цикла while, do-while, for;

- оператор варианта switch;
- Операторы перехода break, continue и return;
- блок {};
- пустой оператор — просто точка с запятой.

Здесь приведен основной набор операторов Java.

Замечание

В языке Java нет оператора goto.

Всякий оператор завершается точкой с запятой.

Можно поставить точку с запятой в конце любого выражения, и оно станет оператором (expression statement). Но смысл это имеет только для операций присваивания, инкремента и декремента и вызовов методов. В остальных случаях это бесполезно, потому что вычисленное значение выражения потеряется.

Линейное выполнение алгоритма обеспечивается последовательной записью операторов. Переход со строки на строку в исходном тексте не имеет никакого значения для компилятора, он осуществляется только для наглядности и читаемости текста.

Блок

Блок заключает в себе нуль или несколько операторов с целью использовать их как один оператор в тех местах, где по правилам языка можно записать только один оператор. Например, {x = 5; y = ?;}. Можно записать и пустой блок, просто пару фигурных скобок {}.

Блоки операторов часто используются для ограничения области действия переменных и просто для улучшения читаемости текста программы.

Операторы присваивания

Точка с запятой в конце любой операции присваивания превращает ее в оператор присваивания. Побочное действие операции — присваивание — становится в операторе основным.

Разница между операцией и оператором присваивания носит лишь теоретический характер. Присваивание чаще используется как оператор, а не операция.

Условный оператор

Условный оператор (if-then-else statement) в языке Java записывается так:

if (логВыр) оператор1 else оператор2

и действует следующим образом. Сначала вычисляется логическое выражение

логвыр. Если результат true, то действует *оператор!* и на этом действие условного оператора завершается, *оператор2* не действует, далее будет выполняться следующий за if оператор. Если результат false, то действует *оператор2,* при этом оператор,! вообще не выполняется.

Условный оператор может быть сокращенным (if-then statement):

if (логВыр) оператор!

и в случае false не выполняется ничего.

Синтаксис языка не позволяет записывать несколько операторов ни в ветви then, ни в ветви else. При необходимости составляется блок операторов в фигурных скобках. Соглашения "Code Conventions" рекомендуют всегда использовать фигурные скобки и размещать оператор на нескольких строках с отступами, как в следующем примере:

```
if (a < x) {

   x = a + b;
 }
else {

   x = a — b;

}
```

Это облегчает добавление операторов в каждую ветвь при изменении алгоритма. Мы не будем строго следовать этому правилу, чтобы не увеличивать объем книги.

Очень часто одним из операторов является снова условный оператор, например:

```
if (n == 0) {

    sign = 0;

}
else if (n < 0){

   sign = -1;

} else {
   sign = 1;

 }
```

При этом может возникнуть такая ситуация ("dangling else"):

```
int ind = 5, x = 100;

if (ind >= 10) if (ind <= 20) x = 0; else x = 1;
```

Сохранит переменная x значение 0 или станет равной 1? Здесь необходимо волевое решение, и общее для большинства языков, в том числе и Java,. правило таково: ветвь

else относится к ближайшему слева услдвию if, не имеющему своей ветви else. Поэтому в нашем примере переменная x останется равной 0.

Изменить этот порядок можно с помощью блока:

```
if (ind > 10) {if (ind < 20) x = 0; else x = 1;}
```

Вообще не стоит увлекаться сложными вложенными условными операторами. Проверки условий занимают много времени. По возможности лучше использовать логические операции, например, в нашем примере можно написать

```
if (ind >= 10 && ind <= 20) x = 0; else x = 1;
```

В приведенном ниже листинге вычисляются корни квадратного уравнения $ax^2 + bx + c = 0$ для любых коэффициентов, в том числе и нулевых.

Листинг Вычисление корней квадратного уравнения

```
class QuadraticEquation{

public static void main(String[] args){

double a = 0.5, Ъ = -2.7, c = 3.5, d, eps=le-8;

if (Math.abs(a) < eps)
```

```
if (Math.abs(b) < eps)

if (Math.abs(c) < eps) // Все коэффициенты равны нулю

System.out.println("Решение —любое число");

else

System.out.println("Решений нет");

else

System.out.println("xl = x2 = " +(-c / b) ) ;

else { // Коэффициенты не равны нулю

if((d = b**b — 4*a*c)< 0.0){ // Комплексные корни

d = 0.5 * Math.sqrt(-d) / a;

a = -0.5 * b/ a;

System.out.println("xl = " +a+ " +i " +d+

",x2 = " +a+ " -i " +d);

} else {
```

```
// Вещественные корни

d =0.5 * Math.sqrt(d) / a;

a = -0.5 * b / a;

System.out.println("x1 = " + (a + d) + ", x2 = " +(a - d));

}

}

)

}
```

В этой программе использованы методы вычисления модуля abs() и квадратного корня sqrt о вещественного числа из встроенного в Java API класса Math. Поскольку все вычисления с вещественными числами производятся приближенно, мы считаем, что коэффициент уравнения равен нулю, если его модуль меньше 0,00000001. Обратите внимание на то, как в методе println о используется сцепление строк, и на то, как операция присваивания при вычислении дискриминанта вложена в логическое выражение.

Операторы цикла

Основной оператор цикла — оператор while — выглядит так:

while (логВыр) оператор

Вначале вычисляется логическое выражение *логВыр;* если его значение true, то выполняется оператор, образующий цикл. Затеем снова вычисляется *лог-выр* и действует оператор, и так до тех пор, пока не получится значение false. Если *логВыр* изначально равняется false, то *оператор* не будет выполнен ни разу. Предварительная проверка обеспечивает безопасность выполнения цикла, позволяет избежать переполнения, деления на нуль и других неприятностей. Поэтому оператор while является основным, а в некоторых языках и единственным оператором цикла.

Оператор в цикле может быть и пустым, например, следующий фрагмент кода:

```
int i = 0;

double s = 0.0;

while ((s += 1.0 / ++i) < 10);
```

вычисляет количество i сложений, которые необходимо сделать, чтобы гармоническая сумма s достигла значения 10. Такой стиль характерен для языка C. Не стоит им увлекаться, чтобы не превратить текст программы в шифровку, на которую вы сами через пару недель будете смотреть с недоумением.

Можно организовать и бесконечный цикл:

while (true) *оператор*

Конечно, из такого цикла следует предусмотреть какой-то выход, например, оператором break, как в листинге 1.5. В противном случае программа зациклится, и вам придется прекращать ее выполнение "комбинацией из трех пальцев" <Ctrl>+<Alt>+<Del> в MS Windows 95/98/ME, комбинацией <Ctrl>+<C> в UNIX или через Task Manager в Windows NT/2000.

Если в цикл надо включить несколько операторов, то следует образовать блок операторов {}.

Второй оператор цикла — оператор do-while — имеет вид do оператор while *(логВыр)*

Здесь сначала выполняется оператор, а потом происходит вычисление логического выражения логвыр. Цикл выполняется, пока логвыр остается равным true.

Существенное различие между этими двумя операторами цикла только в том, что в цикле do-while оператор обязательно выполнится хотя бы один раз.

Например, пусть задана какая-то функция f(x), имеющая на отрезке,[a,b] ровно один корень. В листинге приведена программа, вычисляющая этот корень приближенно методом деления пополам (бисекции, дихотомий).

Листинг Нахождение корня нелинейного уравнения методом бисекции

```
class Bisection{
static double f(double x){
return x*x*x — 3*x*x +3; // Или что-то другое
}
public static void main(String!] args){
double a = 0.0, b = 1,5, c, y, eps = le-8;
do{
c = 0.5 *(a + b); y = f(c);
if (Math.abs(y) < eps) break;
// Корень найден. Выходим из цикла
// Если на концах отрезка [a; c]
// функция имеет разные знаки:
if (f (a) * y < 0.0) b = c;
// Значит, корень здесь. Переносим точку b в точку c
```

```
//В противном случае:

else a * c;

// Переносим точку a в точку c

// Продолжаем, пока отрезок [a; Ь] не станет мал

} while (Math, abs (b-a) >= eps);

System.out.println("x = " +c+ ", f(" +c+ ") = " +y) ;

}

}
```

Класс Bisection сложнее предыдущих примеров: в нем кроме метода main () есть еще метод вычисления функции f(x). Здесь метод f о очень прост: он вычисляет значение многочлена и возвращает его в качестве значения функции, причем все это выполняется одним оператором:

return выражение

В методе main о появился еще один новый оператор break, который просто прекращает выполнение цикла, если мы по счастливой случайности наткнулись на приближенное значение корня. Внимательный читатель заметил и появление

модификатора static в объявлении метода f(). Он необходим потому, что метод f o вызывается из статического метода main o.

Третий оператор цикла — оператор for — выглядит так:

for (*списокВыр* ; логНьр; слисокВыр2) *оператор*

Перед выполнением цикла вычисляется список выражений *списокВыр1.* Это нуль или несколько выражений, перечисленных через запятую. Они вычисляются слева направо, и в следующем выражении уже можно использовать результат предыдущего выражения. Как правило, здесь задаются начальные значения переменным цикла.

Затем вычисляется логическое выражение логвьр. Если оно истинно, true, то действует оператор, потом вычисляются слева направо выражения из списка выражений *списокВыр2.* Далее снова проверяется *логвыр.* Если оно ис тинно, то выполняется оператор и *списокВыр2* и т. д. Как только логйыр станет равным false, выполнение цикла заканчивается.

Короче говоря, выполняется последовательность операторов

списокВыр1; while (логВыр){

оператор

слисокВыр2; }

с тем исключением, что, если оператором в цикле является оператор

continue, то список_выр2 все-таки выполняется.

Вместо *списокВыр1* может стоять одно определение переменных обязательно с начальным значением. Такие переменные известны только в пределах этого цикла.

Любая часть оператора for может отсутствовать: цикл может быть пустым, выражения в заголовке тоже, при этом точки с запятой сохраняются. Можно задать бесконечный цикл:

for (;;) оператор

В этом случае в теле цикла следует предусмотреть какой-нибудь выход.

Хотя в операторе for заложены большие возможности, используется он, главным образом, для перечислений, когда их число известно, например, фрагмент кода ,

```
int s=0;

for (int k = 1; k <= N; k++) s += k * k;

// Здесь переменная k уже неизвестна
```

вычисляет сумму квадратов первых N натуральных чисел.

Оператор continue и метки

Оператор **continue** используется только в операторах цикла. Он имеет две формы. Первая форма состоит только из слова continue и осуществляет немедленный переход к следующей итерации цикла. В очередном фрагменте кода оператор continue позволяет обойти деление на нуль:

```
for (int i = 0; i < N; i++){

if (i '== j) continue;

s += 1.0 / (i - j);

}
```

Вторая форма содержит метку:

continue метка

метка записывается, как все идентификаторы, из букв Java, цифр и знака подчеркивания, но не требует никакого описания. Метка ставится перед оператором или открывающей фигурной скобкой и отделяется от них двоеточием. Так получается *помеченный оператор* или *помеченный блок.*

Вторая форма используется только в случае нескольких вложенных циклов для немедленного перехода к очередной итерации одного из объемлющих циклов, а именно, помеченного цикла.

Оператор break

Оператор break используется в операторах цикла и операторе варианта для немедленного выхода из этих конструкций.

Оператор break метка

применяется внутри помеченных операторов цикла, оператора варианта или помеченного блока для немедленного выхода за эти операторы. Следующая схема поясняет эту конструкцию.

MI: { // Внешний блок

M2: { // Вложенный блок — второй уровень

M3: { // Третий уровень вложенности...

if *(что-то случилось)* break M2;

// Если true, то здесь ничего не выполняется

}

// Здесь тоже ничего не выполняется

}

// Сюда передается управление

}

Поначалу сбивает с толку то обстоятельство, что метка ставится перед блоком или оператором, а управление передается за этот блок или оператор. Поэтому не стоит увлекаться оператором break с меткой.

Оператор варианта

Оператор варианта switch организует разветвление по нескольким направлениям. Каждая ветвь отмечается константой или константным выражением какого-либо целого типа (кроме long) и выбирается, если значение определенного выражения совпадет с этой константой. Вся конструкция выглядит так.

switch (целВыр){

case констВыр1: оператор1

case констВыр2: оператор2

. . . . .

case констВырN: операторN

default: операторDef

}

Стоящее в скобках выражение *целвыр* может быть типа byte, short, int, char, но не long. Целые числа или целочисленные выражения, составленные из констант, *константыр* тоже не должны иметь тип long.

Оператор варианта выполняется так. Все константные выражения вычисляются заранее, на этапе компиляции, и должны иметь отличные друг от друга значения. Сначала вычисляется целочисленное выражение ;целйыр. Если. ОНО совпадает с одной из констант, то выполняется оператор, отмеченный этой константой. Затем выполняются ("fall through labels") все следующие операторы, включая и *операторОе£,* и работа оператора вариайтазаканчивается.

Если же ни одна константа не равна значению выражения, то выполняется *операторОе£* и все следующие за ним операторы. Поэтому ветвь default должна записываться последней. Ветвь default может отсутствовать, тогда в этой ситуации оператор варианта вообще ничего не делает.

Таким образом, константы в вариантах case играют роль только меток, точек входа в оператор варианта, а далее выполняются все оставшиеся операторы в порядке их записи.

Знатокам Pascal

После выполнения одного варианта оператор switch продолжает выполнять все

оставшиеся варианты.

Чаще всего необходимо "пройти" только одну ветвь операторов. В таком случае используется оператор break, сразу же прекращающий выполнение оператора switch. Может понадобиться выполнить один и тот же оператор в разных ветвях case. В этом случае ставим несколько меток case подряд. Вот простой пример.

```
switch(dayOfWeek){

case 1: case 2: case 3: case 4:case 5:

System.out.println("Week-day");, break;

case 6: case 7:

System.out.println("Week-end"); break;

default:

System.out.printlnt"Unknown day");

}
```

Замечание

Не забывайте завершать варианты оператором break.

Массивы

Как всегда в программировании *массив* — это совокупность переменных одного типа, хранящихся в смежных ячейках оперативной памяти.

Массивы в языке Java относятся к ссылочным типам и описываются своеобразно, но характерно для ссылочных типов. Описание производится в три этапа.

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

double[] a, b;

Здесь определены две переменные — ссылки а и ь на массивы типа double. Можно поставить квадратные скобки и непосредственно после имени. Это удобно делать среди определений обычных переменных:

int I = 0, ar[], k = -1;

Здесь определены две переменные целого типа i и k, и объявлена ссылка на целочисленный массив аг.

Второй этап — *определение* (installation). На этом этапе указывается количество

элементов массива, называемое его *длиной,* выделяется место для массива в оперативной памяти, переменная-ссылка получает адрес массива. Все эти действия производятся еще одной операцией языка Java — операцией new *тип,* выделяющей участок в оперативной памяти для объекта указанного в операции типа и возвращающей в качестве результата адрес этого участка. Например,

a = new double[5];

b = new double[100];

ar = new int[50];

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

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

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

for (int i = 0; i < 100; i++) b[i] = 1.0 /i;

for (int i = 0; i < 50; i++) ar[i] = 2 * i + 1;

Первые два этапа можно совместить:

doublet] a = new double[5], b = new double[100];

int i = 0, ar[] = new int[50], k = -1;

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

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

Можно совместить второй и третий этап:

a = new doublet] {0.1, 0.2, -0.3, 0.45, -0.02};

Можно даже создать безымянный массив, сразу же используя результат операции new, например, так:

System.out.println(new char[] {'H', 'e', '1', '1', 'o'});

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

переменных типа double и содержат один и тот же адрес.

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

ar = null;

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

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

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

Кроме ссылки на массив, для каждого массива автоматически определяется целая константа с одним и тем же именем length. Она равна длине массива. Для каждого массива имя этой константы уточняется именем массива через точку. Так, после наших определений, константа a.length равна 5, константа b. length равна 100, а ar. length равна 50.

Последний элемент массива а можно записать так: a [a. length - 1], предпоследний — a

[a. length - 2] и т. д. Элементы массива обычно перебираются в цикле вида:

```
double aMin = a[0], aMax = aMin;

for (int i = 1; i < a.length; i++){

if <a[i] < aMin) aMin = a[i];

if (a[i] > aMax) aMax = a[i];

}

double range = aMax — aMin;
```

Здесь вычисляется диапазон значений массива.

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

(a[2] + a[4]) / a[0] и т. д.

Многомерные массивы

Элементами массивов в Java могут быть снова массивы. Можно объявить:

```
char[] [] c;
```

что эквивалентно

char c[] c[];

или

char c[][];

Затем определяем внешний массив:

c = new char[3][];

Становится ясно, что с — массив, состоящий из трех элементов-массивов. Теперь определяем его элементы-массивы:

c[0] = new char[2];

c[1] = new char[4];

c[2] = new char[3];

После этих определений переменная c.length равна з, c[0] .length равна 2,

c[l].length равна 4 и c[2.length равна 3.

Наконец, задаем начальные значения **c [0] [0] = 'a', c[0][1] = 'r',**

c[1][0] = 'r',c[1][1] = 'a',c[1][2] = 'y' и т.д.

Замечание

Двумерный массив в Java не обязан быть прямоугольным.

Описания можно сократить:

int[] [] d = new int[3] [4];

А начальные значения задать так:

int[][] inds = {{I, 2, 3}, {4, 5, 6}};

В листинге приведен пример программы, вычисляющей первые 10 строк треугольника Паскаля, заносящей их в треугольный массив и выводящей его элементы на экран. Рис. показывает вывод этой программы.

Листинг Треугольник Паскаля

```
class PascalTriangle{

public static final int LINES = 10; // Так определяются констан

public static void main(String[] args) {

int[][] p, = new int [LINES] [];
```

```
p[0] = new int[1];
System, out. println (p [0] [0] = 1);
p[l] = new int[2];
p[l][0] = p[l][1] = 1;
System.out.println(p[1][0] + " " + p[l][l]);
for (int i = 2; i < LINES; i++){
p[i] = new int[i+l];
System.out.print((p[i][0] = 1) + " ");
for (int j = 1; j < i; j++)
System.out. print ( (p[i] [j] =p[i-l][j-l] -bp[i-l][j]) + " ");
System, out. println (p [ i] [i] = 1)
}
}
}
```

Как можно «получить» «случайные» числа:

```
Random rand = new Random();

z[i]=(int) (100 * rand.nextDouble());
```

Работа со строками

В языках C и C++ отсутствует встроенная поддержка такого объекта, как строка. В них при необходимости передается адрес последовательности байтов, содержимое которых трактуется как символы до тех пор, пока не будет встречен нулевой байт, отмечающий конец строки. В пакет java.lang встроен класс, инкапсулирующий структуру данных, соответствующую строке. Этот класс, называемый **String**, не что иное, как объектное представление неизменяемого символьного массива. В этом классе есть методы, которые позволяют сравнивать строки, осуществлять в них поиск и извлекать определенные символы и подстроки. Класс **StringBuffer** используется тогда, когда строку после создания требуется изменять.

ВНИМАНИЕ

И String, и StringBuffer объявлены final, что означает, что ни от одного из этих классов нельзя производить подклассы. Это было сделано для того, чтобы можно было применить некоторые виды оптимизации позволяющие увеличить производительность при выполнении операций обработки строк.

Конструкторы

Как и в случае любого другого класса, вы можете создавать объекты типа String с помощью оператора new. Для создания пустой строки используется конструктор без параметров:

```
String s = new String():
```

Приведенный ниже фрагмент кода создает объект s типа String инициализируя его строкой из трех символов, переданных конструктору в качестве параметра в символьном массиве.

```
char chars[] = { 'a', 'b', 'c' }:
```

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

System.out.println(s):
```

Этот фрагмент кода выводит строку «abc». Итак, у этого конструктора — 3 параметра:

```
String(char chars[], int начальныйИндекс, int числоСимволов);
```

Используем такой способ инициализации в нашем очередном примере:

```
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' }:

String s = new String(chars,2,3);

System.out.println(s);
```

Этот фрагмент выведет «cde».

Специальный синтаксис для работы со строками

В Java включено несколько приятных синтаксических дополнений, цель которых — помочь программистам в выполнении операций со строками. В числе таких операций создание объектов типа String слияние нескольких строк и преобразование других типов данных в символьное представление.

Создание строк

Java включает в себя стандартное сокращение для этой операции — запись в виде литерала, в которой содержимое строки заключается в пару двойных кавычек. Приводимый ниже фрагмент кода эквивалентен одному из предыдущих, в котором строка инициализировалась массивом типа char.

```
String s = "abc";

System.out.println(s);
```

Один из общих методов, используемых с объектами String — метод length, возвращающий число символов в строке. Очередной фрагмент выводит число 3, поскольку в используемой в нем строке — 3 символа.

```
String s = "abc";

System.out.println(s.length);
```

В Java интересно то, что для каждой строки-литерала создается свой представитель класса String, так что вы можете вызывать методы этого класса непосредственно со строками-литералами, а не только со ссылочными переменными. Очередной пример также выводит число 3.

```
System.out.println("abc".Length());
```

Слияние строк

Строку

```
String s = «He is » + age + " years old.";
```

в которой с помощью оператора + три строки объединяются в одну, прочесть и понять безусловно легче, чем ее эквивалент, записанный с явными вызовами тех самых методов, которые неявно были использованы в первом примере:

```
String s = new StringBuffer("He is ").append(age);

s.append(" years old.").toString();
```

По определению каждый объект класса String не может изменяться. Нельзя ни вставить новые символы в уже существующую строку, ни поменять в ней одни символы на другие. И добавить одну строку в конец другой тоже нельзя. Поэтому транслятор Java преобразует операции, выглядящие, как модификация объектов String, в операции с родственным классом StringBuffer.

ЗАМЕЧАНИЕ

Все это может показаться вам необоснованно сложным. А почему нельзя обойтись одним классом String, позволив ему вести себя примерно так же, как StringBuffer? Все дело в производительности. Тот факт, что объекты типа String в Java неизменны, позволяет транслятору применять к операциям с ними различные способы оптимизации.

Последовательность выполнения операторов

Давайте еще раз обратимся к нашему последнему примеру:

```
String s = "He is " + age + " years old.";
```

В том случае, когда age — не String, а переменная, скажем, типа int, в этой строке кода заключено еще больше магии транслятора. Целое значение переменной int передается совмещенному методу append класса StringBuffer, который преобразует его в текстовый вид и добавляет в конец содержащейся в объекте строки. Вам нужно быть внимательным при совместном использовании целых выражений и слияния строк, в противном случае результат может получиться совсем не тот, который вы ждали. Взгляните на следующую строку:

```
String s = "four: " + 2 + 2;
```

Быть может, вы надеетесь, что в s будет записана строка «four: 4»? Не угадали — с вами сыграла злую шутку последовательность выполнения операторов. Так что в результате получается "four: 22".

Для того, чтобы первым выполнилось сложение целых чисел, нужно использовать скобки :

```
String s = "four: " + (2 + 2);
```

Преобразование строк

В каждом классе String есть метод toString — либо своя собственная реализация, либо вариант по умолчанию, наследуемый от класса Object. Класс в нашем очередном примере замещает наследуемый метод toStrring своим собственным, что позволяет ему выводить значения переменных объекта.

```
class Point {
int x, y;
Point(int x, int y) {
this.x = x;
this.y = y;
}
public String toString() {
return "Point[" + x + ", " + y + "]";
```

```
} }

class toStringDemo {

public static void main(String args[]) {

Point p = new Point(10, 20);

System.out.println("p = " + p);

} }
```

Ниже приведен результат, полученный при запуске этого примера.

C:\> Java toStringDemo

p = Point[10, 20]

Извлечение символов

Для того, чтобы извлечь одиночный символ из строки, вы можете сослаться непосредственно на индекс символа в строке с помощью метода charAt. Если вы хотите в один прием извлечь несколько символов, можете воспользоваться методом getChars. В приведенном ниже фрагменте показано, как следует извлекать массив символов из объекта типа String.

```
class getCharsDemo {
```

```
public static void main(String args[]) {
String s = "This is a demo of the getChars method.";
int start = 10;
int end = 14;
char buf[] = new char[end - start];
s.getChars(start, end, buf, 0);
System.out.println(buf);
} }
```

Обратите внимание — метод getChars не включает в выходной буфер символ с индексом end. Это хорошо видно из вывода нашего примера — выводимая строка состоит из 4 символов.

C:\> java getCharsDemo

```
demo
```

Для удобства работы в String есть еще одна функция — toCharArray, которая возвращает в выходном массиве типа char всю строку. Альтернативная форма того же самого механизма позволяет записать содержимое строки в массив типа byte, при этом значения старших байтов в 16-битных символах отбрасываются. Соответствующий метод называется getBytes, и его параметры имеют тот же смысл, что и параметры getChars, но с единственной разницей — в качестве третьего параметра надо использовать

массив типа byte.

Сравнение

Если вы хотите узнать, одинаковы ли две строки, вам следует воспользоваться методом equals класса String. Альтернативная форма этого метода называется equalsIgnoreCase, при ее использовании различие регистров букв в сравнении не учитывается. Ниже приведен пример, иллюстрирующий использование обоих методов:

```
class equalDemo {

public static void main(String args[]) {

String s1 = "Hello";

String s2 = "Hello";

String s3 = "Good-bye";

String s4 = "HELLO";

System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));

System.out.println(s1 + " equals " + s3 + " -> " + s1.equals(s3));

System.out.println(s1 + " equals " + s4 + " -> " + s1.equals(s4));

System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> "
+
```

```
                    s1.equalsIgnoreCase(s4));
} }
```

Результат запуска этого примера :

C:\> java equalsDemo

```
Hello equals Hello -> true
Hello equals Good-bye -> false
Hello equals HELLO -> false
Hello equalsIgnoreCase HELLO -> true
```

В классе String реализована группа сервисных методов, являющихся специализированными версиями метода equals. Метод regionMatches используется для сравнения подстроки в исходной строке с подстрокой в строке-параметре. Метод startsWith проверяет, начинается ли данная подстрока фрагментом, переданным методу в качестве параметра. Метод endsWith проверяет совпадает ли с параметром конец строки.

Равенство

Метод equals и оператор == выполняют две совершенно различных проверки. Если метод equal сравнивает символы внутри строк, то оператор == сравнивает две переменные-ссылки на объекты и проверяет, указывают ли они на разные объекты или на один и тот же. В очередном нашем примере это хорошо видно

— содержимое двух строк одинаково, но, тем не менее, это — различные объекты, так что equals и == дают разные результаты.

```
class EqualsNotEqualTo {
public static void main(String args[]) {
String s1 = "Hello";
String s2 = new String(s1);
System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));
System.out.println(s1 + " == " + s2 + ", -> " + (s1 == s2));
} }
```

Вот результат запуска этого примера:

C:\> java EqualsNotEqualTo

Hello equals Hello -> true

Hello == Hello -> false

Упорядочение

Зачастую бывает недостаточно просто знать, являются ли две строки идентичными. Для приложений, в которых требуется сортировка, нужно знать, какая из двух строк меньше другой. Для ответа на этот вопрос нужно воспользоваться методом compareTo класса String. Если целое значение, возвращенное методом, отрицательно, то строка, с которой был вызван метод, меньше строки-параметра, если положительно — больше. Если же метод compareTo вернул значение 0, строки идентичны. Ниже приведена программа, в которой выполняется пузырьковая сортировка массива строк, а для сравнения строк используется метод compareTo. Эта программа выдает отсортированный в алфавитном порядке список строк.

```
class SortString {
static String arr[] = {"Now", "is", "the", "time", "for", "all",
                       "good", "men", "to", "come", "to", "the",
                       "aid", "of", "their", "country" };
public static void main(String args[]) {
for (int j = 0; i < arr.length; j++) {
     for (int i = j + 1; i < arr.length; i++) {
          if (arr[i].compareTo(arr[j]) < 0) {
              String t = arr[j];
              arr[j] = arr[i];
              arr[i] = t;
```

```
            }
      }
      System.out.println(arr[j]);
}
} }
```

indexOf и lastIndexOf

В класс String включена поддержка поиска определенного символа или подстроки, для этого в нем имеются два метода — indexOf и lastIndexOf. Каждый из этих методов возвращает индекс того символа, который вы хотели найти, либо индекс начала искомой подстроки. В любом случае, если поиск оказался неудачным методы возвращают значение -1. В очередном примере показано, как пользоваться различными вариантами этих методов поиска.

```
class indexOfDemo {
public static void main(String args[]) {
String s = "Now is the time for all good men " +
           "to come to the aid of their country " +
           "and pay their due taxes.";
```

```
System.out.println(s);
System.out.println("indexOf(t) = " + s.indexOf('f'));
System.out.println("lastIndexOf(t) = " + s.lastIndexOf('f'));
System.out.println("indexOf(the) = " + s.indexOf("the"));
System.out.println("lastIndexOf(the) = " + s.lastIndexOf("the"));
System.out.println("indexOf(t, 10) = " + s.indexOf('f' , 10));
System.out.println("lastIndexOf(t, 50) = " + s.lastIndexOf('f' , 50));
System.out.println("indexOf(the, 10) = " + s.indexOf("the", 10));
System.out.println("lastIndexOf(the, 50) = " + s.lastIndexOf("the", 50));
} }
```

Ниже приведен результат работы этой программы. Обратите внимание на то, что индексы в строках начинаются с нуля.

C:> java indexOfDemo

```
Now is the time for all good men to come to the aid of their country
and pay their due taxes.
indexOf(t) = 7
```

```
lastIndexOf(t) = 87

indexOf(the) = 7

lastIndexOf(the) = 77

index0f(t, 10) = 11

lastIndex0f(t, 50) = 44

index0f(the, 10) = 44

lastIndex0f(the, 50) = 44
```

Модификация строк при копировании

Поскольку объекты класса String нельзя изменять, всякий раз, когда вам захочется модифицировать строку, придется либо копировать ее в объект типа StringBuffer, либо использовать один из описываемых ниже методов класса String, которые создают новую копию строки, внося в нее ваши изменения.

substring

Вы можете извлечь подстроку из объекта String, используя метод **substring**. Этот метод создает новую копию символов из того диапазона индексов оригинальной строки, который вы указали при вызове. Можно указать только индекс первого символа нужной подстроки — тогда будут скопированы все символы, начиная с указанного и до конца строки. Также можно указать и начальный, и конечный индексы — при этом в новую строку будут скопированы все символы, начиная с первого указанного, и до (но не включая его) символа, заданного конечным индексом.

"Hello World".substring(6) -> "World"

"Hello World".substring(3,8) -> "lo Wo"

concat

Слияние, или конкатенация строк выполняется с помощью метода concat. Этот метод создает новый объект String, копируя в него содержимое исходной строки и добавляя в ее конец строку, указанную в параметре метода.

"Hello".concat(" World") -> "Hello World"

replace

Методу replace в качестве параметров задаются два символа. Все символы, совпадающие с первым, заменяются в новой копии строки на второй символ.

"Hello".replace('l' , 'w') -> "Hewwo"

toLowerCase и toUpperCase

Эта пара методов преобразует все символы исходной строки в нижний и верхний регистр, соответственно.

"Hello".toLowerCase() -> "hello"

"Hello".toUpperCase() -> "HELLO"

trim

И, наконец, метод trim убирает из исходной строки все ведущие и замыкающие пробелы.

“Hello World “.trirn() -> "Hello World"

valueOf

Если вы имеете дело с каким-либо типом данных и хотите вывести значение этого типа в удобочитаемом виде, сначала придется преобразовать это значение в текстовую строку. Для этого существует метод valueOf. Такой статический метод определен для любого существующего в Java типа данных (все эти методы совмещены, то есть используют одно и то же имя). Благодаря этому не составляет труда преобразовать в строку значение любого типа.

StringBuffer

StringBuffer — близнец класса String, предоставляющий многое из того, что обычно требуется при работе со строками. Объекты класса String представляют собой строки фиксированной длины, которые нельзя изменять. Объекты типа StringBuffer представляют собой последовательности символов, которые могут расширяться и модифицироваться. Java активно использует оба класса, но многие программисты предпочитают работать только с объектами типа String, используя оператор +. При этом Java выполняет всю необходимую работу со StringBuffer за сценой.

Конструкторы

Объект StringBuffer можно создать без параметров, при этом в нем будет зарезервировано место для размещения 16 символов без возможности изменения длины строки. Вы также можете передать конструктору целое число, для того чтобы явно задать требуемый размер буфера. И, наконец, вы можете передать конструктору строку, при этом она будет скопирована в объект и дополнительно к этому в нем будет зарезервировано место еще для 16 символов. Текущую длину StringBuffer можно определить, вызвав метод **length**, а для определения всего места, зарезервированного под строку в объекте StringBuffer нужно воспользоваться методом **capacity**. Ниже приведен пример, поясняющий это:

```
class StringBufferDemo {

public static void main(String args[]) {

StringBuffer sb = new StringBuffer("Hello");

System.out.println("buffer = " + sb);

System.out.println("length = " + sb.length());

System.out. println("capacity = " + sb.capacity());

} }
```

Вот вывод этой программы, из которого видно, что в объекте String-Buffer для манипуляций со строкой зарезервировано дополнительное место.

C:\> java StringBufferDemo

```
buffer = Hello

length = 5

capacity = 21
```

ensureCapacity

Если вы после создания объекта StringBuffer захотите зарезервировать в нем место для определенного количества символов, вы можете *для* установки размера буфера воспользоваться методом **ensureCapacity**. Это бывает полезно, когда вы заранее знаете, что вам придется добавлять к буферу много небольших строк.

setLength

Если вам вдруг понадобится в явном виде установить длину строки в буфере, воспользуйтесь методом setLength. Если вы зададите значение, большее чем длина содержащейся в объекте строки, этот метод заполнит конец новой, расширенной строки символами с кодом нуль. В приводимой чуть дальше программе setCharDemo метод sstLength используется для укорачивания буфера.

charAt и setCharAt

Одиночный символ может быть извлечен из объекта StringBuffer с помощью метода **charAt**. Другой метод

setCharAt позволяет записать в заданную позицию строки нужный символ. Использование обоих этих методов проиллюстрировано в примере:

```
class setCharAtDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("Hello");
System.out.println("buffer before = " + sb);
System.out.println("charAt(1) before = " + sb.charAt(1));
sb.setCharAt(1, 'i');
sb.setLength(2);
System.out.println("buffer after = " + sb);
System.out.println("charAt(1) after = " + sb.charAt(1));
} }
```

Вот вывод, полученный при запуске этой программы.

C:\> java setCharAtDemo

```
buffer before = Hello
charAt(1) before = e
```

```
buffer after = Hi

charAt(1) after = i
```

append

Метод **append** класса StringBuffer обычно вызывается неявно при использовании оператора + в выражениях со строками. Для каждого параметра вызывается метод String.valueOf и его результат добавляется к текущему объекту StringBuffer. К тому же при каждом вызове метод append возвращает ссылку на объект StringBuffer, с которым он был вызван. Это позволяет выстраивать в цепочку последовательные вызовы метода, как это показано в очередном примере.

```
class appendDemo {

public static void main(String args[]) {

String s;

int a = 42;

StringBuffer sb = new StringBuffer(40);

s = sb.append("a = ").append(a).append("!").toString();

System.out.println(s);

} }
```

Вот вывод этого примера:

C:\> Java appendDemo

```
a = 42!
```

insert

Метод **insert** идентичен методу append в том смысле, что для каждого возможного типа данных существует своя совмещенная версия этого метода. Правда, в отличие от append, он не добавляет символы, возвращаемые методом String.valueOf, в конец объекта StringBuffer, а вставляет их в определенное место в буфере, задаваемое первым его параметром. В очередном нашем примере строка "there" вставляется между "hello" и "world!".

```
class insertDemo {
public static void       main(String args[]) {
StringBuffer sb = new StringBuffer("hello world !");
sb.insert(6,"there ");
System.out.println(sb);
} }
```

При запуске эта программа выводит следующую строку:

C:\> java insertDemo

hello there world!

Без строк не обойдешься

Почти любой аспект программирования в Java на каком либо этапе подразумевает использование классов String и StringBuffer. Они понадобятся и при отладке, и при работе с текстом, и при указании имен файлов и адресов URL в качестве параметров методам. Каждый второй байт большинства строк в Java — нулевой (Unicode пока используется редко). То, что строки в Java требуют вдвое больше памяти, чем обычные ASCII, не очень пугает, пока вам для эффективной работы с текстом в редакторах и других подобных приложениях не придется напрямую работать с огромным массивом типа char.

№	функция	диапазон изменения аргумента	n	сумма
1	$y = 3^X$	$0{,}1 \le x \le 1$	10	$S = 1 + \frac{\ln 3}{1!}x + \frac{\ln^2 3}{2!}x^2 + \ldots + \frac{\ln^n 3}{n!}x^n$
2	$y = -\ln\left\|2\sin\frac{x}{2}\right\|$	$\frac{\pi}{5} \le x \le \frac{9\pi}{5}$	40	$S = \cos x + \frac{\cos 2x}{2} + \ldots + \frac{\cos nx}{n}$
3	$y = \sin X$	$0{,}1 \le x \le 1$	10	$S = x - \frac{x^3}{3!} + \ldots + (-1)^n \frac{x^{2n+1}}{(2n+1)!}$
4	$y = XarctgX - \ln\sqrt{1+x^2}$	$0{,}1 \le x \le 0{,}8$	10	$S = \frac{x^2}{2} - \frac{x^4}{12} + \ldots + (-1)^{n+1}\frac{x^{2n}}{2n(2n-1)}$
5	$y = e^x$	$1 \le x \le 2$	15	$S = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \ldots + \frac{x^n}{n!}$
6	$y = e^{x\cos\pi/4} \cdot \cos(x\sin\pi/4)$	$0{,}1 \le x \le 1$	25	$S = 1 + \frac{\cos\frac{\pi}{4}}{1!}x + \ldots + \frac{\cos n\frac{\pi}{4}}{n!}x^n$
7	$y = \cos x$	$0{,}1 \le x \le 1$	10	$S = 1 - \frac{x^2}{2!} + \ldots + (-1)^n \frac{x^{2n}}{(2n)!}$
8	$y = \frac{x\sin\pi/4}{1 - 2x\cos\pi/4 + x^2}$	$0{,}1 \le x \le 0{,}8$	40	$S = x\sin\frac{\pi}{4} + x^2\sin 2\frac{\pi}{4} + \ldots + x^n \sin n\frac{\pi}{4}$
9	$y = \frac{1}{4}\ln\frac{1+x}{1-x} + \frac{1}{2}arctgX$	$0{,}1 \le x \le 0{,}8$	3	$S = x + \frac{x^5}{5} + \ldots + \frac{x^{4n+1}}{4n+1}$
10	$y = e^{\cos x}\cos(\sin x)$	$0{,}1 \le x \le 1$	20	$S = 1 + \frac{\cos x}{1!} + \ldots + \frac{\cos nx}{n!}$
11	$y = (1+2x^2)e^{x^2}$	$0{,}1 \le x \le 1$	10	$S = 1 + 3x^2 + \ldots + \frac{2n+1}{n!}x^{2n}$
12	$y = -\frac{1}{2}\ln(1 - 2x\cos\frac{\pi}{3} + x^2)$	$0{,}1 \le x \le 0{,}8$	35	$S = \frac{x\cos\frac{\pi}{3}}{1} + \frac{x^2\cos 2\frac{\pi}{3}}{2} + \ldots + \frac{x^n\cos n\frac{\pi}{3}}{n}$
13	$y = \frac{1}{2}\ln x$	$0{,}2 \le x \le 1$	10	$S = \frac{x-1}{x+1} + \frac{1}{3}(\frac{x-1}{x+1})^3 + \ldots + \frac{1}{2n+1}(\frac{x-1}{x+1})^{2n+1}$
14	$y = \frac{1}{4}(x^2 - \frac{\pi^2}{3})$	$\frac{\pi}{5} \le x \le \pi$	20	$S = -\cos x + \frac{\cos 2x}{2^2} + \ldots + (-1)^n \frac{\cos nx}{n^2}$
15	$y = \frac{1+x^2}{2}arctgX - \frac{x}{2}$	$0{,}1 \le x \le 1$	30	$S = \frac{x^3}{3} - \frac{x^5}{15} + \ldots + (-1)^{n+1}\frac{x^{2n+1}}{4n^2-1}$
16	$y = \frac{\pi^2}{8} - \frac{\pi}{4}\|x\|$	$\frac{\pi}{5} \le x \le \pi$	40	$S = \cos x + \frac{\cos 3x}{3^2} + \ldots + \frac{\cos(2n-1)x}{(2n-1)^2}$

17	$y=\frac{e^x+e^{-x}}{2}$	$0{,}1 \le x \le 1$	10	$S=1+\frac{x^2}{2!}+\ldots+\frac{x^{2n}}{(2n)!}$
18	$y=\frac{1}{2}-\frac{\pi}{4}\lvert\sin x\rvert$	$0{,}1 \le x \le 0{,}8$	50	$S=\frac{\cos 2x}{3}+\frac{\cos 4x}{15}+\ldots+\frac{\cos 2nx}{4n^2-1}$
19	$y=e^{2x}$	$0{,}1 \le x \le 1$	20	$S=1+\frac{2x}{1!}+\ldots+\frac{(2x)^n}{n!}$
20	$y=(\frac{x^2}{4}+\frac{x}{2}+1)e^{x/2}$	$0{,}1 \le x \le 1$	30	$S=1+2\frac{x}{2}+\ldots+\frac{n^2+1}{n!}(\frac{x}{2})^n$
21	$y=arctgX$	$0{,}1 \le x \le 1$	40	$S=x-\frac{x^3}{3}+\ldots(-1)^n\frac{x^{2n+1}}{2n+1}$
22	$y=(1-\frac{x^2}{2})\cos x-\frac{x}{2}\sin x$	$0{,}1 \le x \le 1$	35	$S=1-\frac{3}{2}x^2+\ldots..+(-1)^n\frac{2n^2+1}{(2n)!}x^{2n}$
23	$y=2(\cos^2 x-1)$	$0{,}1 \le x \le 1$	15	$S=-\frac{(2x)^2}{2}+\frac{(2x)^4}{24}+\ldots+(-1)^n\frac{(2x)^{2n}}{(2n)!}$
24	$y=\ln(\frac{1}{2+2x+x^2})$	$-2 \le x \le -0{,}1$	40	$S=-(1+x)^2+\frac{(1+x)^4}{2}+\ldots+(-1)^n\frac{(1+x)^{2n}}{n}$
25	$y=\frac{e^x-e^{-x}}{2}$	$0{,}1 \le x \le 1$	20	$S=x+\frac{x^3}{3!}+\ldots+\frac{x^{2n+1}}{(2n+1)!}$

Разложение функции в ряд

Действительная функция f(x) называется аналитической в точке ε, если в некоторой окрестности $|x-\varepsilon|<R$ этой точки функция разлагается в степенной ряд (ряд Тейлора):

$$f(x)=f(\varepsilon)+f'(\varepsilon)(x-\varepsilon)+\frac{f''(\varepsilon)}{2!}(x-\varepsilon)^2+\ldots+\frac{f^{(n)}(\varepsilon)}{n!}(x-\varepsilon)^n+\ldots \qquad (1)$$

При ε=0 получаем ряд Маклорена:

$$f(x)=f(0)+f'(0)(x)+\frac{f''(0)}{2!}(x)^2+\ldots+\frac{f^{(n)}(0)}{n!}(x)^n+\ldots \qquad (2)$$

Разность $R_n(x)=f(x)-\sum_{k=0}^{n}\frac{f^{(k)}(\varepsilon)}{k!}(x-\varepsilon)^k$ (3)

называется остаточным членом и представляет собой ошибку при замене функции f(x) полиномом Тейлора.

Для ряда Маклорена

$$R_n(x)=\frac{f^{(n+1)}(\theta\cdot x)}{(n+1)!}x^{n+1} \quad \text{где } 0<\theta<1. \qquad (4)$$

Таким образом, вычисление значения функции можно свести к вычислению суммы числового ряда

$a_1+a_2+\ .\ .\ .\ +a_n+\ .\ .\ .\ \ .$ (5)

Известно, что числовой ряд называется сходящимся, если существует предел последовательности его частных сумм:

$$S=\lim_{n\to\infty}S_n, \qquad (6)$$

где $S_n=a_1+a_2+\ .\ .\ .\ +a_n+\ .\ .\ .\ \ .$

Число S называется суммой ряда.

Очевидно, что $S=S_n+R_n$,

где R_n - остаток ряда, причем $R\to 0$ при $n\to\infty$.

Для нахождения суммы S сходящегося ряда (5) с заданной точностью ε нужно выбрать число слагаемых n столь большим, чтобы имело место неравенство

$|R_n|<\varepsilon$.

Тогда частная сумма S_n приближенно может быть принята за точную сумму S ряда (5).

Приближенно n выбрать так, чтобы имело место неравенство $|S_{n+1}-S_n|<\varepsilon$ или $a_n<\varepsilon$.

Задача сводится к замене функции степенным рядом и нахождению суммы некоторого количества слагаемых $S=\sum a_n(x,n)$ при различных параметрах суммирования x . Каждое слагаемое суммы зависит от параметра x и номера n, определяющего место этого слагаемого в сумме.

Обычно формула общего члена суммы принадлежит одному из следующих трех типов:

а) $\frac{x^n}{n!}$; $(-1)^n\frac{x^{2n+1}}{(2n+1)!}$; $\frac{x^{2n}}{(2n)!}$;

б) $\frac{\cos(nx)}{n}$; $\frac{\sin(2n-1)x}{2n-1}$; $\frac{\cos(2nx)}{4n^2-1}$;

в) $\frac{x^{4n+1}}{4n+1}$; $(-1)^n\frac{\cos(nx)}{n^2}$; $\frac{n^2+1}{n!}(\frac{x}{2})^n$.

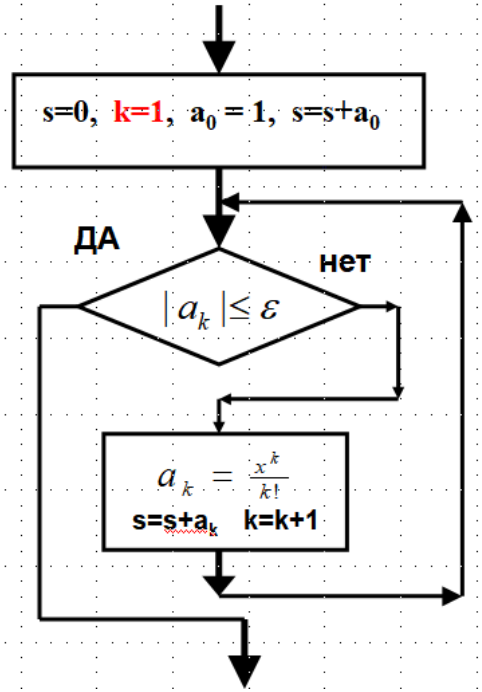
В случае а) для вычисления члена суммы a_n целесообразно использовать рекуррентные соотношения, т. е. выражать последующий член суммы через предыдущий: $a_{n+1}=\psi(x, n)a_n$. Это позволит существенно сократить объем вычислительной работы. Кроме того, вычисление члена суммы по общей формуле в ряде случаев невозможно (например, из-за наличия n!).

В случае б) применение рекуррентных соотношений нецелесообразно. Вычисления будут наиболее эффективными, если каждый член суммы вычислять по общей формуле $a_n=\phi(x, n)$.

В случае в) член суммы целесообразно представить в виде двух сомножителей, один из которых вычисляется по рекуррентному соотношению, а другой непосредственно $a_n=\phi(x, n)*c_n(x,n)$, где $c_n=c_{n-1}\psi(x,n)$.

Пример – как «считать» элементарные функции?

Разложение	Область сходимости
$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!} + \ldots$	$x \in R$
$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots + (-1)^{n+1} \frac{x^{2n-1}}{(2n-1)!} + \ldots$	$x \in R$
$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \ldots + (-1)^n \frac{x^{2n}}{(2n)!} + \ldots$	$x \in R$
$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots + (-1)^{n-1} \frac{x^n}{n} + \ldots$	$x \in (-1,1]$
$(1+x)^m = 1 + mx + \frac{m(m-1)}{1 \cdot 2} x^2 + \frac{m(m-1)(m-2)}{1 \cdot 2 \cdot 3} x^3 + \ldots$	$x \in [-1,1]$, если $m \geq 0$; $x \in (-1,1]$, если $-1 < m < 0$; $x \in (-1,1)$, если $m \leq -1$
$\operatorname{arctg} x = x - \frac{x^3}{3} + \frac{x^5}{5} - \ldots + (-1)^{n-1} \frac{x^{2n-1}}{2n-1} + \ldots$	$x \in [-1,1]$
$\frac{1}{1+x} = 1 - x + x^2 - x^3 + \ldots + (-1)^n x^n + \ldots$	$x \in (-1,1)$



$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

$$e^x = \sum_{k=0}^{\infty} a_k$$

$$a_k = \frac{x^k}{k!}$$

Точность достигнута когда

$$|a_k| \le \varepsilon$$

Рекуррентная формула

Арифметическая прогрессия

$$a_{n+1} = a_n + d \qquad n \in N$$

Геометрическая прогрессия

$$b_{n+1} = b_n \bullet q \qquad n \in N$$

n!=1*2*3*…….(n-2)*(n-1)*n = (n-1)! * n

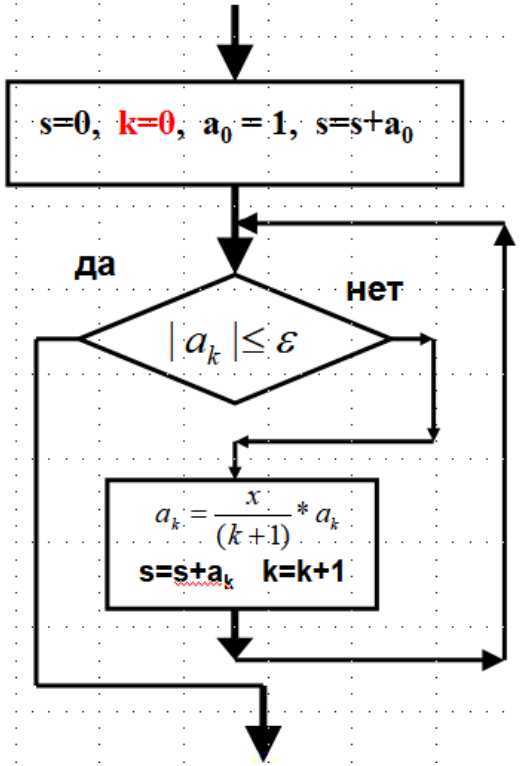
(n+1)! = (n+1)* n!

$X^{n+1} = X^n * X$

$$\frac{a_{k+1}}{a_k} = \frac{x^{k+1} * k!}{(k+1)! * x^k} = \frac{x^k * x * k!}{(k+1) * k! * x^k}$$

$$\frac{a_{k+1}}{a_k} = \frac{x}{(k+1)}$$

$$a_{k+1} = \frac{x}{(k+1)} * a_k \qquad a_{следующий} = \frac{x}{(k+1)} * a_{предыдущий}$$



В итерационных алгоритмах необходимо обеспечить обязательное достижение условия выхода из цикла (**сходимость итерационного процесса**). В противном случае произойдет **"зацикливание"** алгоритма, т.е. не будет выполняться основное свойство алгоритма –– **конечность**.

```
/*
Пример 1-й лабораторной работы.
     5-й вариант y(x)=exp(x)
*/
import java.math.*;
class lr1 {
 // TODO code application logic here
    public static void main(String [] args){
         float a, b, x, er1, er2, y1, y2, y3, eps;
         int k_dot; // количество строк в таблице
         int nsum;
         // исходные данные
         // TODO организовать ввод данных в диалоге
         a=1.0e0f;  b=2.0e0f;  eps=1e-5f;
         k_dot=10; nsum=15;
         // Tab - класс печати таблицы
        // Digit - класс расчетов Y(x) разными способами
         Tab pr=new Tab("X", "Y1", "Y2", "Y3", "er1", "er2");
         // pr.print(.......)
        float h; // шаг по X
        h=(b-a)/(float)k_dot;
         // класс "расчетчик" методы ra1(x) ra2(x) ra3(x)
        Digit dig=new Digit(nsum, eps);
         for(x=a; x <=b;  x=x+h) {
            y1=dig.ra1(x);
            y2=dig.ra2(x);
            y3=dig.ra3(x);
            er1=Math.abs( (y1-y2)/y1 )*100;
           er2=Math.abs( (y1-y3)/y1 )*100;
            pr.print(x, y1, y2, y3, er1, er2);
         };
```

```
        pr.close();
    };
};

class Tab {
    private String n1, n2, n3, n4, n5, n6;
    private
       String br0="+------------";
       String br;
    public Tab(String z1, String z2,
               String z3, String z4,
               String z5, String z6) {
        n1=z1; n2=z2; n3=z3; n4=z4; n5=z5; n6=z6;
        br="";
        for(int i=1; i <=6; i++)
          br= br+br0;
        br=br+"+";
        System.out.println(br);
        System.out.print("|      " + n1 + "     ");
        System.out.print("|    " + n2 +  "      ");
        System.out.print("|    " + n3 + "      ");
        System.out.print("|    " + n4 + "      ");
        System.out.print("|    " + n5 + "     ");
        System.out.println("|   " + n6 + "      |");
        System.out.println(br);
    };
    public void print(float p1, float p2,
                  float p3, float p4,
                   float p5, float p6 ) {
          System.out.printf("| %10.4e ", p1);
           System.out.printf("| %10.4e ", p2);
```

```
            System.out.printf("| %10.4e ", p3);
            System.out.printf("| %10.4e ", p4);
            System.out.printf("| %10.4e ", p5);
            System.out.printf("| %10.4e |\n", p6);
     };
     public void close() {
       System.out.println(br);

     }
};
class Digit {
   private float nsum, eps;
   public Digit(float nsum, float eps) {
       this.nsum=nsum;
       this.eps = eps;
   };
   public float ra1(float x) {
      return (float) Math.exp( (double)x );
   };
   public float ra2(float x) {
       float sum; int k; float a;
       sum=0.0f;
       a=1.0f;
       sum = sum + a;
       for(k=0; k <= nsum; k++ ) {
         a = a * x / (k+1);
         sum+=a;
       }
      return sum;
   };
   public float ra3(float x) {
```

```
        float sum; int k; float a;
        sum=0.0f;
        a=1.0f;
        sum = sum + a;
        k=0;
        while ( Math.abs(a) > eps ) {
          a = a * x / (k+1);
          sum+=a;
          k++;
        }
       return sum;
   };
};
/*
```

```
D:\java>java lr1
```

X	Y1	Y2	Y3	er1	er2
1,0000e+00	2,7183e+00	2,7183e+00	2,7183e+00	8,7709e-06	0,0000e+00
1,1000e+00	3,0042e+00	3,0042e+00	3,0042e+00	1,5873e-05	3,9681e-05
1,2000e+00	3,3201e+00	3,3201e+00	3,3201e+00	7,1810e-06	0,0000e+00
1,3000e+00	3,6693e+00	3,6693e+00	3,6693e+00	6,4977e-06	6,4977e-06
1,4000e+00	4,0552e+00	4,0552e+00	4,0552e+00	1,1759e-05	3,5276e-05
1,5000e+00	4,4817e+00	4,4817e+00	4,4817e+00	0,0000e+00	1,0640e-05
1,6000e+00	4,9530e+00	4,9530e+00	4,9530e+00	9,6272e-06	0,0000e+00
1,7000e+00	5,4739e+00	5,4739e+00	5,4739e+00	8,7110e-06	3,4844e-05
1,8000e+00	6,0496e+00	6,0496e+00	6,0496e+00	0,0000e+00	7,8821e-06
1,9000e+00	6,6859e+00	6,6859e+00	6,6859e+00	7,1320e-06	1,4264e-05

1. Проверить является ли строка палиндромом. (Палиндром - это выражение, которое читается одинакова слева направо и справа налево).
2. Напечатать самое длинное и самое короткое слово в этой строке.
3. Напечатать все слова, которые не содержат гласных букв.
4. Напечатать все слова, которые содержат по одной цифре.
5. Напечатать все слова, которые совпадают с ее первым словом.
6. Преобразовать строку таким образом, чтобы сначала в ней были напечатаны только буквы, а потом только цифры, не меняя порядка следования символов в строке.
7. Преобразовать строку так, чтобы все буквы в ней были отсортированы по возрастанию.
8. Преобразовать строку так, чтобы все цифры в ней были отсортированы по убыванию.
9. Преобразовать строку так, чтобы все слова в ней стали идентификаторами, слова состоящие только из цифр - удалить.
10. Напечатать все слова-палиндромы, которые есть в этой строке(см 1 вариант).
11. Преобразовать строку таким образом, чтобы в ее начале были записаны слова, содержащие только цифры, потом слова, содержащие только буквы, а затем слова, которые содержат и буквы и цифры.
12. Преобразовать строку таким образом, чтобы все слова в ней были напечатаны наоборот.
13. Преобразовать строку таким образом, чтобы буквы каждого слова в ней были отсортированы по возрастанию.
14. Преобразовать строку таким образом, чтобы цифры каждого слова в ней были отсортированы по убыванию.
15. Преобразовать строку таким образом, чтобы в ней остались только слова, содержащие буквы и цифры, остальные слова удалить.
16. Определить какое слово встречается в строке чаще всего.
17. Определить какие слова встречаются в строке по одному разу.
18. Все слова строки, которые начинаются с буквы, отсортировать в алфавитном порядке.
19. Все слова строки, которые начинаются с цифры отсортировать по убыванию.
20. Удалить из строки все слова, которые не являются идентификаторами.
21. Проверить является ли введенная строка правильным текстовым представлением вещественного числа в научной нотации.
22. Подсчитать частоты встречаемости различных слов в строке.
23. Произвести шифрование и дешифрование строки путем циклического вращения кодов символов.
24. Проверить, является ли введенная строка идентификатором языка программирования.
25. Произвести пословный перевод всех слов строки. Подстановочный словарь может содержать не более 10 слов, можно не учитывать изменение форм слова.

```
import java.io.*;
import java.util.*;
import java.util.Scanner;

/* string */
class words {
   private static BufferedReader in =
               new BufferedReader(new InputStreamReader(System.in));
   public static void main(String ar[]) {
      int i, k;
      String s=   new String();
      String word=new String();
      while(true) {
         System.out.println("input string (empty - end):");
         try {
            s=in.readLine(); //Читаем с клавиатуры
         } catch (Exception e) {
            System.out.println( "*** Error of Inupt console ....." );
            s="";
        };
        System.out.println("input string is:");
        System.out.println("<"+s+">");
        if ( s.length() == 0 ) break;
        k=1;
        /* разбор 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
           word=st.nextToken();
           System.out.println(k+" \t <" + word + ">");
           k++;
        };
      };
      System.out.println("Test words ending");
   };
};


/**********пример чтения с использованием Scanner
Scanner in =new Scanner( System.in );
int i = 0;
do {
    try {
       System.out.println( "Введите число: " );
       //   in.nextInt();   in.nextLine();
        Double op1 = in.nextDouble();
        System.out.println( op1 * op1 );
     } catch ( Exception e ) {
         System.out.println( "Число введено неверно!" );
         in.nextLine();
    }
} while ( i++ <= 10 );
********************/
```

```
import java.io.*;
import java.util.*;
import java.util.Scanner;

/* string */
class words {
  private static BufferedReader in =
          new BufferedReader(new InputStreamReader(System.in));
  public static void main(String ar[]) {
    int i, k;
    String s=   new String();
    String word=new String();
    while(true) {
      System.out.println("input string (empty - end):");
      try {
        s=in.readLine(); //Читаем с клавиатуры
      } catch (Exception e) {
        System.out.println( "*** Error of Inupt console ....." );
        s="";
      };
      System.out.println("input string is:");
      System.out.println("<"+s+">");
      if ( s.length() == 0 ) break;
      k=1;
            /* разбор 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
        word=st.nextToken();
        System.out.println(k+" \t <" + word + ">");
        k++;
      };
```

```
    };
    System.out.println("Test words ending");
  };
};


/**********пример чтения с использованием Scanner
Scanner in =new Scanner( System.in );
int i = 0;
do {
   try {
     System.out.println( "Введите число: " );
     //   in.nextInt();   in.nextLine();
      Double op1 = in.nextDouble();
      System.out.println( op1 * op1 );
    } catch ( Exception e ) {
       System.out.println( "Число введено неверно!" );
       in.nextLine();
   }
} while ( i++ <= 10 );
********************/
```

ГЛАВА 5
Работа со

Очень большое место в обработке занимает работа с текстами. Как и многое другое, текстовые строки в языке Java являются объектами. Они 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, 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 ее к типу stringBuffer , меняет длину, потом 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 строки "Это", " одна ", "строка", и 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 .

Класс String

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

Как создать строку

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 ни одного символа, и пустой ссылкой string s = null, не ни на какую строку и не объектом.

Самый способ создать объект с точки зрения ООП — это вызвать его конструктор в операции new. Класс string вам девять :

- string() — создается объект с пустой строкой;
- string (String str) — из одного объекта создается другой, поэтому этот конструктор редко;
- string (StringBuf fer str) — 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, с индекса offset и 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 заданий индексов offset , count или кодировки encoding возникает исключительная ситуация.

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

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

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, местной кодовой странице. В варианте MS Windows это обычно кодовая страница CP1251.

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

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

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

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

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

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

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

- 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 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"), "Cp1251");
          winLikeWin = new String(byteCp1251, "Cp1251");  //Правильно
          winLikeDOS = new String(byteCp1251,: "Cp866");
          winLikeUNIX - new String(byteCp1251, "KOI8-R");
          dosLikeWin = new String(byteCp866, "Cp1251");  // Для консоли
          dosLikeDOS = new String(byteCp866, "Cp866");   // Правильно
          dosLikeUNIX = new String(byteCp866, "KOI8-R");
          unixLikeWin = new String(byteKOISR, "Cp1251");
          unixLikeDOS = new String(byteKOISR, "Cp866");
          unixLikeUNIX = new String(byteKOISR, "KOI8-R");  // Правильно
          System.out.print(msg + "Cp1251: ");
          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.

В первые три строки консоли выводятся массивы байтов byteCP1251 , byteCP866 и byteKOI8R без 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.

строки консоли содержат 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 вносит вывод потока символов на консоль или в файл.

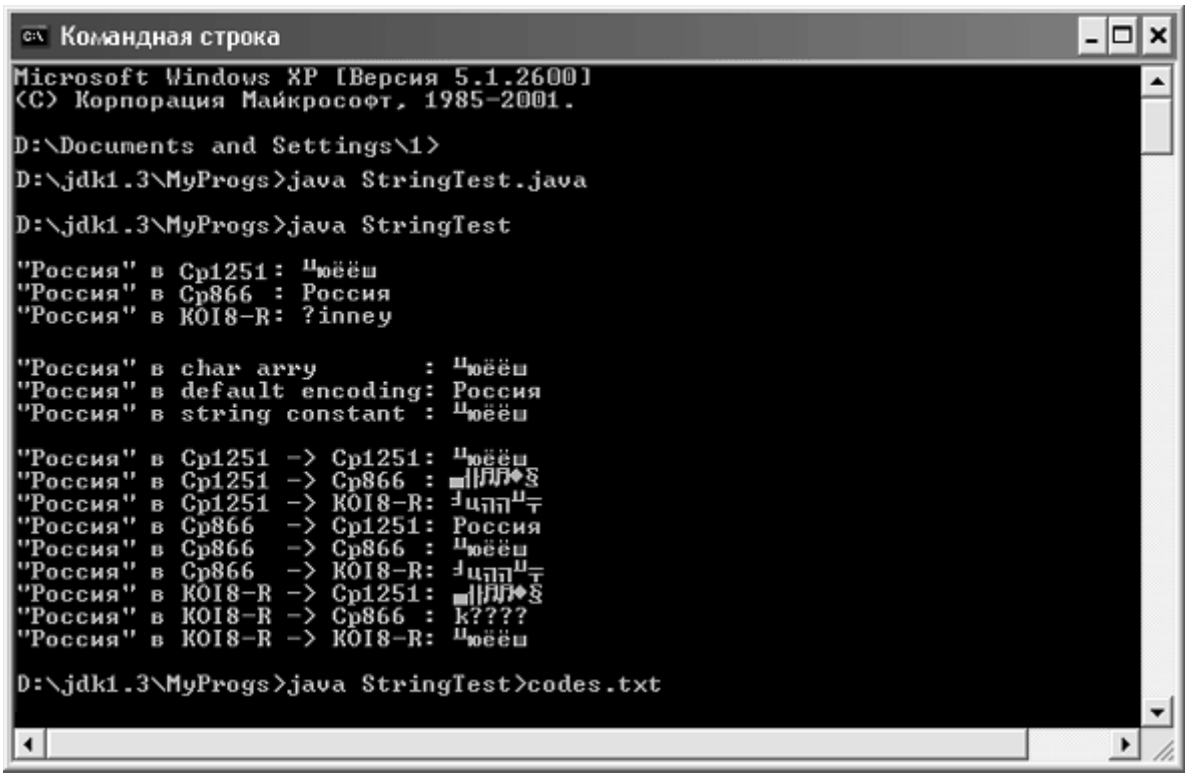


Рис. 5.1. Вывод строки на консоль MS Windows 2000

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

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

В строке рис. 5.1 сделано 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.

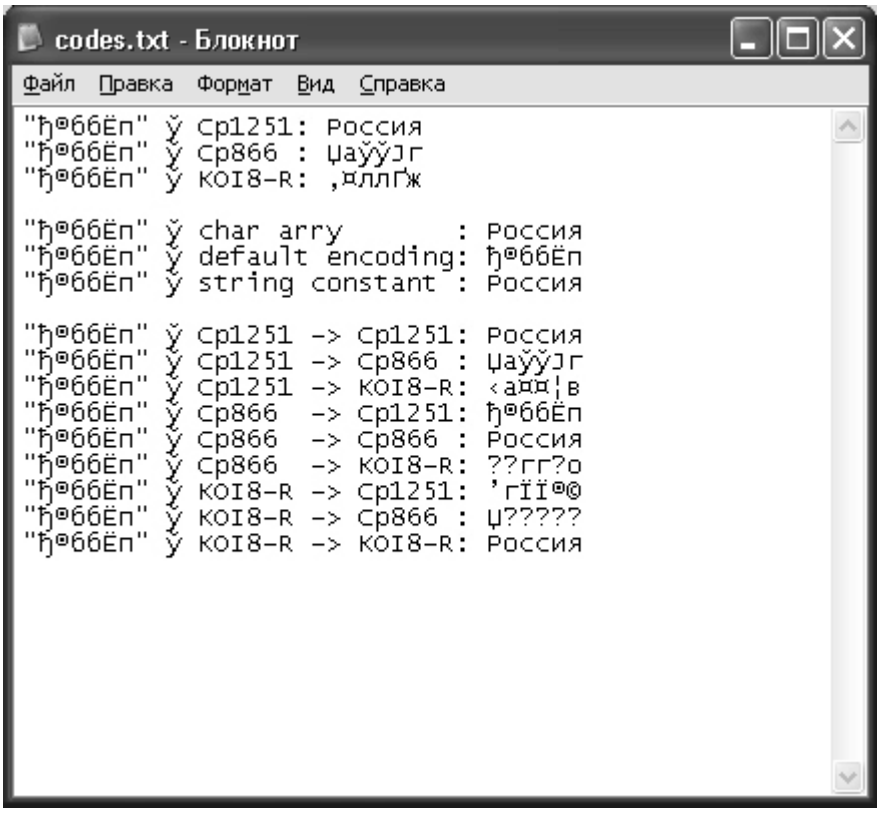


Рис. 5.2. Вывод строки в файл

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

Вопросы мы еще будем в *главах 9 и 18,* а пока заметьте, что при создании строки из массива байтов лучше указывать ту же самую 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 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 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 ".

строк

Со строками можно производить операцию *сцепления строк* (concatenation), знаком плюс +. Эта операция создает новую строку, просто 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 — += — 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" .

строками

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

Как узнать длину строки

Для того чтобы узнать длину строки, т. е. количество символов в ней, надо 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 — объект класса string . Заметьте, что строка — это не массив, у нее нет поля length .

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

Как 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

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

переменная ch будет иметь значение 't'

Все символы строки в виде массива символов можно получить методом

toCharArray() , 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 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 in d + (end - begin) - 1 .

Этот метод создает ситуацию в следующих :

- 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, все символы строки в байтовой кодировке ASCII, то 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, метод getBytes(String encoding) .

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

Как

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

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

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. 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 значение this.charAt(k) - str.charAt(k), т. е. разность кодировок Unicode первйх символов.
3. Во втором случае значение this.length() - str.length() , т. е. разность длин строк.
4. Если строки совпадают, 0.

Если значение str равно null , возникает исключительная ситуация.

Нуль 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 ситуацию, если obj не является строкой. В остальном он работает как метод compareTo(String str).

Эти методы не учитывают символов в локальной кодировке.

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

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

Сравнить подстроку данной строки this с 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 :

- хотя бы один из индексов ind1 или ind2 отрицателен;
- хотя бы одно из ind1 + len или ind2 + len больше длины 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 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) , индекс этого символа в строке или -1 , если символа ch в строке this нет.

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

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

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

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

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

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

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

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

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

Как найти

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 , 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. -1.

Последнее вхождение подстроки sub в данную строку this можно отыскать методом lastindexof (string sub), 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 l astIndexof(String stf, int ind). Если ind больше длины строки, то .поиск идет от конца строки, если ind < o , то -1.

Для того чтобы проверить, не начинается ли данная строка this с подстроки sub , логический метод startsWith(string sub) , 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 s tartsWith(String sub),int ind). Если индекс ind отрицателен или больше длины строки, false .

Для того чтобы проверить, не заканчивается ли данная строка this sub , 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.

выше методы создают ситуацию, если

```
sub == null.
```

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

Как изменить регистр букв

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

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

При этом локальная кодовая таблица по умолчанию. Если нужна другая локаль, то методы toLowerCase(Locale l oc) 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 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 о возвращает новую строку, в которой удалены начальные и конечные символы с кодами, не '\u0020 '.

Как данные другого типа в строку

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

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

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

Кроме того, в каждом классе есть метод tostring () , или просто от класса Object . Он объекты класса в строку. Фактически, метод valueOf о вызывает метод tostring() класса. Поэтому результат 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 длины. Только что созданный объект имеет буфер *емкости* (capacity), по умолчанию для хранения 16 символов. Емкость можно задать в объекта.

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

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

Этот метод изменит емкость, только если minCapacity будет больше длины в объекте строки. Емкость будет увеличена по 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 символы будут равны ' \uOOOO' . Если она будет меньше текущей длины, то строка будет обрезана, последние символы потеряются, точнее, будут заменены символом '\uOOOO' . Емкость при этом не изменится.

Если число newLength окажется , возникнет исключительная ситуация.

Совет

Будьте осторожны, новую длину объекта.

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

Создать объект класса stringBuf fer можно только .

В классе stringBuffer три :

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, строку str .

Как добавить

В классе stringBuffer есть десять методов append (), 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) строку str в конец данной строки. Если ссылка str == null, то строка "null".

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

Два метода к строке массив str и подмассив sub символов,

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 в строку своим методом tostring ().

Как вставить

Десять методов insert () 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 и меньше длины строки, иначе возникнет исключительная ситуация. Строка раздвигается, емкость буфера при . Методы возвращают ссылку ни ту же 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, в строку.

Два метода вставляют массив str и подмассив sub символов, 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 в строку своим методом toString ().

Как удалить

Метод delete tint 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 deieteCharAt (int ind) удаляет символ с указанным индексом ind . Длина строки на единицу.

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

Как заменить

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 , больше длины строки или больше end , возникает исключительная ситуация.

, метод replace () — это выполнение методов delete () и insert ().

Как строку

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

Например, после выполнения

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

получим s == "акортс яашьлобен отЭ".

разбор строки

Задача разбора введенного текста — *парсинг* (parsing) — вечная задача , наряду с и поиском. Написана масса программ-парсеров (parser), текст по различным признакам. Есть даже программы, парсеры по заданным правилам разбора: YACC, LEX и др.

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

В пакет java.utii входит простой класс stringiokenizer , разбор строк.

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, , символами табуляций '\t', перевода строки '\n' и возврата каретки '\r' . 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 в строке deiimeters не имеет значения. 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 - (Ь + с) / Ь * с", " \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.

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

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

Схема очень проста (листинг 5.2).

Листинг 5.2. Разбиение строки на слова :

```
String s = "Строка, которую мы хотим           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()) ;
}
```

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

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

В последних версиях JDK исходные тексты хранятся в архиватором jar файле src.jar, лежащем в корневом каталоге JDK, например, в каталоге **D:\jdk** l.3.

Чтобы распаковать их, перейдите в каталог jdk l.3:

D: > cd jdkl.3

и вызовите архиватор jar следующим образом:

D:\jdkl.3 > jar -xf src.jar

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