БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ

Кафедра численных методов и программирования

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ JAVA

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО КУРСУ
“МЕТОДЫ ПРОГРАММИРОВАНИЯ”

Для студентов механико-математического факультета

МИНСК
БГУ
2002

А в т о р ы – с о с т а в и т е л и
И. Н. Блинов, В. С. Романчик

Р е ц е н з е н т ы:

кандидат физико-математических наук, доцент *И. М. Галкин*
кандидат физико-математических наук *В.И. Адамович*

Рекомендовано Ученым советом
механико-математического факультета БГУ
“5” марта 2002 г., протокол № 5

PDF-версия:
Д. П. Глиндзич, А. В. Грызлов

Содержание

1. Основные понятия языка Java. Приложения и апплеты ……...…4
2. Типы данных и операции. Операторы управления программой…..…9
3. Классы, как новые типы данных. Поля данных и методы……...…15
4. Полиморфизм. Конструкторы……....19
5. Интерфейсы. Пакеты……...25
6. Класс String. Потоки ввода/вывода……....30
7. Обработка исключительных ситуаций……..…36
8. Классы событий……...41
9. Графические интерфейсы пользователя……....48
10. Элементы управления……..…57
11. Потоки и многопоточность……..…64
12. Сетевые программы……..…73

1. ОСНОВНЫЕ ПОНЯТИЯ ЯЗЫКА JAVA. ПРИЛОЖЕНИЯ И АППЛЕТЫ

Язык Java – это объектно-ориентированный язык программирования, используемый для разработки программ, работающих в сети Internet. Разработан в корпорации Sun Microsystems Inc. в 1995 году. Система программирования Java позволяет использовать World Wide Web (WWW) для распространения небольших интерактивных прикладных программ, которые размещаются на серверах Internet, транспортируются клиенту по сети, автоматически устанавливаются и запускаются на месте как часть документа WWW. Такая программа – апплет имеет весьма ограниченный доступ к ресурсам компьютера клиента, поэтому он может предоставить произвольный мультимедийный интерфейс и выполнять сложные вычисления, не привнося при этом риска порчи данных. Другим видом программ являются приложения Java, представляющие переносимые коды, которые могут выполняться на любом компьютере, независимо от его архитектуры. Генерируемый при этом виртуальный код представляет набор инструкций для выполнения на интерпретаторе виртуального кода – виртуальной Java-машине (Java Virtual Machine). Широкое распространение получили сервлеты и JSP (Java Server Pages), предоставляющие клиентам доступ к базам данных и приложениям на сервере.

Язык Java использует синтаксис языка C++ и может рассматриваться как его развитие. Основные отличия от C++ связаны с необходимостью уменьшения размеров программ и увеличения требований к безопасности переносимых приложений, работающих в сети. Java не поддерживает указателей (наиболее опасное средство языка C++), так как возможность работы с произвольными адресами памяти через безтиповые указатели позволяет игнорировать защиту памяти.

В Java все объекты программы расположены в динамической памяти (heap) и доступны по объектным ссылкам. Это решение исключило непосредственный доступ к памяти, но усложнило работу с элементами массивов и сделало ее менее эффективной по сравнению с C++ программами. Необходимо отметить, что объектные ссылки языка Java содержат информацию о классе объектов, на которые они ссылаются, так что объектные ссылки – это не указатели, а дескрипторы объектов. Наличие дескрипторов позволяет JavaVM выполнять проверку совместности типов на фазе интерпретации кода, генерируя исключение в случае ошибки. В Java пересмотрена и концепция дина-

мического распределения памяти: исключены функции освобождения динамически выделенной памяти free()/delete**.** Вместо этого реализована система автоматического освобождения памяти, выделенной с помощью оператора **new** (сборщик мусора).

Стремление упростить Java-программы и сделать их более понятными привело к отказу от файлов-заголовков (h-файлов) и препроцессорной обработки. Файлы-заголовки C++, содержащие прототипы классов и распространяемые отдельно от двоичного кода этих классов, усложняют управление версиями, что дает возможность злонамеренным пользователям получать доступ к приватным данным. В Java-программах спецификация класса и его реализация всегда содержатся в одном и том же файле. Но отказ от препроцессора сделал невозможным параметризацию классов по типам (классам) их членов, что усложнило программирование простых вещей (например, на Java, в отличие от C++, нельзя иметь массив, элементами которого являются объекты произвольного класса).

Java не поддерживает структуры и объединения. Классы Java не поддерживают перегрузку операторов (в том числе: <<, >>), а также аргументов по умолчанию. Java не поддерживает множественное наследование, имеет конструкторы, но не имеет деструкторов (использует автоматическую сборку мусора), не поддерживает typedef, беззнаковые целые, не использует goto.

Наиболее существенные новые возможности, появившиеся в Java, это интерфейсы (аналог абстрактных классов C++) и многопоточность (возможность одновременного выполнения частей программы).

```
// пример #1 : простое приложение: First.java
class First {
   public static void main (String[] args) {
System.out.println("Моя Первая Программа На Java !!! ");    }
}
```

Здесь класс **First** содержит метод **main()** с аргументами-параметрами командной строки **String[] args**, который является открытым членом класса. Программа представляет собой приложение, запускаемое интерпретатором, поэтому ее выполнение обязательно должно начинаться с открытого (**public**) метода **main(),** который виден и доступен любому классу. Ключевое слово **static** объявляет методы и переменные класса, используемые для работы с классом в целом. Символы верхнего и нижнего регистров различаются, как и в C++. Вывод строки *"Моя Первая Программа На Java !!!"* осуществляется методом **println() (ln** – переход к новой строке после вывода)

класса **System,** который включается автоматически вместе с пакетом **java.lang.** Программа помещается в файл, имя которого совпадает с именем класса. Простейший способ компиляции – вызов строчного компилятора: *javac First.java*. При успешной компиляции создается файл *First.class*. Выполнить этот виртуальный код можно с помощью интерпретатора Java: *java First.*

Следующая программа отображает в окне консоли аргументы командной строки функции **main().** Аргументы представляют массив строк, значения которых присваиваются объектам массива **String[] args**. Количество аргументов является значением **args.length**.

```
// пример #2 : Вывод аргументов командной строки : OutArgs.java
class OutArgs {
  public static void main(String[] args) {
        for (int j = 0; j < args.length; j++)
        System.out.println("Аргумент #" + j + "-> " + args[j]);   }
}
```

Запустим это приложение с помощью следующей командной строки:
Java OutArgs 2001 argument2 "Java-string"
Вывод: **Аргумент #0-> 2001**
Аргумент #1-> argument2
Аргумент #2-> Java-string

В следующем примере рассматривается ввод строки с консоли.

```
// пример # 3: Ввод строки с консоли: Input.java
import java.io.*;
class Input {
public static void main(String[] args){
InputStreamReader is = new InputStreamReader(System.in);
BufferedReader bis = new BufferedReader(is);
try {
System.out.println("Введите Ваше имя и нажмите <Enter>:");
String name = bis.readLine();
System.out.println("Привет, "+name); }
catch (IOException e){System.out.println("ошибка ввода "+ e); }
  }
}
```

Одна из целей разработки Java – это создание апплетов (маленьких программ, запускаемых внутри Web-браузера). Поскольку апплеты должны быть безопасными, они ограничены в своих возможностях,

хотя остаются мощным инструментом поддержкиWeb-программирования на стороне клиента.

```
// пример #4 : простой апплет : HelloApplet.java
import java.applet.Applet;
import java.awt.Graphics;
public class HelloApplet extends Applet {
   public void paint(Graphics g) {
     g.drawString("Это Я!", 50, 25);
     g.drawString("Здравствуйте!", 100, 40);    }
}
```

Апплету не нужен метод **main()** – код его запуска помещается в метод **init()** или **paint()**. Для запуска апплета нужно поместить ссылку на него в HTML документ и просмотреть этот документ Web-браузером, поддерживающим Java. При этом можно обойтись очень простым фрагментом (тегом) HTML внутри документа:

```
<applet code = HelloApplet   width=100  height=100>
</applet>
```

Классы в Java содержат переменные-члены класса и функции-члены класса, а также могут содержать конструкторы. Основные отличия от классов C++: все функции определяются внутри классов и называются методами; невозможно создать функцию, не являющуюся методом класса или объявить метод вне класса; ключевое слово inline не поддерживается; спецификаторы доступа **public, private, protected** воздействуют только на то, перед чем они стоят, а не на участок от одного до другого спецификатора как в C++; элементы по умолчанию не устанавливаются в **private**, а доступны для классов из данного пакета. Объявление класса имеет вид:

[спецификатор] class имя_класса [extends суперкласс] [implements интерфейс]

Спецификатор доступа класса может быть **public** (класс доступен объектам данного пакета и вне пакета), **final** (класс не может иметь подклассов), **abstract** (класс содержит абстрактные методы, объекты такого класса могут создавать только подклассы). По умолчанию спецификатор устанавливается в **friendly** (класс доступен в данном пакете). Слово **friendly** при объявлении не используется. Файл, содержащий **public class**, должен иметь то же имя, что и класс.

```
//пример #5: вычисление расстояния между точками Distance.java
class Locate {
   private double x,y;
setXY(double a, double b){x = a; y = b;}//по умолчанию friendly
```

```
double getX(){return x;}
double getY(){return y;}
      }
public class Distance {
  public static void main(String[] args){
//локальные переменные не являются членами класса
    Locate t1 = new Locate();
    Locate t2 = new Locate();
    double dx,dy,distance;
t1.setXY(5, 10);    t2.setXY(2, 6);
  dx = t1.getX() - t2.getX();
   dy = t1.getY() - t2.getY();
distance= Math.sqrt(dx*dx+dy*dy);//Вычисление расстояния
System.out.println("Расстояние равно: "+distance);     }
}
```

Вывод: **Расстояние равно: 5.0**

Здесь используется статический метод **sqrt()** из класса **Math** без объявления объекта указанного класса. Класс **Math** содержит методы для физических и технических расчетов, а также константы **E** и **PI**.

Создание объекта класса – это двухшаговый процесс. Сначала объявляется ссылка на объект класса. Затем с помощью оператора **new** создается экземпляр объекта, например следующим образом:

```
Locate t1; //Объявление ссылки
t1 = new Locate(); //Создание объекта
```

Часто эти два этапа объединяются в один:

```
Locate t1 = new Locate();
```

Оператор **new** вызывает конструктор, поэтому в круглых скобках могут стоять аргументы, передаваемые конструктору. Ссылки на объект аналогичны указателю, однако нельзя поставить ссылку на произвольный участок памяти или манипулировать с ней как с целым числом. Операция присваивания для объектов означает, что две ссылки будут указывать на один и тот же участок памяти. Операции сравнения ссылок типа == имеют мало смысла. Например:

```
// пример # 6 : сравнение строк и объектов : ComparingStrings.java
class ComparingStrings {
        public static void main(String[] args) {
                String s1= "abc" , s2;
                 s2 = s1; // переменная ссылается на ту же строку
System.out.println("одинаковые объекты? "+(s1 == s2)); //true
```

```
s2 = new String(s1); //создание нового объекта копированием s1
System.out.println("одинаковые объекты? " + (s1 == s2));//false
System.out.println("одинаковые значения? " + s1.equals(s2));//true
    }
}
```

Упражнения

1. Выполнить приведенные выше приложения и апплет.
2. Какие отличия выполнения приложения от выполнения апплета?
3. Написать приложение, выводящее три строки с переходом на новую строку и без перехода.
4. Написать приложение для ввода пароля из командной строки и сравнения его со строкой-образцом.

2. ТИПЫ ДАННЫХ И ОПЕРАЦИИ. ОПЕРАТОРЫ УПРАВЛЕНИЯ ПРОГРАММОЙ

Комментарии Java // и /* */ аналогичны C++ . Введен новый вид комментариев /** */, из таких комментариев соответствующие утилиты могут извлекать текст для документирования программ.

Базовые типы данных:

Тип	Размер	По умолчанию	Примеры
boolean	**8**	**false**	**true , false**
byte	**8**	**0**	**–128..128**
char	**16**	**' \x0'**	**'a', '\x20' , '\n'**
short	**16**	**0**	**–32768..32767**
int	**32**	**0**	**–2147483648.. 2147483647**
long	**64**	**0**	**9223720368547758O8L**
float	**32**	**0.0f**	**3.40282347E+38**
double	**64**	**0.00**	**1.797693134486231570E+308**

Тип **char** использует формат UNICODE, со значениями кодов от 0 до 65535.

Кроме базовых типов данных широко используются объекты классов **Boolean, Character, Integer, Long, Float, Double.** Данные классы находятся в библиотеке **java/src/lang,** которая содержит абстрактный класс **Number**, представляющий собой интерфейс для работы со всеми скалярными типами, его подклассы являются классами-оболочками для значений соответствующих типов. Переменную базового типа можно преобразовать к соответствующему объекту, передав ее значение конструктору при объявлении объекта.

Строка в Java представляет объект класса **String**. Можно использовать операцию "+" для объединения строк, а также и методы, имеющиеся в классе **String**. Строковые константы заключаются в двойные кавычки и не заканчиваются символом '**\0**', это не ASCII-строки, а массивы символов.

Для преобразования базовых типов друг к другу применяется операция *(mun),* где в качестве *(mun)* используется один из совместимых базовых типов. Для приведения объектов к другому типу используются методы **valueOf(), toString()** и др. Объекты класса могут быть преобразованы к базовому типу методами **intValue(), longValue()** и т.д.

```
//пример #1 : преобразование типов данных : Types.java
public class Types{
   public static void main(String[] args) {
      String s = Float.toString(10.0F); //Float в String
      String s1 = String.valueOf(3.14159); //Float в String
      Integer i = Integer.valueOf("2002"); //String в Integer
      int n = i.intValue(); //Integer в int
      double d = (double)n; //int в double
      Character ch = new Character('3');
      int m = Character.digit(ch); //Character в int
    System.out.println("s="+s+", s1="+s1);
    System.out.println("i="+i+", n="+n+", d="+d+", m="+m);
    }
   }
```

Java включает два класса для работы с высокоточной арифметикой: **BigInteger** и **BigDecimal,** которые поддерживают целые числа и числа с фиксированной точкой произвольной точности.

Операторы

Операторы Java практически совпадают с операторами C++ и имеют такой же приоритет. Поскольку указатели отсутствуют, то отсутствуют операторы *, &, ->, delete для работы с ними.

Операции над целыми числами: **+, –, *, %,** /, **++,—** и побитовые операции **&,** |, ^, ~ аналогичны C++. Деление на ноль вызывает исключительную ситуацию, переполнение не контролируется.

Операции над числами с плавающей точкой те же, что и в C++, но по стандарту IEEE 754 введена бесконечность **+inf** и **–inf**. Кроме этого существует значение **NaN** (Not a Number), которое возвращается, например, при извлечении квадратного корня из отрицательного числа.

Логические операции выполняются над значениями типа **boolean** (**true** или **false**) и могут быть: **&&-"и", ||-"или", !-"не", ^-"!или"**.

Операторы и их возможные сочетания:

+	+=	–	–=
*	*=	/	/=
|	|=	^	^=
&	**&=**	**%**	**%=**
>	>=	<	<=
!	**!=**	++	—
>>	>>=	<<	<<=
>>>	>>>=	**&&**	||
==	=	~	**?:**
	instanceof	**[ ]**	

Так же как и в C++ может быть использована операция **a** *op* **= b**, эквивалентная **a = a** *op* **b.** Здесь *op* – одна из перечисленных операций. Операторы управления аналогичны операторам C++.

```
if (boolexp) { /*операторы*/}
else { /*операторы*/ }

while (boolexpr) { /*операторы*/ }

do { /*операторы*/ }
while (boolexp);

for(exp1;boolexp;exp3){ /*операторы*/ }
```

Циклы выполняются, пока булевское выражение *boolexp* равно **true**. Аналогично C++ используется и оператор **switch:**

```
switch(exp) {
 case exp1:/*операторы, если exp=exp1*/ break;
case exp2: /*операторы, если exp=exp2*/ break;
default: /* операторы Java */ }
```

Исключение составляют операторы **break, continue**, которые можно использовать с меткой, для обеспечения выхода из вложенных циклов, например:

```
//выход за цикл, помеченный OutHire
OutHire:
while(i < 100){
  while(j < 10){
```

```
    if(j == 0) break OutHire;} }
```

Тернарный оператор "**?**" используется в выражениях:

booleanexp **? value0 : value1**

Если **booleanexp = true**, вычисляется **value0** и оно становится результатом, иначе вычисляется **value1** и оно становится результатом.

Можно проверить, является ли объект экземпляром данного класса с помощью оператора **instanceof,** например:

```
if (MyClass instanceof java.awt.Font) { /*операторы*/}
```

Массивы

В следующем примере рассматривается работа с массивами. Так как массивы являются динамическими, для создания массива требуется выделение памяти с помощью оператора new или инициализации. Имена массивов при этом являются ссылками.

```
/* пример #2  : замена отрицательных элементов массива на максимальный : FindReplace.java*/
class FindReplace{
  public static void main(String args[])  {
        int myArray[];                    //объявление без инициализации
        int mySecond[] = new int[100]; //выделение памяти
int a[]={5,7,10,0,-5,-8,-16,6,-2,-10}; //объявление с инициализацией
int max = a[0];
   for(int i = 0; i < a.length; i++) //поиск max-элемента
        if(max < a[i])  max = a[i];
    for(int i = 0;i < a.length; i++)   { //замена
      if( a[i] < 0 ) a[i] = max;
        mySecond[i] = a[i];
   System.out.println("a[" +i+ "]= " +a[i]);     }
myArray = a; //установка ссылки на массив a
  }
 }
```

Многомерных массивов в Java не существует, зато имеются массивы массивов. Для задания начальных значений массивов существует специальная форма инициализатора, например:

```
int array[][] = { { 0, 1, 2, 3 },  { 4, 5, 6, 7 },  { 9, 7, 5, 3 }, { 0, 3, 6, 9 }   };
```

Следующая программа создает двумерные массивы, инициализирует их и выполняет произведение одного на другой.

```
// пример #3  : умножение двух матриц : Matrix.java
```

```
public class Matrix {
	public int[][] a;
public Matrix(int n, int m) {
	a = new int[n][m];
	for (int i = 0; i < n; i++)
		for (int j = 0; j < m; j++) {
			a[i][j] = (int) (Math.random() * 10);
		}
	show(a);
	}
public Matrix(int n, int m, int k) {
	a = new int[n][m];
	for (int i = 0; i < n; i++)
		for (int j = 0; j < m; j++) {
			a[i][j] = k;
		}
		show(a);
}
public void show(int[][] arr) {
System.out.println("Матрица : "+arr.length+"  "+arr[0].length);
	for (int i = 0; i < arr.length; i++) {
		for (int j = 0; j < arr[0].length; j++) {
			System.out.print(arr[i][j] + " ");
		}System.out.println();
	}
}
public static void main(String[] args) {
	int n = 2, m = 3, l = 4;
	Matrix p = new Matrix(n, m);
	Matrix q = new Matrix(m, l);
	Matrix r = new Matrix(n, l, 0);
	for (int i = 0; i < p.a.length; i++)
		for (int j = 0; j < q.a[0].length; j++)
			for (int k = 0; k < p.a[0].length; k++) {
				r.a[i][j] += p.a[i][k] * q.a[k][j];
			}
	r.show(r.a);
   }
}
```

Переменные и константы

Переменные класса (атрибуты или поля), объявляются в виде:
спецификатор тип имя;

В языке Java используются переменные класса, объявленные один раз для всего класса со спецификатором **static** и переменные экземпляра, создаваемые для каждого экземпляра класса. Переменные объявляются со спецификаторами доступа **public, private, protected, private protected** или по умолчанию без спецификатора. Кроме этого, переменные со спецификатором **final** являются константами.

```
//пример #3 : типы переменных : MyClass.java
class MyClass {
  static int bonus; // переменная класса
  int x; // переменная экземпляра класса
  final int YEAR = 2002; // константа
    // методы
   }
```

Переменные, объявленные в методах, или параметры методов не являются членами класса. Спецификатор **final** для таких переменных использовать нельзя.

Ограничение доступа

Java предоставляет несколько уровней защиты, обеспечивающих возможность настройки области видимости данных и методов. Из-за наличия пакетов Java должна уметь работать с четырьмя категориями видимости между элементами классов:

- подклассы в том же пакете;
- независимые классы в том же пакете;
- подклассы в различных пакетах;
- классы, которые не являются подклассами и не входят в тот же пакет.

Элемент, объявленный **public**, доступен из любого места. Все, что объявлено **private**, доступно только внутри класса и нигде больше. Если у элемента вообще не указан модификатор уровня доступа, то такой элемент будет виден из подклассов и классов того же пакета. Именно такой уровень доступа используется по умолчанию. Если же необходимо, чтобы элемент был доступен извне пакета, но только подклассам того класса, которому он принадлежит, нужно объявить такой элемент **protected**. Если требуется, чтобы элемент был доступен

только подклассам, независимо от того, находятся ли они в данном пакете или нет, используется комбинация **private protected.**

Упражнения

1. Какое значение получит переменная h после выполнения операторов?

```
{int f = 2, g = 5, h;      h = 3+f/g+5*f%g; }
```

2. Как правильно вывести строку "Hello to the world!", при объявлении вида

```
String w[] = {"Hello ", "friends ","world! "};
String p[] = {"it ","to ","that ","the "};
```

а)System.out.println(w[0] || p[1] || p[3] || w[2]);

б)StdOut.writeln(w[1] + p[2] + p[4] + w[3]);

в)System.out.println(w[0] + p[1] + p[3] + w[2]);

г)System.out.println(w[1] + p[2] + p[4] + w[3]);

3. Какое значение надо присвоить переменной X ,чтобы вывести 10 строк?

```
{int count = 0;
 while(count < X ) { System.out.println("Строка : " + count++);} }
```

4. При каком значении X будут выведено 5 строк?

```
int count = 0;
while(++count < X ) { System.out.println("Строка " + count);}
```

5. При каком значении X будут выведены все элементы массива?

```
{int[] values = {1,2,3,4,5,6,7,8};
for(int i=0;i< X; i++)
System.out.println(values[i]); }
```

6. Какие значения получат переменные x,y,z после выполнения кода?

```
{int x=5, y=2, z=3;
if(x >= y) { x /= --z;
if(z != x) y = (x + z)/y; }
else if(y == 2) { y++; z *= y + x; }
else { y = 8; x *= y; }}
```

а)x=2, y=13, z=10; б)x=16, y=8, z=10; в)x=2, y=2, z=2; г)x=5, y=3, z=24.

7. Значение какого типа может быть возвращено методом check4Biz()?

```
if(check4Biz(storeNum) < 10) {}
```

8. Как вызвать метод класса объявленный со спецификатором private?

9. Задать массив строк. Найти самую длинную и самую короткую строки?

3. КЛАССЫ КАК НОВЫЕ ТИПЫ ДАННЫХ. ПОЛЯ ДАННЫХ И МЕТОДЫ

Системная библиотека классов языка Java содержит классы и пакеты, реализующие различные базовые возможности языка. Методы классов, включенных в эту библиотеку, вызываются из JVM во время интерпретации Java-программы. В системе Java вся программа интерпретируется на JVM, так что возможности расширять язык, вводя все новые и новые классы и пакеты в системную библиотеку, практически неограничены.

Один класс (подкласс) может наследовать переменные и методы другого класса (суперкласса), используя ключевое слово **extends**. Подкласс имеет доступ ко всем открытым переменным и методам родительского класса, как будто они находятся в подклассе. В то же время подкласс может иметь методы с тем же именем и сигнатурой, что и методы суперкласса. В этом случае подкласс переопределяет методы родительского класса. В следующем примере перегружаемый метод **show()** находится в двух классах **Bird** и **Eagle**. По правилам Java вызывается метод, наиболее близкий к текущему объекту.

```
// пример #1: наследование и перегрузка методов : BirdSample.java
class Bird {
        float price;
        String wage;
  Bird(float p, String v) { price = p; wage = v; }//конструктор
void show(){
   System.out.println("вес: " + wage + ", цена: "+ price);}
}
 class Eagle extends Bird {
    boolean fly;
Eagle(float p, String v, boolean f) {
 super(p, v); } //вызов конструктора суперкласса
 fly = f; }
void show(){
System.out.println("вес:"+ wage+", цена: "+ price+ ", полет:"+fly);
  }
}
public class BirdSample {
    public static void main(String[] args) {
   Eagle whiteEagle = new Eagle(10.55F, "Белый Орел",true);
   Bird goose = new Bird(0.85F, "Гусь");
   goose.show(); // вызов show() класса Bird
   whiteEagle.show(); // вызов show() класса Eagle
  }
}
```

Способность Java делать выбор метода, исходя из типа времени выполнения, называется полиморфизмом. Поиск метода происходит сначала в данном классе, затем в суперклассе, пока метод не будет найден или не будет достигнут **Object** – суперкласс для всех классов. Отметим, что статические методы могут быть переопределены в подклассе, но не могут быть полиморфными, так как их вызов не затраги-

вает объекты. Нельзя создать подкласс для класса, объявленного со спецификатором **final**:

// ОШИБКА! class First не может быть суперклассом
final class First {/* код*/} // следующий class неверен
class Second extends First{/* код*/}

Методы классов

Все функции Java объявляются внутри классов и называются методами. Невозможно объявить функцию вне класса. Как и в C++ определение функции имеет вид:
retyrn_type function_name(arglist)
{//instructions
return value;}

Если метод не возвращает значение, ключевое слово **return** отсутствует, тип возвращаемого значения будет **void** или отсутствует. Вместо пустого списка аргументов **void** не указывается. Вызов методов осуществляется из объекта или класса (для методов класса):
Objekt_name.method_name(arglist);

Отметим, что методы-конструкторы вызываются автоматически при создании объекта класса с помощью оператора **new**. Автоматически вызывается метод **main()** при загрузке приложения, содержащего класс и метод **main().** При загрузке апплетов автоматически запускаются методы **init(), start(), stop(), paint().**

Для того чтобы создать метод, нужно внутри объявления класса написать объявление метода и затем реализовать его тело. Объявление метода, как минимум, должно содержать тип возвращаемого значения (возможен **void**) и имя метода. В объявлении метода элементы, заключенные в квадратные скобки, являются необязательными.
[доступ][static][abstract][final][native][synchronized]
тип_возвращаемого_значения имя_метода(список параметров)
[throw список_исключений]

Как и для полей данных спецификатор доступа к методам может быть **public, private, protected, private protected** и **friendly** (по умолчанию).

Статические методы

Переменные, объявленные как **static**, являются общими для всех объектов класса и называются переменными класса. Если один объект изменит значение такой переменной, то это изменение увидят все

объекты. Для работы со статическими переменными используются статические методы, объявленные со спецификатором **static**. Такие методы являются методами класса и не содержат указателя **this** на конкретный объект. Вызов такого метода возможен с помощью указания: *имя_класса.имя_метода*

```
public class MyStaticShow {
public static int callMethod(){/* код*/}      }
x = MyStaticShow.callMethod(); // вызов статического метода.
```

Можно вызывать такие методы, например, из класса **Math** без объявления объекта:

```
z = Math.max(x,y);
k = Math.random(); // случайное значение
```

Final-методы

Спецификатор **final** используется для определения констант. Методы, объявленные как **final,** нельзя замещать в подклассах. Например:

```
class A {
final void method() { System.out.println("Это final метод."); }
          }
class B extends A {
void method() { /* ОШИБКА! Нельзя переопределять.*/ }          }
```

Абстрактные методы

Абстрактные методы помещаются в абстрактных классах или интерфейсах, тела таких методов отсутствуют и реализуются в подклассах.

```
public abstract class AbstractShow{
public abstract void abstractMethod(); }
```

Модификатор native

Программа может вызывать методы, написанные на C++. Такие методы объявляются с ключевым словом **native**, которое сообщает компилятору, что метод реализован в другом месте. Например:

```
static native void outFunction(int x);// outFunction() функция C++
```

Модификатор synchronized

При использовании нескольких потоков необходимо синхронизировать методы, обращающиеся к общим данным. Когда компилятор

обнаруживает **synchronized**, он включает код, блокирующий доступ к данным при запуске потока и снимающий блок при его завершении.

Упражнения

1. Какие свойства имеет метод класса, если он объявлен как public final.
2. Объяснить отличия базовых типов Java от C++. Чем отличаются типы boolean и Boolean, int и Integer, String и char[]?
3. Изменение каких данных из одного экземпляра класса влечет их изменение для всех экземпляров класса?
4. Какие свойства имеет метод, если он объявлен как public static?
5. Объявить базовый и производные классы на основе Number,Vector, Matrix**.** Создать и проинициализировать объект класса Matrix.
6. Создать класс Fraction (дробь) и его подкласс MixFraction (смешанная дробь).

4. ПОЛИМОРФИЗМ. КОНСТРУКТОРЫ

Перегрузка и переопределение методов

Полное имя метода включает его метода, класс и параметры. Если два метода с одинаковыми именами находятся в одном классе, списки параметров должны отличаться. Такие методы являются перегружаемыми. Если метод подкласса совпадает с методом суперкласса (порождающего класса), то метод подкласса переопределяет метод суперкласса. Все методы Java являются виртуальными (ключевое слово virtual как в C++ не используется). Переопределение методов является основой концепции динамического связывания, реализующей полиморфизм. Когда переопределенный метод вызывается через ссылку суперкласса, Java определяет, какую версию метода вызвать, основываясь на типе объекта, на который имеется ссылка. Таким образом, тип объекта определяет версию метода на этапе выполнения. В следующем примере рассматривается реализация полиморфизма на основе динамического связывания. Так как суперкласс содержит методы, переопределенные подклассами, то объект суперкласса будет вызывать методы различных подклассов, в зависимости от того, на объект какого подкласса у него имеется ссылка.

```
//пример #1: динамическое связывание методов : DynDispatch.java
class A {
   int i, j;
  A(int a, int b) {
        i = a; j = b; }
  void show() { // вывод i и j
    System.out.println("i и j: " + i + " " + j); }
```

```
 }
class B extends A {
   int k;
     B(int a, int b, int c) {  super(a, b);   k = c;  }
  void show() { // вывод k: переопределенный метод show() из A
    super.show(); // вывод значений из A
    System.out.println("k: " + k);
  }
}
class C extends B {
  int m;
    C(int a, int b, int c, int d) { super(a, b, c);  m = d;  }
void show() { // вывод m: переопределенный метод show() из B
      super.show(); //вывод значений из B
      System.out.println("m: " + m); }
 }
class DynDispatch {
   public static void main(String[] args) {
     A Aob;
     B Bob = new B(1, 2, 3);
     C Cob = new C(5, 6, 7, 8);
     Aob = Bob; // установка ссылки на Bob
  Aob.show(); // вызов show() из B
  System.out.println();
  Aob = Cob; // установка ссылки на Cob
  Aob.show(); // вызов show() из C
 }
}
```

Результат: **i и j: 1 2**
k: 3
i и j : 5 6
k:7
m: 8

Отметим, что при вызове **show()** обращение **super** всегда происходит к ближайшему суперклассу.

В следующем примере экземпляр подкласса создается с помощью **new,** ссылка на него передается объекту суперкласса. При вызове из суперкласса соответственно вызывается метод подкласса.

```
// пример #2 : динамического вызов метода : Dispatch.java
class A {
```

```
   void method() { System.out.println("метод класса A");}
}
class B extends A {
   void method () { System.out.println("метод класса B");}
}
public class Dispatch {
  public static void main(String[] args) {
     A ob_a = new B();
     ob_a. method ();
   }
 }
```

Вывод: **метод класса B**

В следующим примере приведение к базовому типу происходит в выражении: **People s = new Men(); People s = new Women();**

Базовый класс **People** предоставляет общий интерфейс для всех наследников. Порожденные классы перекрывают эти определения для обеспечения уникального поведения.

```
//пример # 3: полиморфизм: Peoples.java
class People {
  void add() {/*пустая реализация*/}
  void delete() {}
}
class Men extends People {
  void add() { System.out.println("добавлен сотрудник-мужчина");}
  void delete() {System.out.println("удален сотрудник-мужчина");}
}
class Women extends People {
  void add() {System.out.println("добавлен сотрудник-женщина ");}
  void delete() {System.out.println("удален сотрудник-женщина");}
}
public class Peoples {
  public static People randPeople() {
     switch((int)(Math.random() * 2)) {
  case 0: return new Men();
  case 1: return new Women();}
}
public static void main(String[] args) {
  People[] s = new People[10];
 for(int i = 0; i < s.length; i++) // заполнение массива сотрудниками
    s[i] = randPeople();
```

```
  for(int i = 0; i < s.length; i++)
    s[i].add();// вызов полиморфного метода
   }
}
```

Главный класс **Peoples** содержит **static** метод **randPeople()**, который возвращает ссылку на случайно выбранный объект подкласса класса **People** каждый раз, когда он вызывается. Приведение к базовому типу производится оператором **return**, который возвращает ссылку на **Men** или **Women.** Функция **main()** содержит массив из ссылок **People,** заполненный вызовами **randPeople()**. На этом этапе известно, что имеется некоторое множество ссылок на объекты базового типа и ничего больше (не больше, чем знает компилятор). Когда происходит перемещение по этому массиву, метод **add()** вызывается для каждого случайным образом выбранного объекта**.**

Конструкторы

Конструктор – это метод, который автоматически вызывается при создании объекта класса и выполняет действия по инициализации объекта. Конструктор имеет то же имя, что и класс; вызывается не по имени, а только вместе с ключевым словом **new** . Конструктор не возвращает значение, но может иметь параметры и быть перегружаемым.

```
//пример # 4 : перегрузка конструктора: Book.java
class Book{
  String title, publisher;
  float price;
Book(String title, String publisher, float price) {/* конструктор*/}
Book(String t, float p){  //второй конструктор
  title = new String(t);
  price = p; }
void printBookInfo(){/*реализация*/}
}
```

Объект класса **Book** может быть создан двумя способами, вызывающими один из конструкторов:

```
Book tips = new Book("Java 2","П.Ноутон, Г.Шилдт",49.95);
Book tips = new Book("Java 2",49.95);
```

Объявление можно отделить от инициализации:

```
Book tips ; // объявление
tips = new Book("Java 2",49.95);// инициализация
```

Если конструктор в классе не определен, Java предоставляет конструктор по умолчанию, который инициализирует объект значениями по умолчанию.

Использование super и this

Ключевое слово **super** используется для вызова конструктора суперкласса и для доступа к члену суперкласса. Например:
super(список_параметров); // вызов конструктора суперкласса
super.i = a; //присваивание значения члену суперкласса
super(x,y); //передача параметров конструктору базового класса

Вторая форма **super** подобна ссылке **this** на экземпляр класса. Каждый экземпляр класса имеет неявную ссылку **this** на себя, которая передается также и методам. После этого можно писать **this.good; this.price**, хотя и необязательно. Ссылку **this** можно использовать в методе для доступа к переменной класса, если в методе есть локальная переменная с тем же именем.

Следующий код показывает, как, используя **this** и перегрузку методов, можно строить одни конструкторы на основе других.

```
//пример # 5 : this в конструкторе : Locate.java
class Locate {
  int x, y;
Locate(int x, int y) {
  this.x = x; this.y = y; }
  Locate() { this(-1, -1); }
 }
```

В этом фрагменте второй конструктор для завершения инициализации объекта обращается к первому конструктору.

Абстрактные классы

Абстрактные классы объявляются с ключевым словом **abstract** и содержат объявления абстрактных методов, которые не реализованы в этих классах. Объекты таких классов создать нельзя, можно создать объекты подклассов, которые реализуют эти методы.

```
//пример #6: абстрактные методы и классы: AbstractDemo.java
abstract class Square {
        abstract int squareIt(int i); //абстрактный метод
        void show(){/*реализация*/}
}
//squareIt() должен быть реализован подклассом Square
```

```
class SquareTwo extends Square {
   int squareIt(int i) { return i*i; }
}
class AbstractDemo {
   public static void main(String[] args) {
     SquareTwo ob = new SquareTwo();
     System.out.println("10 в квадрате равно " + ob.squareIt(10));
  }  }
```

Результат: **10 в квадрате равно 100**

Методы **finalize** аналогичны деструкторам в C++. Исполняющая среда Java будет вызывать его каждый раз, когда сборщик мусора соберется уничтожить объект этого класса.

Упражнения

1 .Записать конструктор для массива объектов, объявленных следующим образом MyClass b[3]={1,2,3};

2. Проинициализировать объект производного класса, если конструктор базового класса задан с двумя параметрами

3. Как объявить массив объектов абстрактного класса?

4. В приведенном ниже примере рассматривается класс двоичное дерево. Объяснить, как строится дерево, как подсчитывается число включений слова, как вводятся строки и выделяются слова.

```
import java.io.*;
import java.util.*;
class Node {String word;
          int count;
          Node left, right;
  Node (String new_word) {word = new_word; count = 1;
    left = right = null;   }
}
class Btree {Node root;
    Btree() {  root = null; }
  void insert(Node node, String new_word) {
    int compare = node.word.compareTo(new_word);
    if (compare == 0) {node.count++; }  // слово уже существует
    else if (compare > 0) {
       if (node.left != null) insert (node.left, new_word);
       else node.left = new Node(new_word);
    }
    else if (compare < 0) {
       if (node.right != null) insert (node.right, new_word);
       else node.right = new Node(new_word);
                    }
  }
  void insert_word (String new_word) {
    if (root == null) { root = new Node(new_word);}
```

```
    else { insert (root, new_word); }
  }
  Node find(Node node, String keyword) {
    int compare = node.word.compareTo(keyword);
    if (compare == 0)  return node; // найдено ключевое слово
     else if (compare > 0) {
      if (node.left != null)  return find (node.left, keyword);
      else return null; /* ключевое слово не найдено*/
    }
     else if (compare < 0) {
      if (node.right != null) return find (node.right, keyword);
     else return null;
    }
    return null;
  }
  int count_word (String keyword) {
    if (root != null) {
       Node node = find (root, keyword);
       if (node != null)  return node.count;
       else  return 0;
                  }
    return 0; // дерево пусто
  }
}
public class Wordtree {
  public static void main (String args[]) {
    Btree btree = new Btree();
    try {// открытие файла и вставка слова в двоичное дерево
FileReader is = new FileReader("Wordtree.java");
BufferedReader bis = new BufferedReader(is);
String thisLine;
       while ((thisLine = bis.readLine()) != null) {
         StringTokenizer st = new StringTokenizer(thisLine," \t\n\r");
         while (st.hasMoreTokens()) {
           btree.insert_word (new String (st.nextToken())); }
       }
     } catch (Exception e) { System.out.println("ошибка: " + e); }
    // получение числа слов
System.out.println("Число слов ' Java ' = "+btree.count_word("Java"));
  }
}
```

5. ИНТЕРФЕЙСЫ. ПАКЕТЫ

Интерфейсы представляют полностью абстрактные классы: ни один из объявленных методов не может быть им реализован. Все ме-

тоды автоматически трактуются как **public** и **abstract,** а все переменные – как **public, static** и **final**. Каждый интерфейс может быть реализован одним или несколькими классами, не связанными по иерархии наследования. Класс может реализовывать любое число интерфейсов, указываемых после ключевого слова **implements**, дополняющего определение класса. На множестве интерфейсов тоже определена иерархия по наследованию, но она не имеет отношения к иерархии классов. В языке Java интерфейсы обеспечивают большую часть той функциональности, которая в C++ представляется с помощью механизма множественного наследования. Определение интерфейса имеет вид:
*[public] interface имя [extends I1,I2,…,IN]{/*реализация*/}*
Класс также может наследовать любое число интерфейсов:
[доступ] class имя_ класса implements I1,I2, …,IN {/…реализация…*/}*

Класс, который реализует интерфейс, должен предоставить полную реализацию всех методов, объявленных в интерфейсе. Кроме этого, данный класс может объявлять свои собственные методы. Если класс включает интерфейс, но полностью не реализует его методы, то этот класс должен быть объявлен как **abstract.**

```
// пример #1 : интерфейс и его реализации : InterfacesDemo.java
interface Auto{
 public String getMark();
 public String getPassangers();
 public String getWheels();
  }
class Bus implements Auto{
  public String getMark(){return "Mercedes";}
  public String getPassangers(){return "90 человек";}
  public String getWheels(){return "6 колес";}
}
class Car implements Auto{
  public String getMark(){return "BMV";}
  public String getPassangers(){return "5 человек";}
  public String getWheels(){return "4 колеса";}
}
public  class InterfacesDemo {
      public static void main(String[] args){
  Car c = new Car();
  Bus b = new Bus();
      printFeatures(c);
      printFeatures(b);
```

```
}
public static void printFeatures(Auto f){
 System.out.println("марка:"+f.getMark()+" вместимость:"+
f.getPassangers()+" количество колес: "+f.getWheels());
  }
}
```

Класс **InterfacesDemo** содержит метод **printFeatures()**, который вызывает методы того объекта, который передается ему в качестве параметра. Вначале ему передается объект, соответствующий легковому автомобилю, затем автобусу (объекты **c** и **b**). Каким образом метод **printFeatures()** может обрабатывать объекты двух различных классов? Все дело в типе передаваемого этому методу аргумента – класса, реализующего интерфейс **Auto.** Вызывать, однако, можно только те методы, которые были объявлены в интерфейсе.

В следующем примере объявляется объектная ссылка, которая использует интерфейсный тип. В такой переменной можно сохранять экземпляр любого класса, который реализует объявленный интерфейс. Когда вызывается метод через такую ссылку, то будет вызываться его правильная версия, основанная на текущем экземпляре. Выполняемый метод разыскивается динамически во время выполнения, что позволяет создавать классы позже кода, который вызывает их методы.

```
// пример #2: динамический вызов функций: TestCall.java
interface Callback {
  void callB(int param); }
class Client implements Callback {
  public void callB(int p) {
    System.out.println("метод callB() вызван со значением = " + p);
  }
  void myMethod() {
     System.out.println("класс, реализующий интерфейс " +
     "может иметь свои собственные методы");
  }
}
class XClient implements Callback {
  public void callB(int p) {
    System.out.print("другая версия метода callB():");
     System.out.println("p в квадрате = " + (p*p));
  }
}
class TestCall{
```

```
    public static void main(String[] args) {
      Callback c = new Client();
      XClient ob = new XClient();
   c.callB(11);
   c = ob; // c присваивается ссылка на другой объект
   c.callB(11);
   }
}
```

Результат: **метод callB() вызван со значением = 11**
другая версия метода callB(): p в квадрате = 121

Пакеты

Оператор **package**, помещаемый в начале исходного программного файла, определяет пакет, т. е. область в пространстве имен классов, в которой определяются имена классов, содержащихся в этом файле. Внутри указанной области можно выделить подобласти, используя тот же оператор **package**; действие оператора **package** аналогично действию объявления директории на имена файлов. Для обеспечения возможности использования коротких имен классов, помещенных в другие пакеты, используется оператор **import**. Приведем общую форму исходного файла Java:

одиночный оператор package (необязателен);
любое количество операторов import (необязательны);
*одиночное объявление открытого (public) класса {/*код*/}*
*любое количество закрытых (private) классов пакета (необязательны) {/*код*/}*

Каждый класс принадлежит некоторому пакету и добавляется в пакет при компиляции. Для добавления класса в какой-то пакет указывается этот пакет после слова **package.** Например:

```
// пример #3 : простейший пакет : AccountBalance.java
package com.MyPack;
class Balance {
  String name;    double bal;
Balance(String n, double b) {name = n; bal = b; }
void show() {
   if(bal < 0) System.out.print("-->> ");
   else System.out.println(name + ": $" + bal);
   }
}
public class AccountBalance {
```

```
    public static void main(String[] args) {
        Balance current[] = new Balance[3];
current[0] = new Balance("Билл Гейтс", 150000000.0);
current[1] = new Balance("Борис Березовский", -170000.02);
current[2] = new Balance("Ясир Арафат", 666000.11);
    for(int i = 0; i < 3; i++) current[i].show();
  }
}
```

Файл начинается с объявления того, что данный класс принадлежит пакету **MyPack.** Другими словами это значит, что файл **AccountBalance.java** находится в каталоге /**com/MyPack.** Нельзя переименовывать пакет, не переименовав каталог, в котором хранятся его классы.

Если пакет не существует, он создается при первой компиляции, если пакет не указан, класс добавляется в неименованный **(unnamed)** пакет. Чтобы получить доступ к классу из другого пакета, перед именем класса указывается имя пакета: **Pakagename.Classname**. Чтобы избежать таких длинных имен, используется ключевое слово **import.**

```
import javax.swing.JApplet;
import java.awt.*;
```

Каталог, который транслятор Java будет рассматривать как корневой для иерархии пакетов, можно задавать с помощью переменной среды окружения **CLASSPATH**.

Упражнения

1. Что неправильно в объявлении класса C:

```
class C extends B, A implements Runnable, D {}
```

2.
```
public interface A {
static final int myCount = 10;
abstract public void method1(int i);
abstract public int method2(float f);
class B implements A { }
```

Чтобы расширить A класс B должен сделать?

а)быть в том же пакете, что и A

б)объявить статическую версию method1(),method2()

с) включить неабстрактную версию method1(), method2().

3. Когда будет позволено преобразование b к классу C?

```
 B b = new B();
 C c = (C) b;
```

а)B суперкласс C; б)C final-класс; с)B и C подклассы одного суперкласса;

д)B подкласс C

6. КЛАСС String. ПОТОКИ ВВОДА/ВЫВОДА

Класс String

Системная библиотека Java содержит классы **String** и **StringBuffer**, поддерживающие работу со строками и определенные в пакете **java.lang**. Эти классы объявлены как **final**, что означает, что от этих классов нельзя производить подклассы. Класс **String** поддерживает несколько конструкторов, например **String(), String(String strobj), String(char[]), String(byte asciichar[])**. Когда Java встречает объект литерал, заключенный в двойные скобки, автоматически создается объект типа **String**. Таким образом, объект класса **String** можно создать, присвоив ссылке на класс значение существующего литерала, или с помощью **new** и конструктора. Класс **String** содержит основные методы для работы со строками: слияние строк (**concat(s)** или +), сравнение строк (е**quals(s)** и **compareTo(s)),** извлечение символа и подстроки (**charAt(n)** и **substring(n,m)**). Длина строки определяется с помощью метода **length()**. Преобразование объекта к строке осуществляется с помощью методов **toString()** (**Integer.toString()**, **Float.toString()** и т.д.) или **valueOf() (String.valueOf())**. Методы **s.toUpperCase()/s.toLowerCase()** преобразует все символы строки **s** в верхний/нижний регистр. Метод **replace(char,char)** заменяет в строке все вхождения первого символа вторым символом.

Класс **StringBuffer** является близнецом класса **String**, но, в отличие от последнего, объекты которого нельзя изменять (новая строка – это новый объект класса **String**), строки, являющиеся объектами класса **StringBuffer**, можно изменять. Объекты этого класса можно преобразовать к классу **String** методом **toString()**. Установка длины буфера осуществляется методом **setLength(n)**. Для добавления и вставки символов или строк используются методы **append(s)** и **insert(n,s)**.

В следующем примере массив символов преобразуется в **String**, затем в **StringBuffer**, и использованы методы этих классов.

```
// пример #1 : использование методов : GetSetChar.java
public class GetSetChar {
    public static void main(String[] args) {
    char s[]={'J','a','v','a'};
    String str = new String(s);
    StringBuffer sbuf = new StringBuffer(str);
    str=str.toUpperCase();//JAVA
System.out.println("до изменения = "+str + sbuf); //JAVAJava
sbuf.setLength(5);  sbuf.setCharAt(4, '2');
```

```
System.out.println("после :" +sbuf +str.substring(0,3) +
sbuf.charAt(3));  //Java2JAVa
        }
      }
```

В следующем примере рассмотрен метод **equals(),** который сравнивает строку **String** с указанным объектом.

```
// пример #2: использование equals() : EqualStrings.java
public class EqualStrings {
  public static void main(String[] args) {
    String s1 = "Java", s2 = "Java";
    String s3 = "Visual Age", s4 = "JAVA";
System.out.println(s1+"=="+s2+" ответ: " + s1.equals(s2));//true
System.out.println(s1+"=="+s3+" ответ: "+ s1.equals(s3));//false
System.out.println(s1+"=="+s4+" ответ: "+s1.equals(s4));//false
  }
}
```

В следующем примере рассмотрена сортировка массива строк методом "пузырька".

```
// пример #3 : сортировка : SortArray.java
class SortArray  {
  public static void main(String args[])  {
String a[] = {"Vika", "Natasha", "Alina", "Dima", "Denis"};
   for(int j = 0; j < a.length; j++)
   for(int i = j+1; i < a.length; i++)
     if(a[i].compareTo(a[j]) < 0)   {
 String t = a[j]; a[j] = a[i]; a[i] = t; }
   for(int i = 0; i < a.length; i++)
 System.out.print(a[i]+"   ");
   }
```

Потоки ввода/вывода.

Потоки ввода последовательности байт являются подклассами абстрактного класса **InputStream**, потоки вывода последовательности байт – подклассами абстрактного класса **OutputStream**. При работе с файлами используются подклассы этих классов соответственно **FileInputStream** и **FileOutputStream**, конструкторы которых открывают поток и связывают его с соответствующим файлом. Для чтения байта или массива байт используются перегружаемый метод **read()**

или **read(byte[] b)**. Метод возвращает −1, если достигнут конец потока данных.

```
//пример #4 : чтение байтов из потока ввода : ReadBytes.java
import java.io.*;
public class ReadBytes {
   public static void main(String[] args){
     int b;
 try {
FileInputStream is = new FileInputStream("ReadBytes.java");
    while ((b = is.read()) != -1){
//прочитанные данные выводятся на консоль
   System.out.println("прочитан байт = " + b);
   }
  is.close(); //закрытие потока ввода
}catch (IOException e){System.out.println("ошибка файла: " + e); }
  }
}
```

Конструктор **FileInputStream("ReadBytes.java")** открывает поток **is** и связывает его с файлом **ReadBytes.java.** Для закрытия потока используется метод **close()**. При чтении из потока можно пропустить **n** байт с помощью метода **skip(): long skip (long n).**

Для вывода байта или массива байт используются потоки вывода – объекты подкласса **FileOutputStream** класса **OutputStream**. В следующем примере для вывода байта в поток используется метод **write()**.

```
// пример # 5 : вывод байта в поток :WriteBytes.java
import java.io.*;
public class WriteBytes {
  public static void main(String[] args){
    int pArray[] = {1, 2, 3, 5, 7, 11, 13, 17};
 try {
FileOutputStream os = new FileOutputStream("bytewrite.dat");
   for (int i = 0; i < pArray.length; i++)
       os.write(pArray[i]);
       os.close();
 } catch (IOException e) { System.out.println("ошибка файла: "+e);}
   }
}
```

Библиотека классов ввода/вывода содержит класс **DataInputStream** для фильтрации ввода**,** методы которого преобра-

зуют введенные данные к базовым типам. Например, методы **readBoolean(), readByte(), readChar(), readInt(), readLong(), readFloat(), readDouble()** используются для ввода данных и преобразования к соответствующему типу.

Класс **DataOutputStream** позволяет записывать данные базового типа в поток вывода аналогично классу **DataInputStream** для ввода. При этом используются методы **writeBoolean(), writeChar(), writeInt(), writeLong(), writeFloat(), writeLine()**. В следующем примере рассматривается запись данных в поток и чтение из потока.

```
// пример #6 : запись в поток и чтение : DataStreams.java
import java.io.*;
public class DataStreams {
  public static void main(String[] args){// запись данных в файл
    try {
FileOutputStream os = new FileOutputStream("data.dat");
DataOutputStream ods = new DataOutputStream(os);
 ods.writeInt(48); // запись целого числа
 ods.writeFloat(3.1416f); // запись вещественного числа
 ods.writeBoolean(true); // запись логического значения
 ods.writeLong(725624); // запись значения типа long
 ods.close();
  }
catch(IOException e){
System.out.println("ошибка записи в файл: " + e);
}
   try { // чтение данных из файла
FileInputStream is = new FileInputStream("data.dat");
DataInputStream ids = new DataInputStream(is);
  int tempi = ids.readInt();// чтение целого числа
System.out.println(tempi);
  float tempf = ids.readFloat();// чтение вещественного числа
System.out.println(tempf);
  boolean tempb = ids.readBoolean();// чтение логического значения
System.out.println(tempb);
  long templ = ids.readLong();// чтение значения типа long
System.out.println(templ);
ids.close();
} catch (IOException e){System.out.println("ошибка " + e); }
  }
}
```

В отличие от классов **FileInputStream** и **FileOutputStream** суперкласс **RandomAccessFile** позволяет осуществлять произвольный доступ как к потокам ввода, так и вывода. Поток рассматривается при этом как массив байт, доступ к элементам осуществляется с помощью метода **seek(long poz).** Для создания потока можно использовать один из конструкторов:

RandomAccessFile(String name, String mode);
RandomAccessFile(File file, String mode);

Параметр **mode** равен **"r"** для чтения или **"rw"** для чтения и записи.

```
// пример #7 : запись и чтение из потока : RandomFiles.java
import java.io.*;
public class RandomFiles {
  public static void main(String[] args){
    int dataArr[] = {12, 31, 56, 23, 27, 1, 43, 65, 4, 99} ;
  try {
RandomAccessFile rf = new RandomAccessFile("temp.dat", "rw");
   for (int i = 0; i < dataArr.length; i++)
      rf.writeInt(data_arr[i]); // запись в файл
   for (int i = dataArr.length - 1; i >= 0; i--) {
   rf.seek(i*4); // длина каждой переменной типа int 4 байта
System.out.println(rf.readInt());// чтение в обратном порядке
   }
rf.close();
} catch (IOException e){
System.out.println("ошибка доступа к файлу: " + e);
   }
  }
}
```

Предопределенные потоки

Класс **System** пакета **java.lang** содержит поля **InputStream in, PrintStream out, err**, объявленные со спецификаторами **public static** и являющиеся стандартными потоками ввода, вывода и вывода ошибок соответственно. Эти потоки связаны с консолью, но могут быть переназначены на другое устройство. В настоящее время для консольного ввода используется не байтовый, а символьный поток. При этом для ввода используется подкласс **BufferedReader** абстрактного класса **Reader** и методы **read()** и **readLine()** чтения символа и строки.

Следующая программа демонстрирует ввод строки, числа и символа с консоли и вывод на консоль и в файл.

```
// пример #8 : ввод  с консоли : ConsoleInput.java
import java.io.*;
public class ConsoleInput {
   public static void main(String[] args){
InputStreamReader is = new InputStreamReader(System.in);
BufferedReader bistream = new BufferedReader(is);
try {char c;
    int number;
System.out.println("введите имя и нажмите <ввод>:");
  String nameStr = bistream.readLine();
System.out.println(nameStr+" введите число:");
  String numberStr = bistream.readLine();
  number = Integer.valueOf(numberStr).intValue();
System.out.println(nameStr+"  вы ввели число "+number);
System.out.println(nameStr+"  введите символ:");
  c = (char)bistream.read();
System.out.println(nameStr+"  вы ввели символ  "+c);
//Вывод в файл
PrintStream ps=new PrintStream(new FileOutputStream ("res.txt"));
   ps.println("привет "+nameStr+c+number);
ps.close();
}catch (IOException e){System.out.println("ошибка ввода "+ e); }
  }
}
```

Для вывода данных в файл в текстовом формате использовался фильтрованный поток вывода **PrintStream** и метод **println()**. Метод **valueOf(String str)** возвращает объект класса **Integer**, соответствующий цифровой строке, метод **intValue()** преобразует значение этого класса к базовому типу **int.**

Упражнения

1. Создать и заполнить файл случайными целыми числами. Отсортировать содержимое файла по возрастанию и результаты вывести на консоль.
2. Прочитать текст Java-программы и все слова "public" в объявлении атрибутов класса заменить на "private".
3. Прочитать текст Java-программы и записать в другой файл в обратном порядке символы каждой строки.
4. Прочитать текст Java-программы и в каждом слове длиннее двух символов все строчные символы заменить прописными. Результат записать в другой файл.
5. В файле, содержащем фамилии студентов и их оценки, записать большими буквами фамилии тех студентов, которые имеют средний балл более "4".

6. Файл содержит символы, слова, целые числа и числа с плавающей запятой. Прочитать из файла и вывести на консоль все данные, тип которых вводится из командной строки.

7. Из файла удалить все слова, содержащие от трех до пяти символов, но при этом из каждой строки должно быть удалено только максимальное четное количество таких слов.

8. Прочитать текст Java-программы и удалить из него все "лишние" пробелы, оставив только один значащий.

9. Из текста Java-программы удалить все виды комментариев.

10. Прочитать строки из файла и поменять местами первое и последнее слово в каждой строке, записать в тот же файл.

7.ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

Исключения представляют собой классы, объекты которых создаются в случае ошибки при выполнении какой-либо функции. Исключения являются наследниками суперкласса **Throwable** и его подклассов **Error** и **Exception** из пакета **java.lang**. Часто используется подкласс **RuntimeException** класса **Exception** и его наследники: **ArithmeticException, ArrayStoreException, ClassCastException, IllegalArgumentException, IllegalMonitorStateException, IndexOutOfBoundsException, SecurityException.**

Обычно используется один из трех подходов обработки исключений: перехват и обработка исключения; объявление исключений в секции **throws** метода и передача вызывающему методу; перехват исключения, преобразование его к другому классу и повторный вызов.
При вводе/выводе используется класс **IOException**. Например:

```
void doRead(){
  try{
     System.in.read(buffer); }
  catch (IOException e) {
     String err = e.toString(); System.out.println(err);}
}
```

Метод **doRread()** в блоке **try** пытается ввести данные из потока **System.in** в буфер, если возникает ошибка, она обрабатывается в блоке **catch**. Иначе блок **catch** пропускается. Исключение **IOException** возбуждается методом **read()**: **public int read() throws IOException;**
Если блок **try-catch** опустить, компилятор укажет на ошибку.

Если метод может генерировать исключения, которые сам не обрабатывает, он должен объявить о таком поведении с помощью ключевого слова **throws**, чтобы вызывающие методы могли защитить себя от этих исключений. Форма объявления:

тип имя_метода(список аргументов) throws список_исключений { }
При этом сам объявляемый метод может содержать блоки **try-catch**, а может и не содержать их. Например, метод **doRead()** можно объявить:

```
void doRead() throws IOException{
System.in.read(buffer);}
```

Обрабатывать исключение будет метод, вызывающий **doRead()**:

```
void myDoRead(){
  try {
     doRead(); }
  catch (IOException e){
     String err = e.toString();
  System.out.println(err); }
}
```

Метод может обрабатывать несколько исключений.

```
// пример #1 : обработка 2-х типов исключений :TwoException.java
class TwoException {
   public static void main(String[] args) {
 try {
        int a = args.length;
    System.out.println("a = " + a);
      int b = 10 / a;
      int c[] = { 10 };
       c[10] = 11;}
catch(ArithmeticException e){
    System.out.println("деление на 0: "+ e);}
catch(ArrayIndexOutOfBoundsException e) {
     System.out.println("превышение границ массива: " + e); }
  System.out.println("после блока try/catch .");
  }
}
```

Исключение "д*еление на 0*" возникнет при запуске без аргументов командной строки (**a = 0**). При наличии аргумента командной строки вызовется исключение "превышение границ массива".

Подклассы исключений должны следовать перед любым из их суперклассов, иначе суперкласс перехватит исключения. Например:

```
catch(Exception e){/*код*/} //суперкласс Exception перехватит
catch(ArithmeticException e) {/*код*/}//не сработает
```

Операторы **try** можно вкладывать друг в друга. Если у оператора **try** низкого уровня нет раздела **catch**, соответствующего возбужден-

ному исключению, поиск будет развернут на одну ступень выше, и будут проверены разделы **catch** внешнего оператора **try.**

```
// пример #2 : вложенные блоки try-catch : MultiTryCatch.java
public class MultiTryCatch {
  static void method() {
    try {
   int array[] = { 1 };
    array[10] = 5;
} catch(ArrayIndexOutOfBoundsException e) {
 System.out.println("выход за границы массива: " + e);
  }
}
public static void main(String[] args) {
 try {
  int a = args.length();
  System.out.println("a = " + a);
  a -= 999 / a;
 method();
} catch (ArithmeticException e) {
   System.out.println("деление на 0: " + e);
  }
 }
}
```

Оператор throw

Оператор **throw** используется для генерации исключения. Для этого нужно иметь объект подкласса класса **Throwable**. Общая форма:
throw ОбъектThrowable;

При достижении этого оператора выполнение кода прекращается. Ближайший блок **try** проверяется на наличие соответствующего обработчика **catch**. Если такой существует, управление передается ему. Иначе проверяется следующий из вложенных операторов **try**.

Ниже приведен пример, в котором сначала создается объект-исключение, затем оператор **throw** генерирует исключение, обрабатываемое в разделе **catch**, в котором генерируется другое исключение.

```
// пример #3 : генерация исключений : ThrowGeneration.java
public class ThrowGeneration {
  static void throwGen() {
    try {
```

```
   throw new NullPointerException("демонстрация"); }
   catch (NullPointerException e) {
   System.out.println("исключение внутри метода throwGen()");
     throw e;
    }
}
public static void main(String[] args) {
  try { throwGen(); }
  catch(NulPointerException e) {
System.out.println("генерация исключения вне метода: " + e);
    }
  }
}
```

Вызываемый метод **throwGen()** создает объект класса **NullPointerException** и генерирует исключение, перехватываемое обработчиком. Код обработчика сообщает о том, что сгенерировано исключение, а затем снова генерирует его, в результате чего оно передается обработчику исключений в методе **main()**.

Если метод возбуждает исключение в с помощью оператора **throw** и блок **catch** отсутствует, то тип класса исключений должен быть указан в операторе **throws** при объявлении метода. Это означает, что обработка исключения возлагается на метод, вызвавший данный метод.

```
// пример #4 : использование throws : ThrowsSample.java
public class ThrowsSample {
    static void method() throws IllegalAccessException {
      System.out.println(" внутри метода ");
throw new IllegalAccessException("демонстрация исключения");
    }
public static void main(String[] args) {
   try {
 method();
}catch (IllegalAccessException e) {
     System.out.println("перехвачено:  " + e);}
  }
}
```

Ключевое слово finally

Иногда нужно выполнить некоторые действия вне зависимости, произошло исключение или нет. В этом случае используется блок **finally,** который выполняется после блоков **try** или **catch**. Например:

*try{/*код, который может вызвать исключение*/}*
*catch(Exception e){/*обработка исключения*/}*
*finally{/*выполняется или после try, или после catch }*

У каждого раздела **try** должен быть по крайней мере или один раздел **catch,** или блок **finally**. Блок **finally** используется для закрытия файлов и освобождения других ресурсов, захваченных для временного использования в начале выполнения метода. Приведем пример:

```
// пример #5 : выполнение блоков finally: SampleFinally.java
class SampleFinally {
  static void procA() {
try {
 System.out.println("внутри метода procA()");
    throw new RuntimeException("демонстрация исключения"); }
finally { System.out.println("блок finally метода procA()");}
 }
static void procB() {
  try {
     System.out.println("внутри метода procB()");   return;}
finally { System.out.println("блок finally метода procB()"); }
}
public static void main(String[] args) {
   try {
        procA();}
catch (Exception e) {/*обработка */}
  procB();
  }
}
```

В методе **procA()** из-за генерации исключения происходит преждевременный выход из блока **try**, но до выхода из функции выполняется раздел **finally**. Метод **procB()** завершает работу выполнением стоящего в **try**-блоке оператора **return,** но и при этом перед выходом из метода выполняется код блока **finally**.

Можно создать собственное исключение как подкласс класса **Exception** и затем использовать при обработке ситуаций, возникающих после вызова методов. Рассмотрим пример:

```
Book getBook() {
    try {
       Book b1 = object.searchBookRecord("Java 2"); }
catch(NoBookFoundException npfe) { /*обработка */}
     }
```

При неудачном поиске книги вызывается **NoBookFoundException**, который использовался в качестве собственного исключения. Он принимает два аргумента. Один из них – сообщение, которое может быть выведено в поток ошибок; другой – реальное исключение, которое привело к вызову нашего исключения. Этот код показывает, как можно сохранить другую информацию внутри пользовательского исключения. Преимущество этого сохранения состоит в том, что если вызываемый метод захочет узнать реальную причину вызова **NoBookFoundException**, он всего лишь должен вызвать метод **getHiddenException().** Это позволяет вызываемому методу решить, нужно ли работать со специфичным исключением или достаточно обработки **NoBookFoundException.**

```
//пример #6 :собственное исключение: NoBookFoundException.java
public class NoBookFoundException extends Exception {
   private Exception hiddenException_;
public NoBookFoundException(String error, Exception excp){
   super(error);
   hiddenException_ = excp;  }
public Exception getHiddenException(){
   return(hiddenException_); }
}
```

Упражнения

1. java.io.OutputStream out;
public int increment() {
try { counter++;
out.write(counter); }

Какой код нужно добавить в метод increment()?

а) }; б) catch (java.io.IOException e) {; с) finally {; д) return.

2. Что будет выведено при вызове manip(2)?

public int manip(int x) { int count = 3;
try { count += x; m1(x); count++; }
catch(Exception e) { count -= x; }
return count; }

3. Что будет выведено при вызове m1(2), если m2() генерирует ArithmeticException?

public int m1(int x) { int count=1;
try { count += x; count += m2(count); count++;
catch(ArithmeticException e) { count -= x; }
finally { count += 3; }
return count; }

8. КЛАССЫ СОБЫТИЙ

Современный подход к обработке событий основан на модели делегирования событий. В этой модели блок прослушивания (**Listener**) ждет поступления события от источника, после чего обрабатывает его и возвращает управление. Источник – это объект, который генерирует событие, если изменяется внутреннее состояние источника. Когда событие происходит, все зарегистрированные блоки прослушивания уведомляются и принимают копию объекта события. Источник должен вызвать метод, регистрирующий блок прослушивания определенного события. Форма метода:

public void addTunListener(TunListener el)

Здесь **el** – ссылка на блок прослушивания события, объект, получающий уведомление о событии. Таким образом, событие – это объект класса событий, который описывает состояние источника.

В корне иерархии классов событий находится суперкласс **EventObject (java.util)**. Этот класс содержит два метода: **getSource()** – возвращающий источник событий и **toString()** – возвращающий строчный эквивалент события. Подкласс **AWTEvent (java.awt)** является суперклассом всех **AWT-**событий. Метод **getID()** определяет тип событий. Пакет **java.awt.event** содержит ряд классов событий:

ActionEvent – генерируется при нажатии кнопки, двойном щелчке по элементам списка, при выборе пункта меню.

AdjustmentEvent – генерируется при изменении полосы прокрутки.

ComponentEvent – генерируется если компонент скрыт, перемещен, изменен в размере или становится видимым.

FocusEvent – генерируется если компонент получает или теряет фокус ввода.

Класс **InputEvent** является абстрактным суперклассом событий ввода (клавиатуры или мыши). Чтобы рассмотреть события, связанные с обработкой событий клавиатуры, необходимо реализовать интерфейс **KeyListener**. При нажатии клавиши генерируется событие со значением **KEY_PRESSED**. Это приводит к запросу обработчика событий **keyPressed()**. Когда клавиша отпускается, генерируется событие со значением **KEY_RELEASED** и выполняется обработчик **keyReleased()**. Если нажатием клавиши сгенерирован символ, то посылается уведомление о событии со значением **KEY_TYPED** и вызывается обработчик **keyTyped()**.

```
/* пример #1 : обработка нажатия клавиши : JMyKey.java */
import javax.swing.*;
import java.awt.*;
```

```
import java.awt.event.*;
class JMyKey extends JFrame implements KeyListener {
        String keymsg = "";
        public JMyKey(){
                super();
                addKeyListener(this);
        }
        public void keyPressed(KeyEvent ke){}
        public void keyTyped(KeyEvent ke){
                keymsg = ke.getKeyChar() + "";
                repaint();
        }
        public void keyReleased(KeyEvent ke){}
        public void paint(Graphics g){
                g.clearRect(0, 0, 300, 200);
                g.drawString(keymsg, 10, 40);
        }
        public static void main(String[] args){
                JMyKey s = new JMyKey ();
                s.addWindowListener( new WindowAdapter() {
public void windowClosing(WindowEvent e){ System.exit(0);}
         });
                s.setSize(new Dimension(300, 200));
                s.setTitle("обработка нажатия клавиш");
                s.setVisible(true);
        }
}
```

Обработка нажатия специальных клавиш (перемещение курсора, функциональных клавиш) производится в **keyPressed()**. Они не доступны через **keyTyped()**.

Абстрактный класс **WindowAdapter** используется для приема и обработки событий окна при создании объекта прослушивания. Класс содержит перегружаемые методы **windowActivated(WindowEvent)**, вызываемый при активации окна, **windowClosing(WindowEvent)**, вызываемый при закрытии окна и др.

При создании кнопки вызывается конструктор **JButton** со строкой, которую нужно поместить на кнопке. **JButton** – это компонент, который автоматически перерисовывается как часть обновления без явного вызова перерисовки. Просто элемент добавляется на форму, и она автоматически заботится о своей перерисовке. Размещение кнопки на

форме производится внутри метода **init()**. Для добавления к интерфейсу новых элементов управления используется метод **add()** класса **Container.**

```
// пример #2 : добавление кнопки в апплет : MyButton.java
import javax.swing.*;
import java.awt.*;
public class MyButton extends JApplet {
    JButton myB = new JButton("старт");
  public void init() {
      Container c = getContentPane();
      c.setLayout(new FlowLayout());
      c.add(myB);
         }
    }
```

В примере был использован "менеджер компоновки" **FlowLayout**, который является причиной того, что управляющие элементы равномерно плавают на форме, слева направо и сверху вниз. Менеджер компоновки – это способ, которым панель решает, где поместить управляющий элемент на форме. Если запустить приведенный апплет, при нажатии кнопки ничего не произойдет, потому что необходимо написать код для реагирования на это событие. Способ, которым это совершается в компонентах Swing – это интерфейс (графические компоненты) и реализация (код, который запускается при возникновении события). С одним компонентом может быть связано несколько событий. Каждое событие содержит сообщение, которое может быть обработано в разделе реализации.

При использовании компонента **JButton** определяется событие, связанное с нажатием кнопки. Для регистрации заинтересованности блока прослушивания в этом событии вызывается метод **addActionListener()** класса **JButton**. Этот метод ожидает аргумент, являющийся объектом, реализующим интерфейс **ActionListener.** Интерфейс содержит единственный метод **actionPerformed()**, который нужно реализовать. Таким образом для присоединения кода к **JButton**, необходимо реализовать интерфейс **ActionListener** в классе и зарегистрировать объект этого класса в **JButton** через **addActionListener()**. Метод будет вызван при нажатии кнопки.

Для иллюстрации результата нажатия кнопки используем компонент **JTextField**. Это поле, где может размещаться текст или текст может быть изменен. Хотя есть несколько способов создания **JTextField**, самым простым является сообщение конструктору нуж-

ной ширины текстового поля. Как только **JTextField** помещается на форму, можно изменять содержимое, используя метод **setText()**.

```
//пример #3 : ответ на нажатие кнопки : MyButton2.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class MyButton2 extends JApplet implements ActionListener
{  String msg;
   JButton yes, no;
   JTextField txt;
public void init() {
 yes = new JButton ("да");
 no = new JButton ("нет");
 txt = new JTextField(12);
   Container c = getContentPane();
   c.setLayout(new FlowLayout());
          c.add(yes);
          c.add(no);
          c.add(txt);
     yes.addActionListener(this);
     no.addActionListener(this);
     }
    public void actionPerformed(ActionEvent ae) {
      String str = ae.getActionCommand();
    if (str.equals("да")) msg = "нажата кнопка <да> ";
    else msg = " нажата кнопка <нет> ";
          txt.setText(msg);
     }
    }
```

Создание **JTextField** и помещение его на панель − это действия, которые выполняются для любого компонента Swing. Аргумент для **actionPerformed()** представляет собой объект типа **ActionEvent**, содержащий всю информацию о событии. В примере **JButton**.**getText()** возвращает текст, который есть на кнопке, и помещает в **JTextField**. В методе **init()** используется **addActionListener()** для регистрации события обеих кнопок.

Классы-адаптеры

Эти классы содержат пустую реализацию всех методов из интерфейсов прослушивания событий, которые они расширяют. При этом

производится обработка только некоторых из событий, предоставляемых конкретным интерфейсом прослушивания. Определяется новый класс, действующий как блок прослушивания событий, расширяя один из имеющихся адаптеров и реализуя только те события, которые требуется обрабатывать.

Например, класс **MouseMotionAdapter** имеет два метода: **mouseDragged(), mouseMoved()**. Сигнатуры этих пустых методов точно такие же, как в интерфейсе **MouseMotionListener**. Если существует заинтересованность только в событиях перетаскивания мыши, то можно просто расширить адаптер **MouseMotionAdapter** и реализовать метод **mouseDragged()**. Событие же перемещения мыши обрабатывала бы пустая реализация **mouseMoved()**.

Обработка событий Frame-окна

Когда происходит событие, связанное с окном, вызываются обработчики событий, определенные для этого окна. При создании оконного приложения используется метод **main()**, создающий для него окно верхнего уровня. После этого программа будет функционировать как GUI-приложение, а не консольная программа. Программа поддерживается в работоспособном состоянии, пока не закрыто окно.

В общем смысле окно является контейнером. Графический контекст инкапсулирован в классе и получается двумя способами: вызовом методов **paint(), update()**; возвращается методом **getGraphics()** класса **Component**.

Событие **FocusEvent** предупреждает программу, что компонент получил или потерял фокус ввода. Класс **InputEvent** является суперклассом для классов **KeyEvent** и **MouseEvent.** Событие **WindowEvent** извещает программу, что пользователь воспользовался одним из системных элементов управления окна.

Для создания графического интерфейса потребуется предоставить место (окно), в котором он будет отображаться. При выполнении апплета за эти действия отвечает браузер. Если программа является приложением, подобные действия она должна выполнять самостоятельно. Место, на котором рисуется пользовательский интерфейс, называют контейнером. Элементы интерфейса пользователя называют компонентами. Для создания графического интерфейса приложения необходимо предоставить ему объект **Frame**, в который будут помещаться используемые приложением компоненты GUI.

В следующем примере работа приложения начинается с метода **main()**, в функции которого входит создание объекта **MyMouseWithFrame** и передача ему управления.

```
//пример #4 : применение адаптеров : MyMouseWithFrame.java
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class MyMouseWithFrame extends Frame implements
ActionListener{
        private Button button = new Button("кнопка");
        String msg = "";
  MyMouseWithFrame() {
        addMouseListener(new MyMouseAdapter(this));
        setLayout(null);
        setBackground(new Color(255, 0, 0));
        setForeground(new Color(0, 0, 255));
        button.setBounds(100, 100, 50, 20);
        button.addActionListener(this);
        add(button);
      }
public static void main(String[] args) {
        MyMouseWithFrame myf = new MyMouseWithFrame();
  myf.addWindowListener(new WindowAdapter() {
                        public void windowClosing(WindowEvent e) {
                                System.exit(0);
                        };
                });
        myf.setSize(new Dimension(250, 200));
        myf.setTitle("Frame - окно");
        myf.setVisible(true);
        }
public void paint(Graphics g) {
        g.drawString(msg, 80, 50);
}
public void actionPerformed(ActionEvent e) {
        msg = "кнопка нажата";
        repaint();
    }
}
class MyMouseAdapter extends MouseAdapter {
```

```
        public MyMouseWithFrame mym;
public MyMouseAdapter(MyMouseWithFrame mym) {
        this.mym = mym;
        }
public void mousePressed(MouseEvent me) {
        mym.msg = "клавиша мыши нажата";
        mym.repaint();
        }
public void mouseReleased(MouseEvent me){
mym.msg = " клавиша мыши отпущена";
        mym.repaint();}
}
```

Конструктор класса **MyMouseWithFrame** использует метод **addMouseListener(new MyMouseAdapter(this))** для регистрации событий "мыши". При создании объекта класса **MyMouseWithFrame** этот метод сообщает объекту, что он заинтересован в обработке определенных событий. Однако, вместо того, чтобы известить его об этом прямо, конструктор организует посылку ему предупреждения через объект класса **MyMouseAdapter** (только один тип событий). Абстрактный класс **MouseAdapter** используется для обработки событий, связанных с "мышью" при создании блока прослушивания и содержит перегружаемые методы **mousePressed(MouseEvent), mouseReleased(MouseEvent).**

Класс **MyMouseWithFrame** также обрабатывает событие класса **WindowEvent**. Когда объект генерирует событие **WindowEvent**, объект **MyMouseWithFrame** анализирует, является ли оно событием **WindowClosing.** Если это не так, объект **MyMouseWithFrame** игнорирует его. Если получено ожидаемое событие, в программе запускается процесс завершения ее работы.

Упражнения

1. Какое событие генерируется при попытке закрытия Frame-окна?
2. Как оргагнизовать блок прослушивания с использованием адаптера?

9. ГРАФИЧЕСКИЕ ИНТЕРФЕЙСЫ ПОЛЬЗОВАТЕЛЯ

Основы оконной графики

Пакет **AWT** (**Abstract Windowing Toolkit)** содержит набор классов, позволяющих выполнять графические операции, создавать такие элементы управления как кнопки, меню, окна и др. Суперкласс **Component** является абстрактным классом, инкапсулирующим все ат-

рибуты визуального компонента. Порожденный от него подкласс **Container** – содержит методы, которые позволяют вкладывать в него другие компоненты (объекты) и отвечает за размещение любых компонентов, которые он содержит. Этот класс пораждает подклассы **Panel** и **Window**. Класс **Panel** используется апплетами. Когда экранный вывод направляется к апплету, он рисуется на поверхности объекта **Panel,** окна, которое не содержит области заголовка, строки меню и обрамления. Класс **Window** (окно верхнего уровня) непосредственно не используется, а используется его подкласс **Frame**, который создает стандартное окно, имеющее строку заголовка, меню, изменяющиеся размеры. Класс **Canvas** не является подклассом **Contanier**, это пустое окно, в котором можно рисовать.

Большинство графических операций – это методы класса **Graphics**. Функции получают объект класса **Graphics** (графический контекст) в качестве параметра и вместе с ним получают текущий цвет, шрифт, положение курсора. Установку контекста обычно осуществляют методы **update()** или **paint().** Рассмотрим несколько методов этого класса:

drawLine (int x1, int y1, int x2, int y2) – рисует линию;

drawRect(int x,int y,int width,int height) и **fillRect (int x, int y, int width, int height)** – рисуют прямоугольник и заполненный прямоугольник;

draw3DRect(int x, int y, int width, int height, boolean raised) – рисует трехмерный прямоугольник;

drawRoundRect(**int x, int y, int width, int height, int arcWidth, int arcHeight)** – рисует округленный прямоугольник;

drawOval(int x, int y, int width, int height) – рисует овал;

drawPolygon (int[] xPoints, int[] yPoints, int nPoints) – рисует полигон, заданный массивами координат x и y;

drawPoiygon(Polygon p) – полигон, заданный объектом **Polygon;**

drawPolyline(int[] xPoints, int[] yPoints, int nPoints) – рисует последовательность связных линий, заданных массивами x и y;

drawArc(int x, int y, int width, int height, int startAngle, int arcAngle) – рисует дугу окружности;

drawImage(Image img, int x, int y, Color bgcolor, ImageObserver observer) – вставляет изображение;

drawString(String str, int x, int y) – рисует строку;

setColor(**Color c)**, **getColor()** – устанавливает и возращает текущий цвет;

getFont() – возвращает текущий шрифт;

setFont(Font font) – устанавливает шрифт в указанный.

```
//пример #1 : вывод текста, линии и овалов : WindowApp.java
import java.awt.*;
import java.awt.event.*;
public class WindowApp extends Frame{
  String msg = "Window-Application";
  int x1 = 30, y1 = 50, x2 = 200, y2 = 50;
 public WindowApp(){
addWindowListener(new MyWindowAdapter());}//прослушивание
   public void paint(Graphics g){
g.drawString(msg, 30, 40); //вывод строки с позиции x = 30 y = 40
g.drawLine(x1, y1, x2, y2);//вывод линии
       int x = 30, y = 200, width = 150, height = 100;
Color c = new Color(255, 100, 100);//установка красного цвета
g.setColor(c);
g.drawOval( x, y, width, height);//овал
g.drawArc (x + 100, y + 50, width - 50,  height, 0, 360);//сектор
}
public static void main(String args[]){
   WindowApp fr = new WindowApp();
fr.setSize(new Dimension(500, 400)); //размер окна
fr.setTitle("awt-Application");
fr.setVisible(true); //видимость
fr.repaint();} //перерисовка, вызов paint()
 }
class MyWindowAdapter extends WindowAdapter{
   public void windowClosing(WindowEvent e){
System.exit(0);}}//закрыть приложение
```

Графические приложения в настоящее время используются довольно редко. В большинстве случаев графика применяется в апплетах. Апплеты представляют классы языка Java, которые размещаются на серверах Internet, транспортируются клиенту по сети, автоматически устанавливаются и запускаются на месте как часть документа WWW. Апплеты позволяют вставлять в документы поля, содержание которых меняется во времени, организовывать "бегущие строки", мультипликацию. Апплеты являются наследниками суперкласса **Applet** и его подкласса **JApplet**. Есть несколько методов, которые управляют созданием и выполнением апплета на Web-странице. Перегружаемый метод **init()** автоматически вызывается для выполнения

начальной инициализации апплета. Метод **start()** вызывается каждый раз, когда апплет переносится в поле зрения Web-браузера, чтобы начать операции. Метод **stop()** вызывается каждый раз, когда апплет выходит из поля зрения Web-браузера, чтобы позволить апплету завершить операции. Метод d**estroy()** вызывается, когда апплет начинает выгружаться со страницы для выполнения финального освобождения ресурсов. Кроме этих методов автоматически загружаемым является метод **paint()** класса **Component**. Метод **paint()** не вызывается явно, а только из других методов, например **repaint()**.

Рассмотрим пример апплета, в котором используются методы **init(), paint(),** метод **setColor()** установки цвета символов, метод **drawString()** рисования строк**.**

```
//пример #2 : вывод строк разными цветами: ColorConstants.java
import java.applet.*;
import java.awt.*;
public class ColorConstants extends Applet {
    public void init(){ }
  public void paint(Graphics g)    {
     g.setColor(Color.yellow);
     g.drawString("<желтый>", 5, 30);// желтым цветом
     g.setColor(Color.blue);
     g.drawString("<синий>", 5, 60);//синим цветом
     g.setColor(Color.green);
     g.drawString("<зеленый>", 5, 90);//зеленым цветом
   }
}
```

Апплету не нужен метод **main(),** код запуска помещается в **init()** и **paint()**. После компиляции имя класса, содержащего байт-код апплета помещается в тег **<applet> ..</applet>** документа HTML. Например:

```
<html>
<applet code=ColorConstants.class  width=500 height=500></applet>
</html>
```

Исполнителем HTML-документа является браузер или AppletViewer.

```
//пример #3 : передача параметров апплету : ReadParam.java
import java.awt.*;
import java.applet.*;
public class ReadParam extends Applet{
double a;  int b;  String name;  boolean statement;
public void start() {//чтение параметров
        String param;
```

```
        name = getParameter("name");
        if(name == null)
          name = "No Name";
        param = getParameter("statement");
        if(param != null)
          statement = Boolean.valueOf(param).booleanValue();
        try {
                param = getParameter("aNumber");
         if(param != null) // если не найден
                  b = Integer.parseInt(param);
          else  b = 0;
                param = getParameter("bNumber");
         if(param != null) // если не найден
                  a = Double.valueOf(param).doubleValue();
          else  a = 1;
        } catch(NumberFormatException e) {
          a = 0; b = 0; statement = false;       }
}
public void paint(Graphics g) {
double d = 0;
        if(statement)  d = Math.pow(a,b);
          else name = "ERROR !";
        g.drawString("Name: " + name, 0, 11);
        g.drawString("Value a: " + a, 0, 28);
        g.drawString("Value b: " + b, 0, 45);
        g.drawString("a power b: " + d, 0, 62);
        g.drawString("Statement: " + statement, 0, 79);
  }
}
```

Если параметр недоступен, **getParameter()** возвращает **null**. HTML-документ для этого примера может иметь вид:

```
<html><head><title>Параметры апплета</title></head><body>
<applet code = ReadParam.class width = 250 height = 300>
<param name = name value ="Value of 'a' raised to the power of 'b'">
<param name = aNumber value = 10>
<param name = bNumber value = 2>
<param name = statement value = true>
</applet></body> </html>
```

В примерах, приведенных ниже, демонстрируется использование методов класса **Graphics** для вывода графических изображений в окно апплета.

```
//пример #4 : отображение полигона : DrawPoly.java
import java.applet.*;
import java.awt.*;
public class DrawPoly extends Applet {
  int poly1_x[] = {  40,  80,   0,  40};
  int poly1_y[] = {   5,  45,  45,   5};
  int poly2_x[] = { 140, 180, 180, 140, 100, 100, 140};
  int poly2_y[] = {   5,  25,  45,  65,  45,  25,   5};
  int poly3_x[] = { 240, 260, 220, 260, 220, 240};
  int poly3_y[] = {   5,  65,  85,  25,  25,   5};
  public void paint(Graphics g)     {
      g.drawPolygon(poly1_x, poly1_y, poly1_x.length);
      g.drawPolygon(poly2_x, poly2_y, poly2_x.length);
      g.drawPolygon(poly3_x, poly3_y, poly3_x.length);
    }
 }
//пример #5 : трехмерный прямоугольник : ThreeDRect.java
import java.applet.*;
import java.awt.*;
public class ThreeDRect extends Applet {
  public void draw3DRect(Graphics g, int x, int y,
    int width, int height, boolean raised)     {
     g.draw3DRect(x, y, width - 1, height - 1, raised);
     g.draw3DRect(x + 1, y + 1, width - 3, height - 3, raised);
     g.draw3DRect(x + 2, y + 2, width - 5, height - 5, raised);
    }
  public void fill3DRect(Graphics g, int x, int y,
    int width, int height, boolean raised)
   { g.draw3DRect(x, y, width-1, height-1, raised);
     g.draw3DRect(x + 1, y + 1, width - 3, height - 3, raised);
     g.draw3DRect(x + 2, y + 2, width - 5, height - 5, raised);
     g.fillRect(x + 3, y + 3, width - 6, height - 6);
    }
  public void paint(Graphics g)     {
     g.setColor(Color.gray);
     draw3DRect(g, 10, 5, 80, 40, true);
     draw3DRect(g, 130, 5, 80, 40, false);
```

```
      fill3DRect(g, 10, 55, 80, 40, true);
      fill3DRect(g, 130, 55, 80, 40, false);
    } }
//пример #6 : вывод в окно GIF-изображения : DrawImage.java
import java.applet.*;
import java.awt.*;
public class DrawImage extends Applet {
  Image img;
  public void init() {
     img = getImage(getCodeBase(), "cow.gif");  }
  public void paint(Graphics g){ g.drawImage(img, 0, 0, this);}
}
//пример #7 : создание компонент AWT : AWTTest.java
import java.applet.*;
import java.awt.*;
public class AWTTest extends Applet {
 Label label = new Label("объект Label"); //метка
 Button button_a = new Button("кнопка"); //кнопка
 Checkbox option_a = new Checkbox("яблоко"); //флажок
      public void init() { //добавление
          add(label);
          add(button_a);
          add(option_a);}
 }
```

В настоящее время при использовании управляющих компонентов в апплетах принято использовать интерфейсные классы, в которых компонент на экране создается средствами Java и в минимальной степени зависит от платформы и оборудования. Такого рода классы компонент были объединены в библиотеку под названием Swing. Пакет **javax.swing** содержит класс **JApplet** подкласс суперкласса **Applet.**

```
//  пример #8 : апплет с компонентом : MyJApplet.java
import javax.swing.*;
import java.awt.*;
public class MyJApplet extends JApplet {
      Jlabel lbl;
  public void init() {
    Container c =  getContentPane();
        lbl = new JLabel("Swing-апплет!"));
           c.add(lbl);
```

```
  }
}
```

В этой программе производится помещение текстовой метки на форму в апплете с помощью класса **JLabel** (в AWT есть соответствующий класс **Label**, лидирующая "**J**" используется также и для других компонентов Swing). Конструктор класса **Jlabel** принимает **String** и использует для создания метки. Метод **init()** отвечает за помещение всех компонентов на форму, используя метод **add()**. Метод **add()** сразу не вызывается, как в библиотеке AWT. Пакет **Swing** требует, чтобы все компоненты добавлялись в "панель содержания" формы, так что требуется вызывать **getContentPane()**, как часть процесса **add()**.

```
// пример #9 : работа методов init(), start(), stop() : InitDemo.java
import java.applet.*;
import java.awt.*;
public class InitDemo extends JApplet {
 private int i ;
 private String msg = null;
 private Color c;
public void init() {
  c = new Color(0,0,255);
     this.setBackground(c);
     this.setForeground(Color.white);
  setFont(new java.awt.Font("Courier", 1, 14));
  msg = "инициализация";
  i = 1;
    }
public void start() {
 int j = i * 25;
 if (j < 255){ c = new Color (j, j, 255 - j); }
 setBackground(c);
 String str = Integer.toString(i);
 msg += " " + str;
 }
public void paint(Graphics g) { g.drawString(msg, 30, 30);}
public void stop() {   i++;   msg = "работа start() -  stop()"; }
}
```

При работе со шрифтами можно узнать какие из них доступны на компьютере и использовать их. Для получения этой информации применяется метод **getAvailableFontFamilyNames()** класса

GraphicsEnvironment. Метод возвращает массив строк, содержащий имена доступных семейств шрифтов.

```
//пример #10 : демонстрация различных шрифтов : FontDemo.java
import javax.swing.*;
import java.awt.*;
public class FontDemo extends JApplet {
 private int i ;
 private String msg = null;
 private String[] fonts;
public void init() {
 GraphicsEnvironment ge =
GraphicsEnvironment.getLocalGraphicsEnvironment();
fonts = ge.getAvailableFontFamilyNames();
   }
public void start() {
 int j = i;
   if (j > fonts.length) i = 0;
   else setFont(new Font(fonts[j], 1, 14));
      String str = Integer.toString(i);
      msg = fonts[j]+ " " + str;
      i++;
 }
public void paint(Graphics g) { g.drawString(msg, 30, 30); }
}
```

Библиотека Swing поддерживает набор графических методов. Вся графика рисуется относительно окна. Вывод в окно выполняется через графический контекст: **paint()** или **update().** Класс **Graphics**, объект которого передается в контекст, определяет функции рисования.

В следующем апплете проверяется принадлежность точки нажатия "мыши" прямоугольнику.

```
//  пример #11 : рисование прямоугольника  : MyRect.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class MyRect extends JApplet implements MouseListener {
  Rectangle a = new Rectangle(20, 20, 100, 60);
public void init() {
 setBackground(Color.yellow);
 addMouseListener(this);
}
```

```
public void mouseClicked(MouseEvent me) {
   int x = me.getX();
   int y = me.getY();
 if (a.inside(x,y))  System.out.println("клик в зеленом Rect");
    else   System.out.println("клик в желтом");
}
public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}
public void mousePressed(MouseEvent e) {}
public void mouseReleased(MouseEvent e) {}
public void paint(Graphics g) {
    g.setColor(Color.green);
    g.fillRect(a.x, a.y, a.width, a.height);
  }
}
```

Упражнения

1. <applet code=MyApplet.class width=200 height=200>
<param name=count value=5>
</applet>
Какой код метода init() читает параметр count в переменную i?
а)int i = new Integer(getParameter("count")).intValue();
б)int i = getIntParameter("count");
с)int i = getParameter("count");
д)int i = new Integer(getIntParameter("count"));
е)int i = new Integer(getParameter("count"));

2. Вы нажали кнопку изменения цвета основания (background) апплета. Какой метод должен быть вызван, чтобы это сделать?
а)setbgcolor(); б)draw(); в)start(); г)repaint().

10. ЭЛЕМЕНТЫ УПРАВЛЕНИЯ

Развитие языка Java идет в направлении перехода к так называемым "легким" компонентам пользовательского интерфейса. В ранних (1.0.x) версиях JDK использовались "тяжелые" компоненты AWT, связанные с аппаратными платформами. Дальнейшее развертывание концепции "write once, run everywhere" (написать однажды, запускать везде) привело к тому, что в версии 1.1.x наметился переход к таким компонентам, которые бы не были завязаны на конкретные "железо" и операционные системы. Такого рода классы компонентов были объединены в библиотеку под названием Swing и доступны разработчикам в составе как JDK, так и отдельного продукта JFC (Java Foundation Classes). Причем для совместимости со старыми версиями JDK старые

компоненты из AWT остались нетронутыми, хотя компания JavaSoft, отвечающая за выпуск JDK, – рекомендует не смешивать в одной и той же программе старые и новые компоненты.

Для работы с окнами используются классы **JFrame** и **JPanel** из библиотеки Swing вместо **Frame** и **Panel** из стандартного набора классов AWT. Панели класса **JPanel** должны добавляться в окна класса **JFrame** исключительно методом **setContentPane()**, а не **add()**, как это делается в AWT.

От **AbstractButton** наследуются два основных кнопочных класса: **JButton** и **JToggleButton**. Первый служит для создания обычных кнопок с разнообразными возможностями, а второй – для создания радиокнопок (класс **JRadioButton**) и отмечаемых кнопок (класс **JCheckBox**). Помимо названных, от **AbstractButton** наследуется пара классов **JCheckBoxMenuItem** и **JRadioButtonMenuItem**, используемых для организации тех меню, пункты которых должны быть оснащены отмечаемыми и радиокнопками.

Процесс создания кнопок достаточно прост: вызывается конструктор **JButton** с меткой, которую нужно поместить на кнопке. Класс **JButton** библиотеки Swing для создания обычных кнопок предлагает несколько различных конструкторов: **JButton(), JButton(String), JButton(Icon), JButton(String, Icon).**

Если используется конструктор без параметров, то получится абсолютно пустая кнопка. Задав текстовую строку, получим кнопку с надписью. Для создания кнопки с рисунком конструктору передается ссылка на класс пиктограммы. Класс **JButton** содержит несколько десятков методов. **JButton** – это компонент, который автоматически перерисовывается как часть обновления. Это означает, что не нужно явно вызывать перерисовку кнопки как и любого управляющего элемента; он просто помещается на форму и сам автоматически заботится о своей перерисовке. Чтобы поместить кнопку на форму, необходимо выполнить это внутри **init().** Перед помещением любого элемента на форму создается новый "менеджер компоновки" типа **FlowLayout**. Менеджер компоновки – это способ, которым панель решает, где поместить управляющий элемент на форме. Каждый раз, когда кнопка нажимается, генерируется action-событие. Оно посылается блокам прослушивания, зарегистрированным для приема события от этого компонента.

```
// пример #1 : создание кнопки : MyButtons.java
import java.awt.*;
import javax.swing.*;
```

```
import java.awt.event.*;
public class MyButtons extends JApplet implements ActionListener {
            String msg;
            JButton yes, maybe;
            JLabel lbl;
public void init() {
  yes = new JButton ("да");
  maybe = new JButton ("возможно");
  lbl = new JLabel("");
            Container c = getContentPane();
            c.setLayout(new FlowLayout());
c.add(yes);
c.add(maybe);
c.add(lbl);
    yes.addActionListener(this);
    maybe.addActionListener(this);
   }
public void actionPerformed(ActionEvent ae) {
  String str = ae.getActionCommand();
      if (str.equals("да")) msg = "нажата кнопка <да>";
      else  msg = "нажата кнопка <возможно>";
            lbl.setText(msg);
      }
}
```

В этом примере метка кнопки используется для того, чтобы определить, какая из них была нажата. Метка возвращается вызовом метода **getActionCommand()** объекта **ActionEvent**, передаваемого методу **actionPerformed().** Метод **setText()** объекта **JLabel** используется для передачи строки в объект **lbl.**

```
// пример #2 : вывод в текстовое поле : MyTextField.java
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class MyTextField extends JApplet implements ActionListener
{ String msg;
  JButton yes, maybe;
  JTextField txt;
public void init() {
    yes = new JButton ("да");
    maybe = new JButton ("возможно");
```

```
    txt = new JTextField(10);
   Container c = getContentPane();
   c .setLayout(new FlowLayout());
c.add(yes);
c.add(maybe);
c.add(txt);
      yes.addActionListener(this);
  maybe.addActionListener(this);
}
public void actionPerformed(ActionEvent ae) {
   String name = ((JButton)ae.getSource()).getText();
    txt.setText(name);
    }
}
```

Создание **JTextField** и помещение его на канву – это шаги, необходимые и для **JButton** или любого компонента Swing. Аргумент для **actionPerformed()** имеет тип **ActionEvent**, который содержит всю информацию о событии и откуда оно исходит. В этом случае описывается кнопка, которая была нажата: **getSource()** производит объект, явившийся источником события, и **JButton**. **getText()** возвращает текст, который есть на кнопке, и помещается в **JTextField** для демонстрации, что код действительно был вызван при нажатии кнопки.

Класс **JComboBox** используется для создания раскрывающегося списка элементов, из которых пользователем производится выбор. Таким образом, данный элемент управления имеет форму меню. В неактивном состоянии компонент типа **JComboBox** занимает столько места, чтобы показывать только текущий выбранный элемент. Для определения выбранного элемента можно вызвать метод **getSelectedItem()** или **getSelectedIndex().**

```
// пример #3 : выпадающий список : MyComboBox.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class MyComboBox extends JApplet implements ItemListener{
   JLabel lbl;
   JComboBox cb;
public void init(){
   lbl = new JLabel("0");
   cb = new JComboBox();
  Container c = getContentPane();
```

```
  c.setLayout(new FlowLayout());
cb.addItem("11"); cb.addItem("22");
cb.addItem("33"); cb.addItem("44");
cb.addItemListener(this);
  c.add(cb);
  c.add(lbl);
}
public void itemStateChanged(ItemEvent ie) {
  int arg = Integer.valueOf((String)ie.getItem()).intValue();
  double res = Math.pow(arg, 2); //возведение в квадрат
    lbl.setText(Double.toString(res));
  }
}
```

При выборе элемента списка генерируется событие **ItemEvent** и посылается всем блокам прослушивания, зарегистрированным для приема уведомлений о событиях данного компонента. Каждый блок прослушивания реализует интерфейс **ItemListener.** Этот интерфейс определяет метод **itemStateChanged().** Объект **ItemEvent** передается этому методу в качестве аргумента. Приведенная программа позволяет выбрать из списка число, возводит его в квадрат и выводит его в объект **JLabel.** Здесь также следует обратить внимание на способы преобразования строковой информации в числовую и обратно.

В следующем примере рассмотрено отслеживание изменения состояния объекта **JCheckBox.**

```
/* пример #4 : отслеживание изменения состояния флажка :
MyCheckBox.java */
import java.applet.*;
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class MyCheckBox extends JApplet implements ItemListener{
   CheckBox cb;
   JLabel lbl;
public void init() {
  cb = new JCheckBox();
   lbl = new JLabel("инициализация");
Container c = getContentPane();
    c.setLayout(new FlowLayout());
cb.addItemListener(this);
    c.add(cb);
```

```
   c.add(lbl);
}
public void itemStateChanged(ItemEvent ie) {
  String msg;
if(ie.getStateChange() == ItemEvent.SELECTED) msg = "True";
 else msg = "False";
        lbl.setText(msg);
  }
}
```

Здесь приведены метод **getStateChange(),** извлекающий из объекта **ItemEvent** константу состояния, в данном случае **SELECTED**, прослушиваемого объекта **JCheckBox.**

Если требуются радиокнопки для получения поведения, вида "исключающее или", необходимо добавить их в "группу кнопок". Но, как показывает приведенный ниже пример, любая **AbstractButton** может быть добавлена в **ButtonGroup**.

Для предотвращения повтора большого количества кода этот пример использует рефлексию для генерации различных типов кнопок. Это происходит в **makeMyPanel()**, которая создает группу кнопок и **JPanel**. Второй аргумент для **makeMyPanel()** – это массив **String**. Для каждого **String**, в **JPanel** добавляется кнопка класса, соответствующего первому аргументу:

```
// пример #5 : различные типы кнопок и флажков: DiffButtons.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;
import java.lang.reflect.*;
public class DiffButtons extends JApplet {
    static String[] str = {"первый","второй","все", };
public void init() {
  Container c = getContentPane();
  c.setLayout(new FlowLayout());
  c.add(makeMyPanel(JButton.class, str));
  c.add(makeMyPanel(JToggleButton.class, str));
  c.add(makeMyPanel(JCheckBox.class, str));
  c.add(makeMyPanel(JRadioButton.class, str));
}
static JPanel makeMyPanel(Class aCl, String[] str) {
  ButtonGroup b = new ButtonGroup();
```

```
  JPanel p = new JPanel();
  String title = aCl.getName();
  title = title.substring(title.lastIndexOf('.') + 1);
       p.setBorder(new TitledBorder(title));
for(int j = 0; j < str.length; j++) {
   AbstractButton abs = new JButton("ошибка");
try {
  Constructor cons = aCl.getConstructor(
new Class[] { String.class
 });
  abs = (AbstractButton)cons.newInstance(
  new Object[]{str[j]});
} catch(Exception e) {System.err.println("нельзя создать: "+aCl); }
     b.add(abs);
     p.add(abs);
    }
return p;
  }
}
```

Заголовок для бордюра берется из имени класса, от которого отсекается вся информация о пути. **AbstractButton** инициализируется с помощью **JButton**, которая имеет метку "ошибка", так что если игнорируется сообщение исключения, то проблема видна на экране. Метод **getConstructor()** производит объект **Constructor**, который принимает массив аргументов типов в массиве **Class**, переданном **getConstructor()**. Затем все, что нужно сделать − это вызвать **newInstance()**, передав этот массив элементов **Object**, содержащий реальные аргументы (в этом случае просто **String** из массива **str**). Здесь немного усложнен простой процесс. Для получения поведения кнопок, вида "исключающее или" создается группа кнопок и добавляется каждая кнопка, для которой определено поведение в группе. После запуска программы видно, что все кнопки, за исключением **JButton**, показывают поведение, вида "исключающее или".

Упражнения

1. Создать форму с несколькими кнопками так, чтобы надпись на первой кнопке при ее нажатии передавалась на следующую, и т.д.
2. Создать выпадающий список так, чтобы при выборе элемента списка на экране появлялись GIF-изображения, двигающиеся по апплету.
3. По экрану движутся одна за одной строки из массива строк. Направление движения и значение каждой строки выбирается случайным образом.

4. Строка составляется из символов появляющихся из разных углов апплета и выстраювающихся друг за другом. Процесс циклически повторяется.

5. По апплету движется окружность. При касании границы она отражается от нее с эффектом упругого сжатия.

6. Направление движения прямоугольника по экрану изменяется щелчком по клавише мыши случайным образом. При этом каждый второй щелчок меняет цвет фона.

7. Длина дуги окружности изменяется нажатием клавиш от 1 до 9.

8. Клавиши “вверх”, “вниз”, “вправо”, “влево” двигают в соответствующем направлении линию. При достижении границ апплета линия появляется с противоположной стороны.

9. Окружность “убегает” от указателя мыши. При приближении на некоторое расстояние окружность появляется в другом месте апплета.

10. Квадрат на экране движется к указателю мыши, когда последний находится в границах апплета.

11. ПОТОКИ И МНОГОПОТОЧНОСТЬ

К большинству современных Web-приложений выдвигаются требования одновременной поддержки многих пользователей и разделения информационных ресурсов. Потоки – средство, которое поможет организовать одновременное выполнение нескольких задач с помощью многопоточности – использования нескольких потоков управления в одной программе. Например, метод **repaint()** создает поток, обновляющий экран, в то время как программа выполняется. Способ добавить анимацию в апплет – использование потока. Этот поток можно запускать, когда апплет становится видимым, и останавливать, когда он невидим. Поток создаёт анимационный эффект повторением вызова метода **paint()** и отображением вывода в новой позиции. Существуют два способа запуска класса в потоке: расширение класса **Thread** и реализация интерфейса **Runnable.**

```
// пример #1 : расширение класса Thread : Talk.java
class Talk extends Thread {
  public void run() {
   for (int i = 0; i < 8; i++) { System.out.println("Talking");
 try {Thread.sleep(400);//остановка на 400 миллисекунд   }
 catch (InterruptedException e) {
System.out.println("метод sleep() не работает - "+e);  }
            }
      }
}
```

При реализации интерфейса **Runnable** необходимо определить абстрактный метод **run(),** который содержится в интерфейсе. Например:

```
// пример #2 : реализация интерфейса Runnable : TalkWalk.java
class Walk implements Runnable {
   public void run() {
      for (int  i = 0; i < 8; i++) {  System.out.println("Walking");
     try {  Thread.sleep(300); } catch (InterruptedException e) {
System.out.println(“ метод sleep() не работает - ”+e);}
      }
   }
}
public class TalkWalk extends JApplet {
   Talk talk  = new Talk();   //новый поток
   Thread walk = new Thread(new Walk());//новый поток
   public void init() {       talk.start();       walk.start();    }
}
```

Использование двух потоков для объектов классов **Talk** и **Walk** приводит к поочередному выводу строк: *Talking Walking.*

При запуске программы объект класса **Thread** может быть в одном из четырёх состояний: "новый", "работоспособный", "неработоспособный" и "пассивный". При создании потока он получает состояние "новый" и не выполняется. Для перевода потока из состояния "новый" в состояние "работоспособный" следует выполнить метод **start(),** который вызывает метод **run()** – основное тело потока. Интерфейс **Runnable** не имеет метода **start(),** а только единственный метод **run().** Поэтому для запуска такого потока как **Walk** следует создать объект класса **Thread** и передать поток **Walk** его конструктору:

```
Thread walk = new Thread(new Walk());
```

Поток переходит в состояние "неработоспособный" вызовом методов **suspend(), sleep(), wait()** или методов ввода/вывода, которые предполагают задержку. Поток переходит в "пассивное" состояние, если вызван метод **stop()** или метод **run()** завершил выполнение. После этого, чтобы выполнить поток еще раз, необходимо создать новую копию потока. Для задержки потока на некоторое время (в миллисекундах) можно перевести его в режим ожидания с помощью метода **sleep()**. Метод **sleep()** генерирует прерывание **InterruptedException**, которое программа может перехватить. Рассмотрим пример:

```
// пример #3 : остановка потока : SuspendResume.java
import javax.swing.*;
import java.awt.*;
class MyThread extends Thread {
  public void run(){
```

```
    while (true) {
       try { sleep(400); } catch (InterruptedException e) {}
       System.out.println("Thread работает!");
      }
   }
}
public class SuspendResume extends JApplet {
  MyThread mythread = new MyThread ();
  public void init(){mythread.start();}
  public boolean mouseDown(Event evt, int x, int y) {
mythread.suspend();
     System.out.println("Thread остановлен");
     return(true);
   }
  public boolean mouseUp(Event evt, int x, int y) {
System.out.println("Thread возобновлен");
      mythread.resume();
      return(true);
   }
}
```

Метод **suspend()** позволяет приостановить выполнение потока, пока не возникло какое-либо событие, например, пока не нажата кнопка. Выполнение возобновляется вызовом метода **resume().**

Потоку можно назначить приоритет от 0 (константа **MIN_ PRIORITY**) до 10 **(MAX_PRIORITY)** с помощью метода **setPriority(),** получить приоритет можно с помощью метода **getPriority().**

```
// пример #4 : установка приоритета : ThreadPriority.java
import javax.swing.*;
class MyThread extends Thread {
myThread(String name)
   {  super(name);  }
  public void run(){
     for (int i = 0; i < 8; i++)
       {  System.out.println(getName() + " " + i);
        try { sleep(10); }catch (InterruptedException e) {}
       }
   }
}
public class ThreadPriority extends JApplet {
  MyThread min_thread = new MyThread ("Thread Min");
```

```
  MyThread max_thread = new MyThread ("Thread Max");
  MyThread norm_thread = new MyThread ("Thread Norm");
 public void init(){
      min_thread.setPriority(Thread.MIN_PRIORITY);
      max_thread.setPriority(Thread.MAX_PRIORITY);
      norm_thread.setPriority(Thread.NORM_PRIORITY);
        min_thread.start(); max_thread.start(); norm_thread.start();
    }
}
```

Поток с более высоким приоритетом может монополизировать вывод на консоль. Приостановить выполнение потока можно с помощью метода **sleep()** класса **Thread**. Альтернативный способ состоит в вызове метода **yield(),** который делает паузу и позволяет другим потокам выполнить свою задачу.

```
// пример #5 : запуск и остановка потоков : InfiniteThread.java
import java.applet.Applet;
public class InfiniteThread extends Applet implements Runnable{
Thread myThread;
  public void init() {
  System.out.println("в методе init() – старт thread");
     myThread = new Thread(this);
     myThread.start();
    }
  public void start(){
 System.out.println("в методе start() – продолжение thread");
     myThread.resume();
    }
  public void stop() {
  System.out.println("в методе stop() – приостановка thread");
     myThread.suspend();
    }
  public void destroy() {
System.out.println("в методе destroy() – уничтожение thread");
     myThread.resume();
     myThread.stop();
    }
  public boolean mouseDown(java.awt.Event e, int x, int y){
    System.out.println("число потоков  " + myThread.activeCount());
    return true;
    }
```

```
 public void run()    {
  int i = 0;
     for( ; ; ) { i++;
          System.out.println("В потоке " + i + " раз");
  try { myThread.sleep(1000); } catch (InterruptedException e) {}
         }
   }}
```

Использование потоков в апплетах

Очень часто возникает ситуация, когда много потоков, обращающихся к некоторому общему ресурсу, начинают мешать друг другу. Например, когда один поток читает запись из файла, а другой записывает информацию в файл. Для предотвращения такой ситуации используется ключевое слово **synchronized**.

```
// пример #6 : освобождение ресурсов апплетом : AppletThread.java
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class AppletThread extends Applet implements Runnable {
  final Font font = new Font("Arial", Font.BOLD, 40);
  String msg = "Java 2";
  Thread t = null;
   boolean stop;
public void init() {
   setBackground(Color.blue);
   setForeground(Color.yellow);
}
public void start() {
   t = new Thread(this);
   stop = false;
        t.start(); }
public void run() {
         char ch;
while(true){
       try{ repaint();
        Thread.sleep(300);
        ch = msg.charAt(0);
        msg = msg.substring(1, msg.length()); msg += ch;
            if(stop) break;
} catch(InterruptedException e){}
```

```
            }
}
public void paint(Graphics g) {
    g.setFont(font);
    g.drawString(msg,50,50);}
public void stop() {
stop = true; t = null; }
}
```

Рассмотрим класс **TwoThread,** в котором создается два потока. В этом же классе создаётся экземпляр класса **SourceValue**, содержащий переменную типа **String**. Экземпляр **SourceValue** передается в качестве параметра обоим потокам. Первый поток добавляет экземпляр класса **SourceValue**, а второй извлекает его. Для избежания одновременных действий методы, осуществляющие чтение и запись переменной, объявляются **synchronized**. Синхронизированный метод изолирует объект, содержащий этот метод, после чего объект становится недоступным для других потоков. Изоляция снимается, когда поток полностью выполнит соответствующий метод. Другой способ снятия изоляции – вызов метода **wait()** из изолированного метода.

```
// пример #7 : синхронизированные потоки : TwoThread.java
import java.util.*;
import java.awt.event.*;
public class TwoThread{
    public static void main(String[] args){
       SourceValue sv = new SourceValue();
        ThreadB t1 = new ThreadB(sv, "добавление");
        ThreadB t2 = new ThreadB(sv, "извлечение");
   t1.start();
   t2.start();                              }
}
class ThreadB implements Runnable{
 SourceValue res;
 String name;
ThreadB (SourceValue res , String name){
   this.name = name;
   this.res = res;
}
public void run(){
  int j = 0;
while (true){
```

```
    if (name.equals("добавление")){
                res.addSource("значение:  " + j);
                j++;
                if(j == 5) break;}
else System.out.println("это значение: "+res.getSource());
  try{sleep(100);}
 catch(InterruptedException e){}
                }
   }
}
```

Имеется два класса – открытый **TwoThread** и закрытый **ThreadB**. Основные действия происходят в классе **TwoThread**. Здесь создается экземпляр класса **SourceValue** и два экземпляра класса **ThreadB**, после чего экземпляр **SourceValue** передаётся обоим экземплярам **ThreadB** (потокам). Метод **start()** запускает поток и вызывает метод **run().** После окончания метода **run()** поток уничтожается. Внутри запускающего поток метода **run()** вызывается метод **addSource()** объекта класса **SourceValue**, для первого по времени запуска потока и **getSource()** – для второго.

```
class SourceValue {
        String theSource = "";
        boolean reading = false;
public synchronized void addSource(String str){
  while(reading){
    try{ wait(); }
catch(InterruptedException e){}
                    }
 System.out.println("добавление: "+str);
    theSource = str;
    reading = true;
        notifyAll();//уведомляет другие потоки о снятии изоляции
}
public synchronized String getSource(){
        while(!reading){
  try{ wait(); } catch(InterruptedException e){}
}
     reading = false;
        notifyAll();
return theSource;
   }
```

}

Дополнительная переменная **reading** определяет состояние потока относительно операций чтения или записи. Как только метод **addSource()** начнет выполняться, он сразу же заблокирует объект. И пока он изменяет значение переменной, ни один другой поток не может вызвать синхронизированный метод **getSource()**. После записи значение булевой переменной устанавливается **true**. Для того чтобы поток чтения узнал о снятии изоляции, вызывается метод **notifyAll().** После этого поток чтения может производить свои действия.

В следующем примере производится чтение информации из одного файла и запись в другой файл. Создаются и открываются два различных потока ввода/вывода, каждый из которых соединен с файлом.

```
//пример #8: синхронизированные потоки в файлах: FileThrd.java
import java.io.*;
public class FileThrd{
  public static void main(String[] args){
        Files f = new Files();
        new ThreadForRead(f);
        new ThreadForWrite(f);          }
}
class ThreadForRead implements Runnable{
        Files f1;
        boolean action = true;
ThreadForRead (Files f){
    this.f1 = f;
     new Thread(this,"ThreadForRead").start();
}
public void run(){
  while (action){    action = f1.fileRead();    }
  }
}
class ThreadForWrite implements Runnable{
        Files f2;
        boolean action = true;
ThreadForWrite (Files f) {
  this.f2 = f;
  new Thread(this, "ThreadForWrite").start();
}
public void run(){
        while(action){ action = f2.fileWrite();}
```

```
  }
}
class Files{
        boolean action = true, NoEof = true, lastByte = false;
        int size,b;
        InputStream fInput;
        OutputStream fOutput;
Files(){
 try{
  fInput = new FileInputStream("for_read.txt");
  fOutput = new FileOutputStream("for_write.txt");
}catch(FileNotFoundException e){System.out.println("нет файла!");}
        }
synchronized boolean fileRead(){
        if(!action)   try{
                wait(); }
                        catch(InterruptedException e){}
   try{
 b = fInput.read();
 action = false;
   System.out.println("прочитан: "+(char)b);
 size = fInput.available();
        if(size == 0) {
 NoEof = false;
 lastByte = true;
    fInput.close();
}
       notify();
}catch(IOException e){System.out.println("в fileRead"+e);}
    return NoEof;
}
synchronized boolean fileWrite(){
        if(action)   try {
              wait(); }
                      catch(InterruptedException e){};
try{
 if (size == 0){
  if (lastByte) fOutput.write((char) b);
    fOutput.close();
    NoEof = false;
```

```
                }
  else{
    fOutput.write((char) b);
    action = true;
     notify();
    }
  System.out.println("записан:"+(char)b);
}  catch(IOException e){System.out.println("в fileWrite"+e);}
   return NoEof;
        }
}
```

Упражнения

1. class Counter { int count = 0;
int increment() { int n=count; count = n + 1; return n; } }
Что произойдет, если два потока одновременно вызовут метод increment()?
а) ошибка компиляции; б) ничего особенного; в) выведено одинаковое значение для обоих потоков; г) ConcurrentAccessException; д) в обоих потоках выполнится count = n + 1; в то же время переменная не будет увеличена.

2. Создать апплет, используя поток: строка движется горизонально, отражаясь от границ апплета и меняя при этом случайным образом свой цвет.

3. Создать апплет используя поток: строка движется по диагонали. При достижении границ апплета все символы строки случайным образом меняют регистр.

4. Организовать сортировку массива методами Шелла, Хора, пузырька, на основе бинарного дерева, быстрой сортироки в разных потоках.

5. Реализовать сортировку графических объектов, используя алгоритмы из упражнения 4.

12. СЕТЕВЫЕ ПРОГРАММЫ

Java делает сетевое программирование простым благодаря наличию специальных средств и класса **Network**. Рассмотрим некоторые виды сетевых приложений. Internet-приложения включают Web-браузер, e-mail, сетевые новости, передачу файлов и telnet. Основной используемый протокол – TCP/IP.

Приложения клиент/сервер используют компьютер, выполняющий специальную программу – сервер, которая предоставляет услуги другим программам – клиентам. Клиент – это программа, получающая услуги от сервера. Клиент-серверные приложения основаны на использовании верхнего уровня протоколов. На TCP/IP основаны следующие протоколы:

HTTP – Hypertext Transfer Protocol (WWW);

NNTP – Network News Transfer Protocol (группы новостей);

SMTP – Simple Mail Transfer Protocol (посылка почты);
POP3 – Post Office Protocol (чтение почты с сервера);
FTP – File Transfer Protocol (протокол передачи файлов);
TELNET – Удаленное управление компьютерами;

Каждый компьютер по протоколу TCP/IP имеет уникальный IP-адрес. Это 32-битовое число, обычно записываемое как четыре числа, разделенные точками, каждое из которых изменяется от 0 до 255. IP-адрес может быть временным и выделяться динамически для каждого подключения или быть постоянным, как для сервера. Обычно при подключении к компьютеру вместо числового IP адреса используются символьные имена (например – www.bsu.by), называемые именами домена. Специальная программа DNS (Domain Name Sever) преобразует имя домена в числовой IP-адрес. Получить IP-адрес в программе можно с помощью объекта класса **InetAddress** из пакета **java.net**.

```
/*пример #1 : вывод IP-адреса локального компьютера, подключенного к Internet : MyLocal.java */
import java.net.*;
 public class MyLocal {
   public static void main(String[] args){
        InetAddress myIP = null;
   try {
         myIP = InetAddress.getLocalHost();}
   catch (UnknownHostException e) {}
                System.out.println(myIP);
        }
}
```

Метод **getLocalHost()** класса **InetAddress** создает объект **myIP** и возвращает IP-адрес.

Следующая программа демонстрирует, как получить IP-адрес из имени домена с помощью сервера имен доменов (DNS), к которому обращается метод **getByName().**

```
/*пример #2 : извлечение IP-адреса из имени домена :
IPfromDNS.java */
import java.net.*;
public class IPfromDNS {
  public static void main(String[] args){
        InetAddress bsu = null;
   try {
        bsu = InetAddress.getByName("www.bsu.by"); }
```

```
  catch (UnknownHostException e){ }
 System.out.println(bsu);
  }
}
```

Будет выведено: *www.bsu.by/217.21.43.2*

Для явной идентификации услуг к IP-адресу присоединяется номер порта через двоеточие, например 217.21.43.2:31. Номера портов от 1 до 1024 используются, например, для запуска двух программ серверов на одном компьютере. Если порт явно не указан, браузер воспользуется значением по умолчанию: **20 – FTP-**данные**, 21 – FTP-**управление**, 23 – TELNET, 53 – DNS, 80 – HTTP, 110 – POP3, 119 – NNTP.**

Адрес URL (Universal Resourse Lacator) состоит из двух частей – префикса протокола (http, ftp…) и URI (Universal Resource Identifier). URI содержит Internet-адрес, необязательный номер порта и путь к каталогу, содержащему файл, например: http://www.bsu.by:80/index.htm.

URI не может содержать такие специальные символы, как пробелы, табуляции, возврат каретки. Их можно задавать через шестнадцатиричные коды. Например **%20** обозначает пробел. Другие зарезервированные символы: / – разделитель каталогов, ? – следует за аргументами запросов, # – разделитель аргументов, + – специальный символ.

Можно создать объект класса **URL**, указывающий на ресурсы в Internet. В следующем примере объект **URL** используется для доступа к HTML-файлу, на который он указывает и отображает его в окне браузера с помошью метода **showDocument().**

```
//пример #3 : вывод документа в браузер : MyShowDocument.java
import java.applet.*;
import java.net.*;
import java.awt.*;
public class MyShowDocument extends Applet {
        URL press = null;
public void init() {
   try {
 press = new URL("http://www.pressball.by");
  }catch (MalformedURLException e){
     System.out.println("ошибка: " + e.getMessage());
  }
}
public boolean mouseDown(Event evt, int x, int y) {
// при щелчке происходит переход к странице  www.pressball.by
   getAppletContext().showDocument(press, "_blank");
```

```
   return true;
  }
}
```

Метод **showDocument()** может содержать параметры для отображения страницы различными способами: "**_self**" – выводит документ в текущем окне, "**_blank**" – документ в новое окно, "**_top**" – вверху окна, "**_parent**" – документ в родственном окне, "**строка**" – документ в окне с данным именем.

В следующем примере методы **getDocumentBase()** и **getCodeBase()** используются для получения URL страницы апплета и URL апплета.

```
//пример #4 : получение URL-ов : MyDocumentBase.java
import java.applet.*;
import java.net.*;
import java.awt.*;
public class MyDocumentBase extends Applet {
  public void init() {
    URL html = getDocumentBase();
    URL codebase = getCodeBase();
  System.out.println("URL страницы : " + html);
  System.out.println("URL апплета : " + codebase);
   }
}
```

В следующей программе читается содержимое HTML-файла с сервера и выводится в окно консоли.

```
//пример #5 : чтение HTML-файла : MyURLTest.java
import java.net.*;
import java.io.*;
public class MyURLTest {
  public static void main(String[] args) {
        try {
         URL    homeBSU = new URL("http://www.bsu.by");
InputStreamReader isr = new
InputStreamReader(homeBSU.openStream());
BufferedReader d = new BufferedReader(isr);
 String line = d.readLine();
          while (line != null) {
                        System.out.println(line);
                        line = d.readLine(); }
               }
        catch (IOException e) {
```

```
System.out.println("ошибка: " + e.getMessage());}
    }
}
```

Сокеты и сокетные соединения

Сокеты – это сетевые разъемы, через которые осуществляются двунапрвленные поточные соединения между компьютерами. Сокет определяется номером порта и IP-адресом. При этом IP-адрес используется для идентификации компьютера, номер порта – для идентификации процесса, работающего на компьютере. Когда одно приложение знает сокет другого, создается сокетное соединение. Клиент пытается соединиться с сервером, инициализируя сокетное соединение. Сервер ждет, пока клиент не свяжется с ним. Первое сообщение, посылаемое клиентом на сервер, содержит сокет клиента. Сервер в свою очередь создает сокет, который будет использоваться для связи с клиентом, и посылает его клиенту с первым сообщением. После этого устанавливается коммуникационное соединение.

Сокетное соединение с сервером создается с помощью объекта класса **Socket**. При этом указывается IP-адрес сервера и номер порта (80 для HTTP). Если указано имя домена, то Java преобразует его с помощью DNS-сервера к IP-адресу:

```
try { socket = new Socket("localhost",8080); }
catch (IOException e){ System.out.println("ошибка: " + e); }
```

Сервер ожидает сообщения клиента и должен быть запущен с указанием определенного порта. Объект класса **ServerSocket** создается с указанием конструктору номера порта и ожидает сообщения клиента с помощью метода **accept()**, который возвращает сокет клиента:

```
Socket socket = null;
try {   server = new ServerSocket(8080);
        socket = server.accept(); }
catch (IOException e) { System.out.println("ошибка: " + e); }
```

Клиент и сервер после установления сокетного соединения могут получать данные из потока ввода и записывать данные в поток вывода с помощью методов **getInputStream()** и **getOutputStream()** или к **PrintStream** для того, чтобы программа могла трактовать поток как выходные файлы.

В следующем примере для посылки клиенту строки *"привет!"* сервер вызывает метод **getOutputStream()** класса **Socket.** Клиент получает данные от сервера с помощью метода **getInputStream()**. Для разъединения клиента и сервера после завершения работы сокет закрывает-

ся с помощью метода **close()** класса **Socket**. В данном примере сервер посылает клиенту строку *"привет!"* после чего разрывает связь.

```
//пример #6 : передача клиенту строки : MyServerSocket.java
import java.io.*;
import java.net.*;
  public class MyServerSocket{
     public static void main(String[] args) throws Exception{
        Socket s = null;
try {//посылка строки клиенту
   ServerSocket server = new ServerSocket(8030);
    s = server.accept();
PrintStream ps = new PrintStream(s.getOutputStream());
       ps.println("привет!");
       ps.flush();
   s.close(); // разрыв соединения
 }catch (IOException e){System.out.println("ошибка: " + e); }
   }
}
```

```
//пример #7 : получение клиентом строки : MyClientSocket.java
import java.io.*;
import java.net.*;
public class MyClientSocket {
     public static void main(String[] args) {
        Socket socket = null;
  try {//получение строки клиентом
    socket = new Socket("www.bsu.by", 8080);
    BufferedReader dis = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
  String msg = dis.readLine();
   System.out.println(msg);
} catch (IOException e) {System.out.println("ошибка: " + e); }
   }
}
```

Аналогично клиент может послать данные серверу через поток вывода с помощью метода **getOutputStream(),** а сервер может получать данные с помощью метода **getInputStream()**.

Если необходимо протестировать подобный пример на одном компьютере, можно соединится самому с собой, используя статические методы **getLocalHost()** класса **InetAddress** для получения динамического IP-адреса компьютера, который выделяется при входе в Internet.

Многопоточность

Сервер должен поддерживать многопоточность, иначе он будет не в состоянии обрабатывать несколько соединений одновременно. Сервер содержит цикл, ожидающий нового клиентского соединения. Каждый раз, когда клиент просит соединения, сервер создает новый поток. В следующем примере создается класс **NetServerThread**, расширяющий класс **Thread.**

```
/*пример #8 : сервер для множества клиентов :
NetServerThread.java*/
import java.net.*;
import java.io.*;
 public class NetServerThread extends Thread {
        Socket socket;
        int i;
        PrintStream ps;
 public NetServerThread(Socket s) {
        socket = s;
  try {ps = new PrintStream(s.getOutputStream()); }
  catch (IOException e)  { System.out.println("ошибка: " + e); }
}
public static void main(String[] args) {
        Socket s = null;
        try{
          ServerSocket  server = new ServerSocket(8030);
          s = server.accept();
        NetServerThread nst = new NetServerThread(s);
        nst.start();
        }catch(Exception e){ System.out.println("ошибка: " + e); }
        }
public void run() {
         while (true) {
                String  line = "сообщение: "+i++;
                  if (line == null) return; // клиент ушел
                      send(line);           }
        }
public void send(String msg) {
        ps.println(msg);
        System.out.println(msg+"<передача>");
        ps.flush();
```

```
        }
}
```

Сервер передает сообщение, посылаемое клиенту. Для клиентских приложений поддержка многопоточности также необходима. Например, один поток ожидает выполнения операции ввода/вывода, а другие потоки выполняют свои функции.

```
/* пример #9 : получение сообщения клиентом в потоке :
NetClientThread.java */
import java.net.*;
import java.io.*;
public class NetClientThread extends Thread {
  BufferedReader br = null;
  Socket s = null;
public NetClientThread() {
      try {//соединение с кольцевым адресом
   s = new Socket("127.0.0.1", 8030);
InputStreamReader isr = new
InputStreamReader(s.getInputStream());
br = new BufferedReader(isr);
  } catch (IOException e) {      System.out.println("ошибка: " + e);  }
}
public static void main(String[] args) {
        NetClientThread nct = new NetClientThread();
        nct.start();  }
public void run() {
              while (true) {
  try {   String msg = br.readLine();
                                if (msg == null) break;
                               else System.out.println(msg);}
 catch (IOException e) {  System.out.println("ошибка: " + e); }
        }
    }
}
```

Сервер должен быть готов до того, как клиент попытается осуществить сокетное соединение. При этом может быть использован IP-адрес локального компьютера.

Упражнения

1. import java.io.*;
   import java.net.*;
   public class NetClient {

public static void main(String args[]) { try {
Socket skt = new Socket("host",88);
Каким способом будет подключен объект skt?
а)никаким ; б)POP3; в)FTP; г)HTTP

2. Организовать соединение между клиентом и сервером используя визуальные компоненты?

3. Как получить содержание страницы используя его URL?
а)Socket content = new Socket(new URL(url)).collect();
б)Object content = new URL(url).getContent();
в)String content = new URLHttp(url).getString();
г)Object content = new URLConnection(url).getContent();
д)String content = new URLConnection(url).collect();

Вот и все, что мы смогли изложить на отведенных нам 80-ти страницах. Имейте в виду, что это только основа. Теперь примените полученные знания при изучении и использовании сервлетов, JSP, JDBC, EJB, RMI и прочих API.

Литература

1. М.Чен, С.Гриффис, Э.Изи., Программирование на Java: 1001 совет. Попурри.,Минск – 1997.1000с.
2. Д.Родли., Создание Java апплетов.: ДиаСофт.,Киев-1996. 380с.
3. К.Джамса, С.Лалани, С.Уикли., Программирование в Web для профессионалов.:Попурри.,Минск.- 1997.630с.
4. Ноутон О., Шилдт Г., Java 2 в подлиннике.:BHV., СПб – 2000.,1050с.
5. Эккель Б. Философия Java. :BHV., СПб – 2001. 850с.
6. Шарма В., Шарма Р. Разработка Web-серверов для электронной коммерции. - 2001., 350с.
7. Кубенский А. Создание и обработка структур данных на Java.: BHV., СПб – 2000.,322с.

Учебное издание

Блинов Игорь Николаевич
Романчик Валерий Станиславович

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ JAVA

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО КУРСУ “МЕТОДЫ ПРОГРАММИРОВАНИЯ”

Для студентов механико-математического факультета

В авторской редакции

Подписано в печать 12.06.2002. Формат 60x84/16. Бумага офсетная.
Печать офсетная. Усл. печ.л.2,79. Уч.-изд.л.1,59. Тираж 100 экз. Зак.

Белорусский государственный университет
Лицензия ЛВ №315 от 14.07.98.
220050, Минск, пр. Ф.Скорины, 4.

Отпечатано в Издательском центре БГУ,
220030, Минск, ул. Красноармейская, 6.