Министерство образования и науки российской федерации

НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

ТЕХНОЛОГИЯ РАЗРАБОТКИ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММ НА JAVA

Учебно-методическое пособие
по дисциплинам «Объектно-ориентированное программирование» и «Технологии программирования» для студентов для студентов IV курса очной формы обучения и V курса заочной формы обучения АВТФ (направление 230100.62 «Информатика и вычислительная техника» и специальность 230101.65 «Вычислительные машины, комплексы, системы и сети»

НОВОСИБИРСК
2012

Составитель: Васюткина И.А., канд. техн. наук, доцент

Рецензенты: Малявко А.А., канд. техн. наук, доцент
Гунько А.В., канд. техн. наук, доцент

Работа подготовлена на кафедре Вычислительной техники

ОГЛАВЛЕНИЕ

Предисловие

Данное учебно-методическое пособие содержит теоретический материал, изучаемый студентами IV курса очной формы обучения в дисциплине «Объектно-ориентированное программирование», IV курса очной формы обучения и V курса заочной формы обучения АВТФ в дисциплине «Технологии программирования».

В 8 главах пособия рассмотрены технологии разработки приложений и апплетов на языке Java в среде NetBeans 7.0.; разработки иерархий классов и интерфейсов; проектирования графического интерфейса пользователя, обработки событий; работы с байтовыми и символьными потоками ввода-вывода, сериализации и десериализации объектов; создания многопотоковых и «клиент-серверных» сетевых приложений; использования классов-коллекций для хранения и обработки данных, а также разработки универсальных generic- классов.

В каждой главе имеется достаточно большое количество примеров программ для лучшего понимания, излагаемого материала. В конце каждой главы имеются практические задания для закрепления материала и получения навыков программирования. Завершается каждая глава вопросами для самопроверки полученных знаний.

ГЛАВА 1. ОСНОВЫ ПРОГРАММИРОВАНИЯ НА JAVA. СОЗДАНИЕ ПРОСТЕЙШИХ ПРИЛОЖЕНИЙ И АППЛЕТОВ В NETBEANS 7.0

При появлении Java двумя основными формами Java-программ являлись **приложение** и **апплет**.

Java-приложения выполняются под управлением специального интерпретатора (java.exe). Приложения похожи на программы, созданные, например, с использованием языка C/C++, хотя для своей работы они требуют присутствия Java виртуальной машины (JVM). Это полноправные приложения, которые существуют и выполняются в локальных компьютерных системах пользователей.

Java-апплеты разработаны для функционирования в сети и выполняются как часть Web-станиц. Апплеты требуют наличия соответствующего Java-браузера, так как они должны загружаться по сети с Web-сервера в обеспечивающую их работоспособность среду исполнения Java на локальном компьютере.

Инструментальная среда разработки программ на Java

Для создания программ на Java возможно использование нескольких сред разработки. Это может быть ***Microsoft Visual J++, JBuilder,IntelliJ Idea, Eclipse или NetBeans IDE***.

Использование среды NetBeans 7.0

1. Создание нового Java-проекта. (File – New Project) (рис. 1.1).
2. Настройка проекта (рис. 1.2). Необходимо задать имя проекта (в примере LabWorkN), его расположение и имя главного класса (в примере labworkn.Main). Обратите внимание, что имя разделено точкой. До точки – имя пакета (package), после – имя класса. Названия пакетов принято писать строчными буквами, а заглавные использовать только в именах классов.

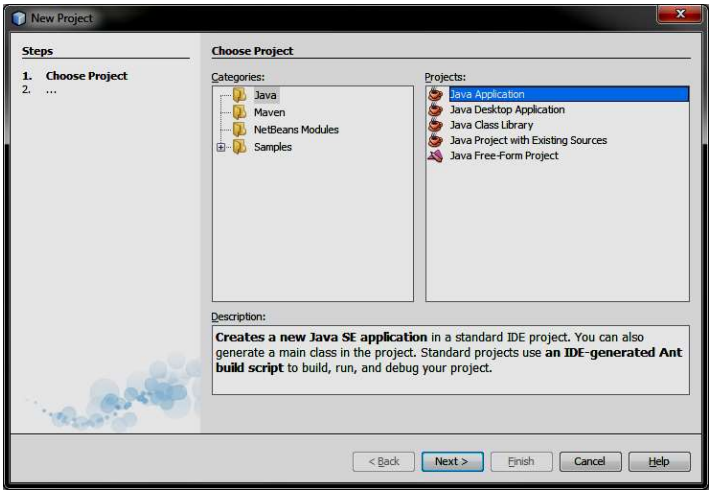


Рис. 1.1. Создание нового проекта

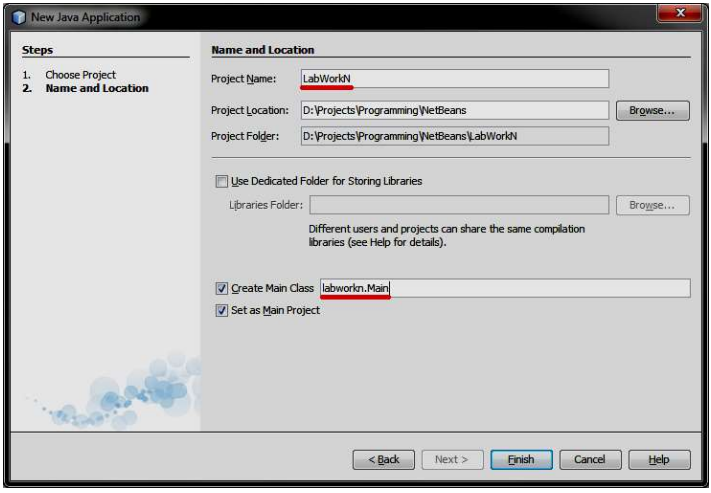


Рис. 1.2. Задание имен проекта и главного класса

Добавление класса, интерфейса, апплета и т.д. в проект (правая кнопка мыши по пакету) (рис. 1.3).

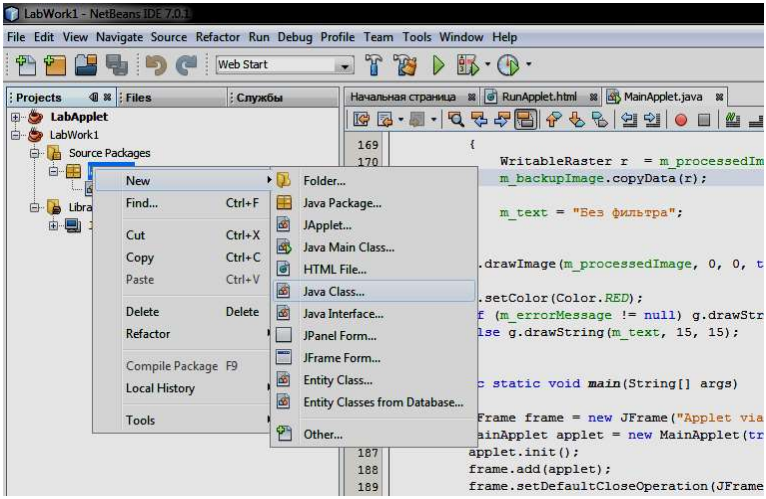


Рис. 1.3. Добавление новых модулей в проект

3. Если проект не был скомпилирован или был изменен, при запуске (через панель инструментов (см. рис.1.4), меню Run или F6) будет произведена компиляция. Результат выполнения консольного приложения, описание ошибок и исключений выводится в окно Output. Компиляция может быть произведена отдельно, без запуска (Build Main Project или Clean and Build Main Project).

4. Для отладки программы устанавливаем точку останова щелчком по полю с номером строки кода. Появится красный квадрат напротив этой строки. При помощи команды Debug Main Project (Ctrl + F5) запускаем программу в режиме отладки (см. рис.1.5).

5. Создание простейшего приложения. Java-файл будет содержать следующий код:

```
import java.util.*;

public class Hello {
```

```
    public static void main(String args[]) {
        System.out.println("Hello, world");
        Date d=new Date();
        System.out.println("Date:"+d.toString());
    }
}
```

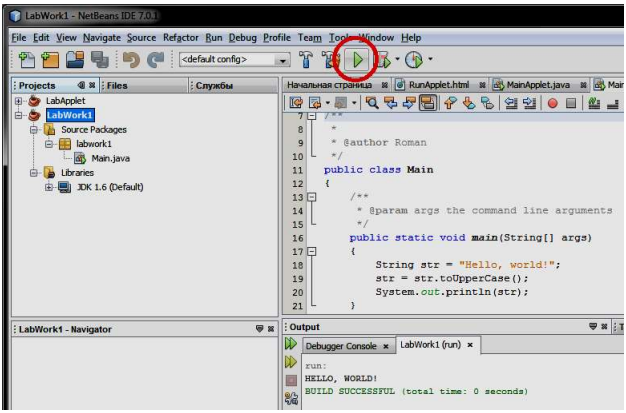


Рис. 1.4. Запуск программы

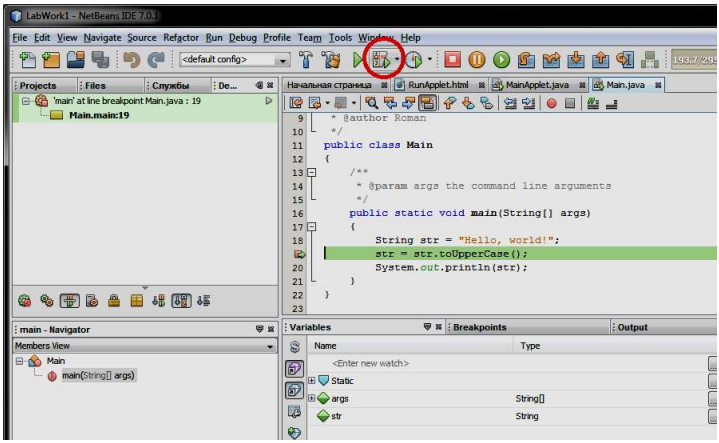


*Рис. 1.5.*Отладка программы

Так как класс Hello объявлен с модификатором public, то имя файла с его исходным кодом должно совпадать с именем класса. Для классов с модификатором по умолчанию имена файлов могут быть любыми (расширение обязательно .java).

Все классы являются производными (или подклассами) от существующих классов. В случае - если не определен **суперкласс,** то по умолчанию предполагается, что таким суперклассом является класс **Object**.

Структура Java-программы

Все Java-программы содержат в себе четыре разновидности блоков: классы (classes), методы (methods), переменные (variables) и пакеты (package).

Классы представляют собой основу объектно-ориентированных свойств языка. Классы содержат переменные и методы. Методы есть не что иное как функции или подпрограммы. В переменных хранятся данные.

Пакеты содержат в себе классы и помогают компилятору найти те классы, которые нужны ему для компиляции пользовательской программы. Классы, входящие в один пакет, особым образом зависят друг от друга. Например, приложение Hello импортирует пакет java.util, в котором содержится класс Date.

Java-программа может содержать в себе любое количество классов, но один из них всегда имеет особый статус и непосредственно взаимодействует с оболочкой времени выполнения (**первичный класс**). Для приложений первичный класс должен обязательно содержать метод **main**().

Переменные

В Java существует два вида переменных:

- **примитивные типы** (primitive types). К ним относятся стандартные, встроенные в язык типы для представления численных значений, одиночных символов и булевских

(логических) значений. Все примитивные типы имеют предопределенный размер занимаемой ими памяти.

- **ссылочные типы** (reference type) - относятся типы, определенные пользователем (классы, интерфейсы) и типы массивов. Все ссылочные типы являются динамическими типами, для них выделяется память во время работы программы.

Переменные примитивных типов передаются в методы по значению, тогда как ссылочные переменные всегда передаются по ссылке.

Примитивные типы. Всего в Java определено восемь примитивных типов: int (4b), short (2b), byte (1b), long (8b), float (4b), double (8b), boolean (true, false), char (2b).

Первые шесть типов предназначены для хранения численных значений, с ними можно производить арифметические операции. Тип char предназначен для хранения символов в стандарте Unicode (2 байта). Булевские (логические) переменные могут иметь одно из двух допустимых значений: true или false.

Ссылочные типы. Переменной ссылочного типа выделяется память при помощи оператора new. Таким образом, каждая переменная ссылочного типа является объектом или экземпляром соответствующего типа.

Типы, определенные пользователем. Язык Java позволяет определять новые типы помощью новых классов. Рассмотрим пример определения и использования нового класса (нового типа) MyType:

Пример 1.1. Файл NewClass.java

```
class MyType  {              // объявляется класс
    public int myData=5;  // переменная-элемент класса
    MyType() {                        // конструктор без параметров
        System.out.println("Constructor without parameters");
    }
    MyType(int v) {                  // конструктор с одним параметром
        System.out.print("Constructor with one parameter");
```

```
        System.out.println(" Setting myData="+v);
        myData=v;
    }
    public void myMethod() {        // метод класса
        System.out.print("myMethod!");
        System.out.println(" myData="+myData);
    }
}

//  Реализация объектов и действия с ними
public class NewClass {    // первичный класс
    public static void main(String[] args)    {
        // объект obj1 - реализация класса MyType
        MyType obj1=new MyType();
        obj1.myMethod();  // использование метода
        // доступ к открытой переменной
        System.out.println("---obj1.myData="+obj1.myData);
        // объект obj2 - реализация класса MyType
        MyType obj2=new MyType(100);
        // доступ к открытой переменной
        System.out.println("----obj2.myData="+obj2.myData);
    }
}
```

Класс String (тип строковых переменных). Класс String – это класс неизменяемых строк. Данный класс представляет собой гибрид примитивных и ссылочных типов. В основе своей тип String является ссылочным типом, содержащий в себе методы и переменные. Но в то же время этот тип проявляет некоторые свойства примитивного типа, что выражается, в том, как осуществляется присвоение значение этим переменным при помощи знака операции = (но можно для инициализации создаваемых объектов пользоваться и явным вызовом конструктора), например:

```
String S1="Hello";                    // 1-ый способ инициализации
String S2=new String("Hello");    // 2-ый способ инициализации
```

Кроме того, строковый тип проявляет свойства примитивных типов в том, что для конкатенации (сложения) двух строк можно использовать знак операции +, например:

```
String S0="Variable ";
int myInt=3;
String S1=S0+"myInt"+myInt;
String S2=new String(S0+"myInt"+myInt);
```

Несмотря на поддержку таких операций, строковые переменные типа String являются в то же время и объектами, для них можно вызывать методы класса String, например, узнать длину строки:

```
int len=S1.length();
```

Итак, тип String является особым - это единственный класс, переменные которого могут объявляться и использоваться без применения оператора new.

Типы массива. Типы массива используются для определения массивов - упорядоченного набора однотипных переменных. Можно определить массив любого существующего типа, включая типы, определенные пользователем. Кроме того, можно пользоваться и массивами массивов или многомерными массивами.

Создание массивов требует использования оператора new, например:

```
// объявление ссылочной переменной типа массив
int intArray[];
intArray=new int[3];   // создание массива целого типа
// создание двумерного массива вещественного типа
float[][] flArray = new float[2][3];
// создание массива и инициализация
double[] a = {0.01, -3.4, 2.89, 4.5, -6.7};
// создание и инициализация двумерного массива переменных целых
int[][] arrayOfInts = { { 32, 87, 3, 589 }, { 12, 1076, 2000, 8 }, { 622, 127, 77, 955 } };
```

Массивы Java имеют три важных преимущества перед массивами других языков:

- программисту не обязательно указывать размер массива при его объявлении;
- любой массив в языке Java является переменной - а это значит, что его можно передать как параметр методу и использовать в качестве значения, возвращаемого методом;
- размер массива в любой момент времени можно узнать через специальную переменную length, имеющуюся в каждом массиве, например:

```
int len=intArray.length;
```

Классы

Статические и динамические элементы класса. Если при определении элемента не используется ключевое слово static, то этот элемент по умолчанию является динамическим (dynamic).

Динамические методы и переменные всегда являются элементами объекта класса, и доступ к ним осуществляется через переменную-объект.

ИмяОбъекта.ИмяПеременнойЭкземпляра
ИмяОбъекта.ИмяМетодаЭкземпляра(<Параметры>)

Статические методы и переменные связаны с классом, а не с экземпляром класса, и являются элементами класса. Их можно использовать без создания объекта этого класса, доступ осуществляется через имя класса.

ИмяКласса.ИмяПеременнойКласса
ИмяКласса.ИмяМетодаКласса(<Параметры>)

Статические методы класса могут работать только со статическими переменными класса, для которых также не важно наличие реализованного объекта. Эти элементы класса используются всеми объектами класса (для всех объектов существует только один экземпляр статической переменной).

Пример 1.2. Файл TestElements.java

```
class StaticAndDynamic {
    int i=0;                // переменная экземпляра
    static int j=0;         // переменная класса
    // статические методы
    static void setJ(int k)     {
        System.out.println("Static Method"); j=k;
    }
    static int getJ()       {
        System.out.println("Static Method"); return j;
    }
    // динамические методы
    void setI(int k)        {
        System.out.println("Dynamic Method"); i=k;
    }
    int getI()              {
        System.out.println("Dynamic Method"); return i;
    }
    int summa()   {
    // в динамических методах можно использовать статические
переменные
        System.out.println("Dynamic Method"); return i+j;
    }
}
public class TestElements         {
    public static void main(String args[]){
         int ii,jj;
        // использование статической переменной
         StaticAndDynamic.j=6;      jj=StaticAndDynamic.j;
        System.out.println("Main, jj="+jj);
        // вызов статических методов
        StaticAndDynamic.setJ(4);
        jj=StaticAndDynamic.getJ();
        System.out.println("Main, jj="+jj);

        StaticAndDynamic obj=new StaticAndDynamic();
        // использование динамической переменной
        obj.i=3;      ii=obj.i;
        System.out.println("Main, ii="+ii);
        // вызов динамическим методов
```

```
        obj.setI(8);      ii=obj.getI();
        System.out.println("Main, ii="+ii);
        ii=obj.summa();
        System.out.println("Main, summa="+ii);
    }
}
```

Модификаторы доступа. Модификаторы доступа используются для управления доступностью элементов класса из других частей программы (в других классах).

Элемент, объявленный с ключевым словом **public** (открытый), доступен во всех классах, как в своем пакете, так и во всех классах в любом другом пакете. Этот модификатор можно использовать при объявлении класса. Тогда этот класс доступен для всех классов других пакетов. В каждом файле должен содержаться только один public класс и имя файла должно совпадать с именем такого класса.

Элемент, объявленный с модификатором **protected** (защищенный) в некоем классе A, доступен во всех классах, являющихся подклассами класса A.

Модификатор **private** (закрытый) сильнее всего ограничивает доступность элемента. Он его делает невидимым за пределами данного класса. Даже подклассы данного класса не смогут обращаться к элементу, объявленному как private.

Если тип доступа к элементу не указан (**доступ по умолчанию**), то он доступен из всех классов, входящих в данный пакет.

Пример 1.3. Файл TestModificators.java

```
class A         {
    private int n;
    A() { k=2; n=11; }
    int summa() { return k+n; }
    public int getN() { return n; }
    public void setN(int nn) { n=nn; }
}

class TestModificators {
```

```
    public static void main(String args[]){
        A obj=new A();              // создание объекта класса A
        // получить значение переменных
        int kk=obj.k;       System.out.println("k="+kk);
        int nn=obj.getN();          System.out.println("n="+nn);
        obj.k=10;          obj.setN(15);
        int s=obj.summa();          System.out.println("summa="+s);
    }
}
```

Наследование классов

Наследование (inheritance), упрощает практическое использование классов, так как позволяет расширять уже написанные и отлаженные классы, добавляя к ним новые свойства и возможности. Таким образом создается то, что называется **подклассом** первоначального класса. Класс, который при этом наследуется (расширяется), называется **суперклассом.** При расширении класса имеется возможность использования всех написанных и отлаженных методов этого класса. Это свойство, называемое **повторным использованием кода** (code reuse), является одним из главных преимуществ объектно-ориентированного программирования.

Для задания наследования указывается ключевое слово **extends**. Например, объявление класса MyClass подклассом класса SuperClass, можно записать так:

```
class MyClass extends SuperClass        { ..}
```

Специальные переменные

Каждый класс Java содержит три заранее определенные переменные, которыми можно пользоваться: **null, this, super**. Первые две переменные относятся к типу Object. Коротко говоря, null представляет собой несуществующий объект**,** this указывает на текущий экземпляр класса, переменная super разрешает доступ к открытым переменным и методам, определенным в суперклассе.

Переменная null. Прежде чем использовать какой-то класс, его нужно реализовать, т.е. создать объект (экземпляр) этого класса. До этого ссылка на объект имеет значение **null**.

Переменная this. Переменная this используется для указания ссылки на переменную экземпляра. Переменная this всегда указывает на текущий класс, поэтому, чтобы в методе получить доступ к скрытой переменной, объявленной в текущем классе, нужно явным образом указать принадлежность переменной к классу:

this. ИмяПеременной

Еще одна возможность использования этой ссылочной переменной - вызов в конструкторе класса другого конструктора этого же класса:

this(<ПараметрыДругогоКонструктораКласса>);

Переменная super. Переменная super ссылается на суперкласс объекта. При помощи нее можно получить доступ к затененным переменным и замещенным методам родительского класса**.** Затенение переменной класса в его подклассе возникает при объявлении в подклассе переменной с таким же именем, что и имя переменной суперкласса.

super.ИмяПеременной

При замещении метода в классе создается метод, имеющий то же самое имя и список параметров, что и метод в суперклассе. Однако можно использовать переменную super для выполнения замещенного метода суперкласса:

super.ИмяМетодаСуперкласса(*ПараметрыМетодаСуперкласса*);

Используя переменную super, в конструкторе подкласса можно вызвать конструктор родительского класса. Нужно заметить, что при создании объектов подкласса сначала автоматически вызывается конструктор суперкласса без

параметров (если он не определен, то вызывается задаваемый по умолчанию конструктор без параметров), а затем только конструктор подкласса. Но если в конструкторе подкласса есть явный вызов конструктора суперкласса, то автоматического вызова конструктора суперкласса без параметров не происходит. Сначала вызывается требуемый явным вызовом конструктор суперкласса, а затем только конструктор подкласса.

super(<ПараметрыКонструктораСуперкласса>;

Этот оператор должен быть первым исполняемым оператором конструктора.

Пакеты и импортирование

Классы являются основными строительными блоками любой Java-программы. Пакеты содержат в себе наборы классов (а также исключения и интерфейсы).

Основное назначение пакетов - создание библиотек кода. Пакеты используются для организации и категоризации классов.

Импортирование пакетов. Существует ряд способов доступа к классам в пакетах, основным из которых является импортирование пакета в начале программ:

import ИмяПакета .;*
или
import ИмяПакета.ИмяКлассаПакета;

В этом случае импортирование просто должно предшествовать всем другим строкам программы. Например:

```
import java.applet.*; // импортируются все public классы пакета
import java.awt.*;
import java.net.URL   ; // импортируется только указанный класс
```

Надо заметить, что импортируются всегда только классы, которые используются в программе. После этого классы можно

использовать просто по именам, без указания на принадлежность тому или иному пакету, например:

ИмяКласса Obj = new ИмяКласса(<*Параметры*>);

В случае, когда класс необходимо использовать только один раз, можно явно сослаться на класс, не импортируя пакет. Для этого перед именем класса просто указывается имя пакета, в котором находится этот класс, например:

ИмяПакета.ИмяКласса Obj =
new ИмяПакета.ИмяКласса(Параметры);

Создание пакетов. Для создания пакета (т.е. добавления класса в пакет) в первой строке программы указывается следующее предложение:

package ИмяПакета;

Содержимое пакета может храниться в одном или нескольких файлах. Каждый такой файл должен содержать только один общедоступный (public) класс. При компиляции этих файлов, получающиеся в результате файлы с расширением .class, будут помещены в каталог, с именем пакета.

Апплеты

Апплеты выполняются под управлением виртуальной машины Java, встроенной в браузер. Апплет встраивается в документ HTML и выглядит как окно заранее заданного размера. Он может рисовать в своем окне произвольные изображения или текст. Двоичный файл с исполняемым кодом Java располагается на Web-сервере. В документ HTML с помощью специального оператора организуется ссылка на этот двоичный файл. Когда пользователь загружает в браузер документ HTML с апплетом, файл апплета переписывается с Web-сервера на локальный компьютер пользователя.

Настройка запуска апплета двойного назначения в NetBeans 7.0

1. Щелкаем правой кнопкой по названию проекта и выбираем Properties.
2. В появившемся окне выбираем вкладку Web Start (рис. 1.6).

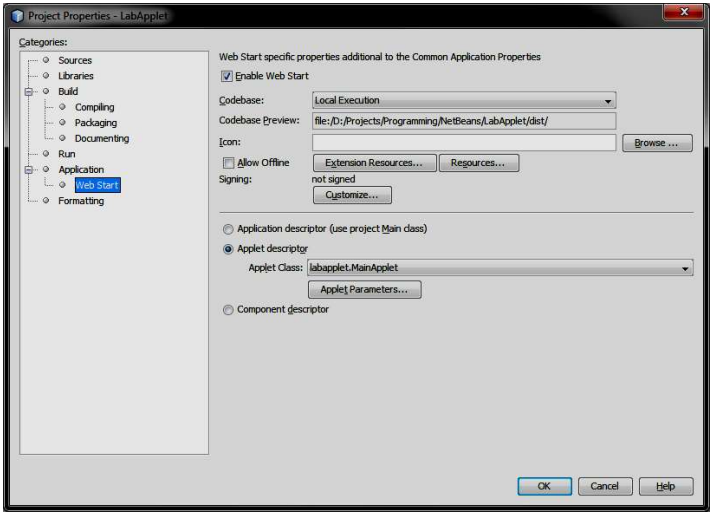


Рис.1.6. Задание параметров апплета

Активируем Enable Web Start. Выбираем в списке Codebase «Local Execution». Выбираем опцию Applet descriptor. Кнопка «Applet Parameters…» открывает окно редактирования параметров.

Здесь можно настроить ширину и высоту окна апплета, а также задать любой набор параметров. Примечательно, что в данной версии NetBeans для того чтобы текстовый параметр сохранился, после ввода текста надо нажать Enter.

3. Для апплета двойного назначения важно наличие главного класса, т.е. класса содержащего метод public static void

main(String[] args). Для настройки в окне Properties выбираем вкладку Run (рис.1.7).

Комбобокс Configuration определяет способ запуска. Web Start – для апплета, <default config> - обычный запуск.

В строке Main Class необходимо выбрать главный класс. Без этого приложение не сможет запуститься в десктопном режиме.

Поле Arguments задает параметры командной строки. Параметры передаются с метод main через массив строк String[] args.

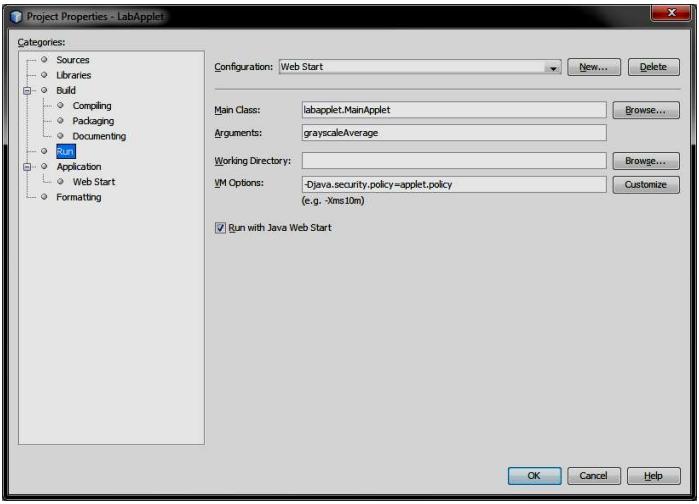


Рис.1.7. Задание параметров апплета двойного назначения

Исходный текст Java-файла простейшего апплета выглядит следующим образом:

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

public class Hello extends Applet{
    public void init() {
        resize(150,150);
    }
```

```
    public void paint(Graphics g)  {
        g.drawString("Hello, WWW", 10, 20);
    }
}
```

При первом запуске апплета в NetBeans в папке build проекта создается файл HTML-документа со ссылкой на апплет, например:

```
<html>
<head> <title>Hello</title> </head>
<body>
<applet codebase="classes" code="javaapplication1/Hello.class"
 width=200 height=200></applet>
</body>
</html>
```

Класс Hello определяется как public, а это значит, что он доступен для других объектов. Кроме того, явно установлен суперкласс Applet (java.applet.Applet). Этот класс должны расширять все апплеты.

Усовершенствованная программа Hello перерисовывает строчку текста в той точке, где пользователь щелкнул мышью.

Пример 1. 4. Файл Hello.java

```
import javax.swing.event.MouseInputAdapter;
import java.applet.*;
import java.awt.*;
import java.awt.event.MouseEvent;

public class Hello extends Applet   {
    int curX=50, curY=50;
    MouseInputAdapter p;
    public  Hello()   {
        // обработчик события
       p = new MouseInputAdapter(){
       public void mousePressed(MouseEvent e) {
            curX=e.getX(); curY=e.getY();
            repaint();
```

```
        }};
      this.addMouseListener(p);
    }
    public void init()  {
      resize(640,480);
    }
     public void paint(Graphics g)  {
       g.drawString("Hello, WWW",curX,curY);
     }
}
```

Обратите внимание, что в методе mousePressed вызывается метод repaint(). Этот метод сообщает оболочке времени выполнения, что необходимо обновить изображение в окне. В примере жирным шрифтом выделен код, связанный с обработкой события. В приведенном коде используется анонимный класс, созданный на основе класса MouseInputAdapter**,** в котором переопределен метод mousePressed.

Основные методы класса Applet. В первичном классе апплета необходимо определить как минимум два метода – init и paint. Метод init выполняет инициализацию апплета, а с помощью метода paint выполняется отрисовка апплета. Другие методы определяются в случае необходимости создания некоторых специальных эффектов.

В методе init**()** по умолчанию вызывается метод resize(), определенный в суперклассе. При помощи этого оператора изменяются размеры окна апплета, установленные в параметрах тега <APPLET>.

Перед удалением апплета из памяти вызывается метод **destroy(),** определенный в классе Applet как пустая заглушка. В методе destroy можно выполнить все необходимые операции перед удалением апплета.

Метод **start()** вызывается после метода init() в тот момент, когда пользователь начинает просматривать документ HTML со встроенным в него апплетом. Метод start**()** можно модифицировать, если при каждом посещении пользователем

страницы с апплетом необходимо выполнять какую-либо инициализацию.

Обратным методу start**()** является метод **stop().** Он вызывается, когда пользователь покидает страницу с апплетом и загружает в окно навигатора другую страницу. Этот метод вызывается перед вызовом метода destroy**().** Метод stop**()** можно дополнить кодом, который должен работать при остановке апплета.

Метод **paint()** выполняет рисование в окне апплета. Определение этого метода находится в классе java.awt.Component.

Передача параметров апплету. Параметры апплетов следуют после открывающего HTML-тега <APPLET> и перед закрывающим тегом </APPLET>. Они определяются как пары, состоящие из двух опций - NAME (имя) и VALUE (значение), внутри тегов <PARAM>, например как в этом примере HTML-документа:

```
<applet code=Hello.class width=200 height=200>
<param name=first value="Hello!">
<param name=second value="How are you?">
<!-- Здесь можно расположить альтернативный текст,
выводящийся в окнах навигаторов, не поддерживающих работу апплетов -->
</applet>
```

Для того, чтобы получить значения этих параметров в апплете, необходимо использовать метод **getParameter()**, определенный в классе Applet, например:

```
String firstParam=getParameter("first");
String secondParam=getParameter("second");
```

Теперь переменные firstParam и secondParam содержат строки "Hello!" и "How are you?" соответственно.

Поскольку метод getParameter() всегда возвращает строки, то для получения численного значения параметра необходимо сделать преобразование в число, напрмер:

```
String loopString=getParameter("loop");
int loopInt=Integer.valueOf(loopString).intValue();
```

Создадим шаблон апплета AppletWithParam, который имеет возможность обрабатывать параметры.

Пример 1.5. Апплет AppletWithParam

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

public class AppletWithParam extends Applet  {

// Поля инициализируются значениями по умолчанию
private String m_String_1 = "First string";
private String m_String_2 = "Second string";

// Имена параметров, нужны для функции getParameter
private final String PARAM_String_1 = "String_1";
private final String PARAM_String_2= "String_2";
public AppletWithParam() {
// код конструктора
}
public String getAppletInfo()  {
    return "Name: AppletWithParam\r\n" +"";
}
// Метод возвращает ссылку на массив с описаниями
// параметров в виде { "Name", "Type", "Description" },
public String[][] getParameterInfo()  {
    String[][] info = {
        { PARAM_String_1, "String", "Parameter description" },
        { PARAM_String_2, "String", "Parameter description" },
    };
    return info;
}
public void init()  {
// Чтение всех параметров и запись значений в поля класса
    String param;
    param = getParameter(PARAM_String_1);
    if (param != null)    m_String_1 = param;
    param = getParameter(PARAM_String_2);
```

```
        if (param != null)    m_String_2 = param;
        resize(320, 240);
    }
    public void destroy()  {  // код завершения работы апплета  }
    public void paint(Graphics g)  {
        g.drawString("Applet with Parameters",10, 20);
        g.drawString(m_String_1,10, 40);
        g.drawString(m_String_2,10, 60);
    }
    public void start()  {
    // код, который должен выполняться при запуске апплета
    }
    public void stop()  {
    // код, который должен работать при остановке апплета
    }
    // код, необходимый для работы апплета
    }
```

Листинг HTML-файла**:**

```
<html>
<head>
<title>AppletWithParam</title>
</head>
<body>
<hr>
<applet
code=AppletWithParam.class name= AppletWithParam
 width=320 height=240 >
<param name=String_1 value="New first string">
<param name=String_2 value="New second string">
</applet>
<hr>
<a href=" AppletWithParam.java">The source.</a>
</body>
</html>
```

В методе paint**()** выведем строки, получаемые апплетом в виде параметров:

```
public void paint(Graphics g) {
```

```
        g.drawString(m_String_1, 10, 20);
        g.drawString(m_String_2, 10, 50);
    }
```

URL-адреса, загрузка и вывод графических изображений. Создадим апплет ParamUrlImage, в котором через параметр апплета передается имя gif-файла, содержимое которого затем выводится в окне апплета. В каталог, где содержится HTML-файл, следует поместить файл с изображением simple.gif

Пример 1.6. Загрузка изображений

```
import java.net.URL;

public class ParamUrlImage extends Applet  {
    Image Im;
    private String m_FileName = "simple.gif";
    private final String PARAM_String_1 = "fileName";
    public String[][] getParameterInfo() {
    String[][] info ={{ PARAM_String_1, "fileName", "name of file" },};
        return info;
   }
    public void init() {
        String param;
        param = getParameter(PARAM_String_1);
        if (param != null)  m_FileName = param;
        Im=getImage( getDocumentBase(),m_FileName);
        resize(320, 240);
    }
    public void paint(Graphics g){
        g.drawImage(Im,0,0,this);
        g.drawString("Applet with Parameters",10, 20);}
    }
```

Загрузить изображение можно при помощи URL-адреса на файл с изображением. URL, или Uniform Resource Locators (“унифицированная ссылка на ресурс”), - полный адрес объекта в сети WWW. В Java есть отдельный класс для обработки URL-адресов - **java.net.URL**. Самый простой способ создания объекта URL состоит в использовании конструктора URL(String add) с

передачей ему WWW-адреса. Фрагмент кода, где происходит реализация класса URL, может сгенерировать исключение MalformedURLException, если вдруг объект не может по какой-либо причине быть создан (например, строка не является URL-адресом). Компилятор требует обработать возможность возникновения исключительной ситуации при помощи блоков try-catch**,** например:

```
try  {
   URL addUrl=new URL("http:// ermak.cs.nstu.ru/");
}  catch(MalformedURLException e)  {
// код здесь выполняется, если строка URL неправильна
}
```

Загружать файлы можно только с того сервера, с которого был загружен сам апплет (это хорошая практика программирования).

Для получения URL на каталог, содержащий класс работающего апплета, используется метод **getDocumentBase()** класса Applet. Так имя загружаемого графического файла содержится в переменной m_FileName класса, то для загрузки графического изображения можно использовать следующий фрагмент, помещаемый в метод **init()** класса ParamUrlImage:

```
Image Im;
try  {
Im=getImage(getDocumentBase(), m_FileName);
} catch(MalformedURLException e)  {
 // код здесь выполняется, если строка URL неправильна
    Im=createImage(0,0); // создание пустого изображения
}
```

Для загрузки изображения в режиме работы приложения можно использовать метод getToolkit:

```
Im = getToolkit().getImage(m_ FileName);
```

Класс **Image** определяет простое двумерное графическое изображение. Этот метод может импортировать изображения любого графического формата, поддерживаемого браузером (чаще всего используются форматы GIF и JPEG).

Двойная буферизация графического изображения. Для ускорения вывода картинок на экран и для устранения эффекта мерцания изображений в процессе вывода большинство апплетов использует двойную буферизацию изображения - сначала загружая изображение в оперативную память, а затем выводя его в окне апплета за один шаг.

Для того, чтобы использовать двойную буферизацию, апплет должен включать в себя метод **imageUpdate()**, который он использует для контроля над тем, какую часть картинки метод drawImage() загрузил в память. После того, как все изображение оказывается в памяти, программа может выводить его быстро, без искажений, которые возникают при одновременной загрузке изображения и выводе его на экран.

Пример 1.7. Двойная буферизация

```
import java.applet.Applet;
import java.awt.*;
import java.net.*;

public class QuickPicture extends Applet  {
Image pic;
boolean picLoaded=false;          // проверка загрузки  изображения
private String m_FileName = "simple.gif";
private final String PARAM_String_1 = "fileName";
int x=0, y=0;
Image offScreenImage;
public void init(){
    String param;
    param = getParameter(PARAM_String_1);
    if (param != null) m_FileName = param;
    pic=getImage(getDocumentBase(), m_FileName);
    resize(320, 240);
}
public void paint (Graphics g){
    // создание виртуального экрана
    int width = getSize().width;
    int height = getSize().height;
```

```
        offScreenImage=createImage(width,height);
        // получение его контекста
        Graphics offScreenGraphics= offScreenImage.getGraphics();
        // вывод изображения на виртуальный экран
        offScreenGraphics.drawImage(pic,x,y,this);
        if(picLoaded)    {
    // 4-м параметром в drawImage() передается null.
    // Он не позволяет функции вызывать метод imageUpdate()
    //  в процессе вывода
          g.drawImage(offScreenImage,0,0,null);
          showStatus("Done");
        }
        else
            showStatus("Loading image");
        }
    /* Каждый раз, когда апплет вызывает метод drawImage(), он создает
поток, вызывающий метод imageUpdate(), который можно
переопределить в классе апплета и использовать для того, чтобы
определить, какая часть изображения загружена в память.*/

public boolean imageUpdate(Image img, int infoflags,int x, int y,int w, int h)
{
    if(infoflags==ALLBITS)   {
       picLoaded=true;   // изображение загружено полностью
       repaint();              // перерисовать окно апплета
       return false;           // больше метод imageUpdate не вызывать
    }
    return true;               // изображение загружено не полностью
    }
}
```

События и их обработка

Событие - это информация, сгенерированная в ответ на некоторые действия пользователя (перемещение мыши, нажатие клавиши мыши или клавиши на клавиатуре). События также могут быть сгенерированы в ответ на изменение среды - к примеру, когда окно апплета заслоняется другим окном.

В современной технологии Java используется так называемое делегирование событий.

Апплет, обрабатывающий события. Создадим апплет Event, отображающий строку текста в месте положения щелчка мыши. Кроме того, добавим перемещение строки текста с помощью клавиш управления курсором.

Пример 1.8. Обработка событий от мыши и клавиатуры

```
import javax.swing.event.MouseInputAdapter;
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class Event extends Applet  {
int x=0, y=0;
public Event()  {
   // обработчик события от мыши
     MouseInputAdapter pm;
     pm = new MouseInputAdapter() {
      public void mousePressed(MouseEvent e) {
           x=e.getX(); y=e.getY();
           System.out.println(x);
           repaint();
      }};
      this.addMouseListener(pm);
   // обработчик события от клавиатуры
   KeyAdapter pk;
   pk = new KeyAdapter(){
   public void keyPressed(KeyEvent e) {
           System.out.println(e);
           int keyCode = e.getKeyCode();
           switch(keyCode)          {
            case KeyEvent.VK_DOWN:
                    y = y + 5; repaint(); break;
             case KeyEvent.VK_UP:
                    y = y - 5; repaint(); break;
             case KeyEvent.VK_RIGHT:
                    x = x + 5; repaint(); break;
```

```
            case KeyEvent.VK_LEFT:
                x = x - 5; repaint(); break;
            }
          }
        };
        this.addKeyListener(pk);
    }
    public void init()   {   }
    public void paint (Graphics g)   {
        g.drawString("Applet with Events",x, y);
    }
    }
```

Для обработки событий от кнопок мыши используется тип MouseListener. Это интерфейс. Для обработки этих событий надо, чтобы класс реализовывал указанный интерфейс, то есть в классе должны быть реализованы все методы этого интерфейса. Обработка событий от нажатия на клавиши клавиатуры выполнено также, как и от мыши, но используется другой адаптер - KeyAdapter, интерфейс (KeyListener) и метод (Key-Pressed).

Апплеты двойного назначения

Апплет двойного назначения - это программа, которая может работать и под управлением браузера, и автономно, как самостоятельное приложение. Создать апплеты двойного назначения достаточно легко. Следует лишь ввести оба метода main() и init() в одну и ту же программу, при этом выполняя в методе main() некоторые специфические действия.

Прежде всего в методе main() необходимо создать рамку окна, в котором будет отображаться вся графика (объект класса **Frame).** Для этого объекта обязательно надо переопределить обработку события, связанного с закрытием окна-рамки, так как по умолчанию окно не закрывается при нажатии на кнопку закрытия.

С помощью экземпляра апплета можно вызвать методы init**()** и start**()**, запуская апплет в методе main() так, как обычно это делает браузер. А затем апплет просто вставляется в окно-рамку.

Передавать приложению параметры можно в командной строке. Т.е. передача параметров апплету двойного назначения должна дублироваться при помощи командной строки и при помощи тега <PARAM> HTML-файла.

Пример 1. 9. Апплет двойного назначения

```
import javax.swing.event.MouseInputAdapter;
import java.applet.Applet;
import java.awt.event.*;
import java.awt.*;

public class Combi extends Applet  {
int x=10, y=20;
public Combi(){
   // обработчики событий от мыши и клавиатуры
}
public void init()   {   }
public void paint (Graphics g)   {
     g.drawString("Applet with Events",x, y);
}
public static void main(String args[])     {
     Frame fr = new Frame("Апплет двойного назначения");
     Combi c  =    new Combi();
     c.init();
     fr.add(c);
     fr.setSize(400,300);
     fr.setVisible(true);
      // обработка события закрытие окна-рамки
     fr.addWindowListener(new WindowAdapter(){
          public void windowClosing(WindowEvent e)   {
             System.exit(0);
          } });
    }
}
```

Модифицируйте метод paint() так, чтобы в окне апплета выводился режим работы программы: “Application” или “Applet".

Вместо класса Frame возможно использовать класс **JFrame** из библиотеки **Swing**. Возможно использование метода этого класса **setDefaultCloseOperation**(int operation) для определения события, связанного с закрытием окна. Вместо класса Applet возможно использование класса **JApplet** из того же пакета.

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

public class FrameUse {
   public static void main(String[] args) {
      JFrame frame = new JFrame ("Пример");
      int width = 400;
      int height = 300;
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setSize(width,height);
      frame.setVisible(true);
   }
}
```

Практические задания

1. Изучить основные понятия и термины Java.
2. Проверить и объяснить работу всех приложений, рассматриваемых в данной работе. Должны быть созданы следующие приложения NewClass (пример 1), TestElements (пример 2), TestModificators (пример 3).
3. Проверить и объяснить работу всех апплетов, рассматриваемых в данной работе. Должны быть созданы следующие апплеты: Hello (пример 4), AppletWithParam (пример 5), ParamUrlImage (пример 6), QuickPicture (пример 7), Event (пример 8), Combi (пример 9).
4. Разработать апплет двойного назначения с использованием классов из библиотеки Swing и AWT. Примерная формулировка задания предполагает:

- создание апплета двойного назначения с отрисовкой изображения в точке щелчка мыши,
- перемещение изображения по клавишам от клавиатуры,
- ввод через параметры апплета или параметры командной строки имени файла с изображением,
- использование двойной буферизации при отрисовки изображения.

Вопросы для самопроверки

1. Чем отличаются Java-приложения и Java-апплеты?
2. Что такое первичный класс приложения? Какой обязательный метод он должен содержать?
3. Какие существуют виды переменных Java, чем они отличаются друг от друга?
4. Какие примитивные типы определены в Java?
5. Что делает конструктор класса? Должен ли он обязательно явно присутствовать в объявлении класса?
6. Какие существуют виды ссылочных типов? Как реализуются ссылочные переменные?
7. Что такое типы, определенные пользователем?
8. В чем особенности строковых переменных?
9. Чем массивы Java отличаются от массивов других языков, их преимущества?
10. Как переменные различных видов передаются в качестве параметров методам?
11. Что такое элементы класса и элементы экземпляра класса, чем они отличаются друг от друга? Как нужно указывать, что переменная или метод является элементом класса, а не экземпляра?
12. Для чего используются модификаторы доступа? Какие существуют модификаторы доступа, как они ограничивают доступ к элементам?
13. Что позволяет делать процесс наследования? Что такое суперкласс и подкласс?
14. Что такое повторное использование кода?

15. Какие заранее определенные переменные содержит каждый класс Java?
16. Что можно сделать при помощи переменной this?
17. Что можно сделать при помощи переменной super?
18. Что такое скрытие переменной, затемнение переменной и замещение метода?
19. Как импортировать классы из пакетов?
20. Как добавить класс в пакет?
21. Чем выполнение апплета отличается от выполнения простого Java-приложения?
22. Чем отличаются первичные классы приложения и апплета?
23. Какие методы должен переопределять класс апплета?
24. Каковы принципы функционирования апплета?
25. Как передаются параметры апплету?
26. Как загрузить графическое изображение в апплет, приложение?
27. Как ускорить вывод графических изображений, загружаемых из файла и устранить мерцание при выводе изображений?
28. Что такое апплеты двойного назначения? Как они работают?

ГЛАВА 2. ОБРАБОТКА СОБЫТИЙ. ГРАФИКА.

Графика в Java

Графические операции всегда выполняются над объектом Graphics (контекст отображения). Например, в апплетах для вывода в окно используется метод paint(), которому передается параметр - объект класса Graphics.

Некоторые методы класса Graphics:

clearRect - очищает указанный прямоугольник, заполняя цветом фона;

clipRect - задает область ограничения вывода;

copyArea - копирует область экрана;

create - создает новый объект, который является копией исходного объекта;
draw3DRect - рисует прямоугольник с объемным эффектом;
drawArc - рисует дугу текущим цветом;
drawBytes - рисует указанные байты текущим шрифтом и цветом;
drawChars - рисует указанные символы текущим шрифтом и цветом;
drawImage - рисует указанное изображение типа Image;
drawLine - рисует линию между точками;
drawOval - рисует овал внутри указанного прямоугольника текущим цветом;
drawPolygon - рисует многоугольник текущим цветом;
drawRect - Рисует контур прямоугольника текущим цветом;
drawRoundRect - Рисует контур прямоугольника с закругленными краями;
drawString - Рисует указанную строку текущим шрифтом и текущим цветом;
fill3DRect - раскрашивает цветом прямоугольник с объемным эффектом;
fillArc - заполняет дугу текущим цветом;
fillOval - заполняет овал текущим цветом;
fillPolygon - заполняет многоугольник текущим цветом;
fillPolygon - заполняет объект класса Polygon текущим цветом;
fillRect - заполняет прямоугольник текущим цветом;
fillRoundRect - заполняет прямоугольник с закругленными краями;
setPaintMode - устанавливает режим заполнения текущим цветом.

Цвет. Для задания текущего цвета используется метод setColor() класса Graphics. Создадим случайный цвет и установим его, g - объект Graphics:

```
g.setColor( new Color((float)Math.random(), (float)Math.random(),
 (float)Math.random()));
```

Цветовая модель языка Java представляет собой 24-разрядную модель RGB (красный, синий, зеленый), следовательно объекты класса **Color** могут содержать 24 разряда цветовой информации (что соответствует 16 миллионам различных цветов).

Для использования цвета необходимо сначала создать объект Color и передать в него значения красного, зеленого и синего цвета (существуют два конструктора - для задания целочисленных значений (каждое значение от 0 до 255) и значений с плавающей точкой (каждое значение от 0.0 до 1.0)). Совместно эти значения и определяют цвет.

```
Color clr1=new Color(255,0,0);          // создать красный цвет
Color clr2=new Color(255,255,255);      // создать белый цвет
Color clr3=new Color(0,0,0);            // создать черный цвет
```

Можно использовать заранее определенные цвета Color.white, Color.lightGray, Color.gray, Color.darkGray, Color.black, Color.red, Color.pink, Color.orange, Color.yellow, Color.green, Color.magenta, Color.cyan, Color.blue. При использовании этих цветов не нужно создавать новый объект Color, можно записать следующее:

```
g.setColor(Color.red);
```

Помимо установки цвета отображения текста и графики методом setColor(), можно установить цвет фона и цвет переднего плана методами **setBackground()** и **setForeground()** класса Component.

Шрифты. Класс Graphics позволяет размещать на экране текст с использованием установленного шрифта. Для задания шрифта необходимо создать объект класса Font с параметрами: название гарнитуры шрифта (тип String), стиль шрифта (тип int) и размер шрифта в пунктах (тип int):

```
Font f=new Font("Times Roman",Font.BOLD,72);
```

Хотя можно выбрать любое начертание (гарнитуру) шрифта, рекомендуется использовать достаточно ограниченный набор шрифтов ("Times Roman", "Courier", "Helvetica"), которые имеются на всех системах.

Для задания стиля шрифта используются константы класса Font: Font.PLAIN, Font.BOLD и Font.ITALIC. Эти стили можно объединять при помощи операции +. После создания шрифта его можно установить для использования при помощи метода setFont() класса Graphics:

```
g.setFont(new Font("Times Roman",Font.BOLD,72));
```

Некоторые методы класса Font:

getFamily - получает название шрифта, зависящее от платформы;
getName - получает логическое имя шрифта;
getStyle - получает стиль шрифта;
getSize - получает размер шрифта в пунктах;
isPlain - возвращает true, если шрифт простой;
isBold - возвращает true, если шрифт полужирный;
isItalic - возвращает true, если шрифт курсивный.
getFont - получает шрифт из списка параметров системы

Если выбрать шрифт, который не установлен на конкретной машине, Java заменит его стандартным шрифтом (например, Courier). Для того, чтобы узнать какие шрифты доступны, можно воспользоваться методом **getAllFonts(),** определенным в классе **GraphicsEnvironment.** Ну, а если Вы захотите определить разрешение экрана и его размер, то можно воспользоваться методами класса **Toolkit**: **getScreenResolution()** и **getScreenSize()**.

Создадим апплет двойного назначения **FontsList**, в окне которого отображается список всех доступных апплету шрифтов.

Пример 2.1. Апплет двойного назначения FontsList

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

public class FontsList extends JApplet{
   Font f[];
   int count = 0, x = 10, y = 20;
   double w, h;
   int tek = 0;
```

```
  public FontsList()    {    }
   public void start()     {
       Dimension d =getSize();
       w = d.getWidth(); h = d.getHeight();
   }
  public void init()    {
       GraphicsEnvironment ge =
GraphicsEnvironment.getLocalGraphicsEnvironment();
       f = ge.getAllFonts();
       MouseListener ml = new MouseAdapter(){
         public void mouseClicked(MouseEvent e) {
             repaint();
         }
        };
       addMouseListener(ml);
       resize(800, 300);
  }
  public void paint(Graphics g)    {
     g.clearRect(0, 0,(int)w,(int)h);
     int i;
     for ( i = tek; i < f.length; i++)       {
        String s="";
        s +=f[i].getFontName();           s +=" ";
        s +=f[i].getFamily();          s +=" ";
        s +=f[i].getSize();          s +=" ";
        s +=f[i].toString();
        g.drawString(s,x,y);
        y = y + 20;
        if( y >= h)       { y = 20; break;  }    // переход на начало
      }
      if (i==f.length)  {   i = 0;  y = 20; }    // на  начало списка
     tek = i;
  }
  public static void main(String[] args) {
     JFrame frame = new JFrame ("Пример");
     FontsList appl = new FontsList();
     appl.init();     appl.start();
     frame.getContentPane().add(appl);
     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     frame.setSize(800, 300);
```

```
        frame.setVisible(true);
    }
}
```

Если мы хотим изменить цвет шрифта и вид его, то можно добавить код в метод paint:

```
g.setFont(new Font(f[i].getFontName(),Font.PLAIN,14));
g.setColor(new Color((float)Math.random(), (float)Math.random(),
(float)Math.random()));
```

Модель делегирования событий в Java 1.1

Модель событий, применяемая в Java 1.1, подходит для использования в AWT и в Swing. В этой модели каждое событие - это класс, наследуемый от класса java.util.EventObject. Все AWT события наследуются от класса java.awt.AWTEvent. Для удобства различные типы событий AWT помещены в отдельный пакет java.awt.event.

Модель событий базируется на концепции "слушаю события". Объект, ожидающий какое-либо событие, "слушает" его, является event listener. Объект, вырабатывающий событие, (источник события) поддерживает список объектов, ожидающих событие, и оповещает все объекты в списке о его появлении. Источник события имеет методы для добавления объектов, ожидающих события, и методы для удаления таких объектов.

Источник события при возникновении события запускает метод – обработчик события и передает в него объект типа EventObject или производного от него. Для того чтобы запустить нужный метод все объекты типа listener должны реализовывать определенный интерфейс. Интерфейс может определять несколько методов. Например, класс MouseEvent представляет несколько событий: нажатие кнопки, отпускание кнопки, и другие. В таблице 2.1 приводятся тип объекта, интерфейс для его обработки и методы, определяемые в каждом интерфейсе.

Для каждого из интерфейсов, содержащих более одного метода, определен класс-адаптер, который содержит пустые тела

методов. Классы-адаптеры имеют имена, такие же, как и имена интерфейсов с заменой Listener на Adapter: например, MouseAdapter, WindowAdapter.

Если Вы реализовали интерфейс или создали класс адаптера, нужно создать объект для того, чтобы он "слушал" событие. Затем зарегистрировать этот объект в источнике события. В AWT источник события всегда компонента: кнопка, список, и т.д. Регистрация делается с помощью методов с именами вида **addXXXListener**. Удаление объекта из списка слушающих выполняется с помощью **removeXXXListener**. Здесь XXX - имя типа события, генерируемого источником.

Таблица 2.1.

Класс события, интерфейс и методы для обработки.

Event Class	**Listener Interface**	**Listener Methods**
ActionEvent	ActionListener	actionPerformed()
AdjustmentEvent	AdjustmentListener	adjustmentValueChanged()
ComponentEvent	ComponentListener	componentHidden()
		componentMoved()
		componentResized()
		componentShown()
ContainerEvent	ContainerListener	componentAdded()
		componentRemoved()
FocusEvent	FocusListener	focusGained()
		focusLost()
ItemEvent	ItemListener	itemStateChanged()
KeyEvent	KeyListener	keyPressed()
		keyReleased()
		keyTyped()
MouseEvent	MouseListener	mouseClicked()
		mouseEntered()
		mouseExited()
		mousePressed()
		mouseReleased()
	MouseMotionListener	mouseDragged()

		mouseMoved()
TextEvent	TextListener	textValueChanged()
WindowEvent	WindowListener	windowActivated()
		windowClosed()
		windowClosing()
		windowDeactivated()
		windowDeiconified()
		windowIconified()
		windowOpened()

Таблица 2.2.

Компоненты AWT и типы событий, которые они могут генерировать.

Компонент	**События, генерируемые компонентом**	**Значение**
Button	ActionEvent	Пользователь нажал на кнопку
Checkbox	ItemEvent	Пользователь щелкнул на элементе, изменив выбор
CheckboxMenuItem	ItemEvent	Пользователь выбрал пункт из меню
Choice	ItemEvent	Пользователь щелкнул на элементе, изменив выбор
Component	ComponentEvent	Компонент передвинут, изменил размеры, стал видимым или невидимым.
	FocusEvent	Компонент получил или потерял фокус
	KeyEvent	Пользователь нажал или отпустил клавишу
	MouseEvent	Пользователь щелкнул кнопкой мыши, мышь попала в пределы компонента или ушла за его границы, или пользователь тянет объект. MouseEvent имеет два интерфейса

		прослушивания, MouseListener и MouseMotionListener.
Container	ContainerEvent	Компонент добавлен или удален из контейнера
List	ActionEvent	Пользователь дважды щелкнул на элементе списка
	ItemEvent	Пользователь выбрал пункт из списка
MenuItem	ActionEvent	Пользователь выбрал пункт меню
Scrollbar	AdjustmentEvent	Пользователь передвинул движок
TextComponent	TextEvent	Пользователь изменил текст
TextField	ActionEvent	Пользователь закончил редактирование текста
Window	WindowEvent	Окно открыто, или закрыто. или минимизировано, или максимизировано

1. Пример обработки события с использованием реализации интерфейса.

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

public class SimpleClick extends Applet  implements MouseListener {
    StringBuffer buffer;
    public void init() {
        addMouseListener(this);
        buffer = new StringBuffer();
        addItem("initializing... ");
    }
    public void start() {       addItem("starting... ");    }
    public void stop() {       addItem("stopping... ");    }
    public void destroy() {   addItem("preparing for unloading..."); }
    void addItem(String newWord) {
       System.out.println(newWord);
       buffer.append(newWord);
       repaint();
```

```
        }
        public void paint(Graphics g) {
            g.drawRect(0, 0,    getSize().width - 1,   getSize().height - 1);
            g.drawString(buffer.toString(), 5, 15);
        }
// Остальные методы интерфейса реализуются как пустые методы
        public void mouseEntered(MouseEvent event) {    }
        public void mouseExited(MouseEvent event) {    }
        public void mousePressed(MouseEvent event) {    }
        public void mouseReleased(MouseEvent event) {    }
        public void mouseClicked(MouseEvent event) {
              addItem("click! ");
         }
    }
```

2. Пример обработки события с использованием вложенных классов, наследующих классы-адаптеры.

```
    public class Scribble2 extends Applet {
      int last_x, last_y;
      public void init() {
        this.addMouseListener(new MyMouseAdapter());
        this.addMouseMotionListener(new MyMouseMotionAdapter());
        Button b = new Button("Clear");
        b.addActionListener(new MyActionEventListener());
        this.add(b);
      }
    class MyMouseAdapter extends MouseAdapter {
         public void mousePressed(MouseEvent e) {
              last_x = e.getX();    last_y = e.getY();
          }
    }
    class  MyMouseMotionAdapter extends MouseMotionAdapter   {
         public void mouseDragged(MouseEvent e) {
              Graphics g = getGraphics();
               int x = e.getX(), y = e.getY();
              g.setColor(Color.black);
              g.drawLine(last_x, last_y, x, y);
              last_x = x; last_y = y;
             }
```

```
}
class MyActionEventListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        Graphics g = getGraphics();
        g.setColor(getBackground());
        g.fillRect(0, 0, getSize().width, getSize().height);
    }
  }
}
```

3. Пример обработки события с использованием вложенного анонимного класса.

```
public class Scribble3 extends Applet {
  int last_x, last_y;
  public void init() {
      this.addMouseListener(new MouseAdapter() {
          public void mousePressed(MouseEvent e) {
            last_x = e.getX();       last_y = e.getY();
          }
      });
      this.addMouseMotionListener(new MouseMotionAdapter() {
          public void mouseDragged(MouseEvent e) {
              Graphics g = getGraphics();
              int x = e.getX(), y = e.getY();
              g.setColor(Color.black);
              g.drawLine(last_x, last_y, x, y);
              last_x = x; last_y = y;
          }
       });
       Button b = new Button("Clear");
       b.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e) {
              Graphics g = getGraphics();
              g.setColor(getBackground());
              g.fillRect(0, 0, getSize().width, getSize().height);
          }
      });
      this.add(b);
   }
```

```
}
```

Рассмотрим апплет двойного назначения LinesDraw, в окне которого можно рисовать прямые линии при помощи мыши. Для того чтобы нарисовать в окне апплета линию, пользователь должен установить курсор в начальную точку, нажать клавишу мыши и затем, не отпуская ее, переместить курсор в конечную точку. После отпускания клавиши координаты линии должны сохраняться апплетом в массиве, после чего будет происходить перерисовка апплета. Для того чтобы стереть содержимое окна апплета, необходимо сделать двойной щелчок в окне. При этом из массива координат должны будут удалены все элементы.

Пример 2.2. Рисование в окне

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Vector;

public class DrawLines extends JApplet  {
   int xDown, yDown; // координаты нажатия клавиши
   int xPrev, yPrev;     // предыдущие координаты конца линии
   Vector lines;                 // массив координат линий
   public void init()   {
      lines = new Vector();
          MouseListener ml = new MouseAdapter(){
          public void mousePressed(MouseEvent e) {
             xPrev = e.getX();    yPrev = e.getY();
         }
         public void mouseReleased(MouseEvent e) {
             xDown = e.getX();      yDown = e.getY();
             int k = e.getClickCount();
             System.out.println(k);
             if ( k < 2)    {
               Rectangle p=new Rectangle(xPrev, yPrev, xDown-xPrev,
yDown-yPrev);
               lines.addElement(p);
       }
       else       lines.removeAllElements();
```

```
        repaint();        }};
        addMouseListener(ml);
      }
      public void paint(Graphics g)   {
        Dimension appSize = getSize();
        g.setColor(Color.yellow);
        g.fillRect(0,0,appSize.width, appSize.height);
        g.setColor(Color.black);
       int size = lines.size();
       for ( int i = 0; i < size; i++ )      {
          Rectangle p=(Rectangle)lines.elementAt(i);
          g.drawLine(p.x, p.y, p.x+p.width, p.y+p.height);
       }
    }
    public static void main(String[] args) {
       JFrame frame = new JFrame ("Пример");
       DrawLines appl = new   DrawLines();
       appl.init();      appl.start();
       frame.getContentPane().add(appl);
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       frame.setSize(800, 300);
       frame.setVisible(true);
    }
}
```

А теперь попробуем выполнить обработку событий при движении мыши. Просто добавим в метод init код:

```
MouseMotionListener mm = new MouseMotionAdapter()      {
    public void mouseDragged(MouseEvent e)        {
        xDown = e.getX();     yDown = e.getY();
Rectangle p=new Rectangle(xPrev,yPrev,xDown-xPrev, yDown-yPrev);
        lines.addElement(p);
        xPrev = xDown;    yPrev = yDown;
       repaint();
     }
    };
    addMouseMotionListener(mm);
```

Рассмотрим апплет двойного назначения KeyCodes, в окне которого отображаются символы, соответствующие нажимаемым клавишам, код соответствующей клавиши и коды модификации. В процессе отображения новые строки должны появляться в верхней части окна, а старые сдвигаться вниз после отпускания клавиши (должна быть организована свертка в окне).

Пример 2.3. Приложение KeyCodes.

```
public class KeyCodes extends JApplet{
   int yHeight;                // высота символов шрифта,
   Dimension appSize; // текущий размер окна апплета
     public void init()      {
        KeyListener mm = new KeyAdapter()    {
          public void keyPressed(KeyEvent e)   {
            Graphics g = getGraphics();
            FontMetrics fm =  g.getFontMetrics();
            yHeight = fm.getHeight();
String s = "Char - " + e.getKeyChar() + ", Key  - " + e.getKeyCode()
+ ", Modifiers - " + e.getModifiers();
            g.drawString(s, 10, yHeight);
          }
           public void keyReleased(KeyEvent e)              {
             Graphics g = getGraphics();
             g.copyArea(0,1,appSize.width-1,appSize.height-yHeight-
5,0,yHeight+1);
             g.setColor(Color.YELLOW);
             g.fillRect(1,1,appSize.width-2,yHeight+1);
           }
         };
         addKeyListener(mm);
      }

     public void paint(Graphics g)      {
       appSize = getSize();
       g.setColor(Color.YELLOW);
       g.fillRect(1,1,appSize.width-1,appSize.height-1);
      }

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

```
        JFrame frame = new JFrame ("Пример");
        KeyCodes appl = new    KeyCodes();
        appl.init();
        frame.getContentPane().add(appl);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(800, 300);
        frame.setVisible(true);
    }
}
```

Таймер

В данном приложении реализуется пример выполнения действия по таймеру. Для реализации таймера в Java существует класс **Timer** в пакете **java.util**. Основные методы таймера: schedule и cancel, которые запускают и останавливают выполнение таймера. Кроме того, для таймера необходимо подготовить задание – класс наследующий от класса **TimerTask** (класс Updater в примере). В этом классе необходимо переопределить метод public void run(), который вызывается, когда таймер делает свой очередной отсчет.

Пример 2. 4. Приложение Timer.

```
import java.util.Timer;

public class MainApplet extends JApplet  {
   private Timer m_timer = new Timer();
   private boolean m_runViaFrame = false;
   private double m_time = 0;

   private class Updater extends TimerTask     {
      private MainApplet m_applet = null;
      // Первый ли запуск метода run()?
      private boolean m_firstRun = true;
      // Время начала
      private long m_startTime = 0;
      // Время последнего обновления
      private long m_lastTime = 0;
```

```
    public Updater(MainApplet applet)        {
        m_applet = applet;
    }
    @Override
    public void run()    {
        if (m_firstRun)      {
            m_startTime = System.currentTimeMillis();
            m_lastTime = m_startTime;
            m_firstRun = false;
        }
        long currentTime = System.currentTimeMillis();
        // Время, прошедшее от начала, в секундах
        double elapsed = (currentTime - m_startTime) / 1000.0;
        // Время, прошедшее с последнего обновления, в секундах
        double frameTime = (m_lastTime - m_startTime) / 1000.0;
        // Вызываем обновление
        m_applet.Update(elapsed, frameTime);
        m_lastTime = currentTime;
    }
}
public MainApplet()    {    Init();   }
public MainApplet(boolean viaFrame)    {
    m_runViaFrame = viaFrame;
    Init();
}
private void Init()    {
    // таймер будет вызываться каждые 100 мс
    m_timer.schedule(new Updater(this), 0, 100);
}
public void Update(double elapsedTime, double frameTime)    {
    m_time = elapsedTime;
    this.repaint();
}
@Override
public void paint(Graphics g)     {
    g.setColor(Color.WHITE);
    g.fillRect(0, 0, this.getWidth(), this.getHeight());
    g.setColor(Color.BLACK);
    String str = "Time = " + Double.toString(m_time);
    g.drawString(str, 15, 15);
```

```
    }
    public static void main(String[] args)      {
        JFrame frame = new JFrame("Timer example");
        MainApplet applet = new MainApplet(true);
        frame.add(applet);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(800, 600);
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }
}
```

Графика 2D

На базе Java реализован пакет Java2D, который позволяет создавать мощную графику. Использование этого пакета возможно только в Java2-приложениях, построенных на основе Java Foundation Classes. Вместо типа Graphics здесь возможно использование типа Graphics2D и всех методов, предоставляемых классом Graphics2D.

Ниже приводен пример апплета двойного назначения, наследующего от Japplet, рисующий окружность с градиентной заливкой.

Пример 2.5. Использование Graphics2D.

```
import java.awt.geom.Ellipse2D;

public class DrawFig extends JApplet  {
  public void init() {  }
  public void paint(Graphics g)  {
     Graphics2D g2d = (Graphics2D)g;
     GradientPaint gradient =
   new GradientPaint(0,0,Color.red,175,175,Color.yellow,true);
     g2d.setPaint(gradient);
      Ellipse2D.Double circle =
      new Ellipse2D.Double(10,10,350,350);
     g2d.fill(circle);
     g2d.setPaint(Color.black);
     g2d.draw(circle);
  }
```

```
    public static void main(String[] args) {
        JFrame frame = new JFrame ("Пример");
        DrawFig appl = new   DrawFig();
        appl.init();       appl.start();
        frame.getContentPane().add(appl);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 400);
        frame.setVisible(true);
    }
}
```

Вероятностный подход

Вероятность события можно запрограммировать очень просто. Допустим, необходимо определить выполнится ли событие с вероятностью P (от 0.0 до 1.0). Для этого генерируем случайное число (float)Math.random(). Оно как раз будет лежать в диапазоне от 0.0 до 1.0. Если это число меньше либо равно P, то событие исполняется, в противном случае – нет. При такой схеме отдельно нужно учитывать только нижний предел (0.0). Если P = 0.0, то событие не исполняется никогда.

Практические задания

1. Изучить основные понятия и термины обработки событий по модели делегирования событий на Java.
2. Разобрать приведенные примеры **FontsList**, **LinesDraw, KeyCodes, Timer**, **DrawFig**.
3. Разработать программу. Основная задача – разработка упрощенной имитации поведения объектов (все последующие лабораторные работы будут расширять это задание). Объекты реализуются через наследование: абстрактный класс, интерфейс → наследники.

Рабочий цикл программы:

- запускается процесс симуляции, генерируются объекты классов согласно заданию;
- симуляция завершается, выводится статистическая информация.

Для решения задачи:

- Создать интерфейс IBehaviour, задающий поведение объекта;
- Создать иерархию классов, определяющие объекты по варианту и реализующие интерфейс IBehaviour.
- Создать класс Habitat, определяющий размер рабочей области и хранящий список объектов, с параметрами заданными вариантом. Предусмотреть в классе метод Update, вызывающийся по таймеру и получающий на вход время, прошедшее от начала симуляции. В данном методе должны генерироваться новые объекты и помещаться в поле визуализации в случайном месте. Визуализация объекта – схематично, плюсом будет, если объект будет похож на оригинал;

Рабочее окно программы – область визуализации среды обитания объектов;

4. Симуляция должна запускаться по клавише **B** и останавливаться по клавише **E**. При остановке симуляции список уничтожается. Время симуляции должно отображаться текстом в области визуализации и скрываться/показываться по клавише **T**;
5. По завершению симуляции в поле визуализации должна выводиться информация о количестве и типе сгенерированных объектов, а также время симуляции. Текст должен быть форматирован, т.е. выводиться с использованием разных шрифтов и цветов.
6. Параметры симуляции задаются в классе Habitat.

Вариант 1

Объект – муравей. Бывают 2 видов: рабочий и воин. Рабочие рождаются каждые N_1 секунд с вероятностью P_1. Воины рождаются каждые N_2 секунд с вероятностью P_2.

Вариант 2

Объект – пчела. Бывают 2 видов: трутень и рабочий. Трутни рождаются каждые N_1 секунд, если их количество менее K% от общего числа пчел, в противном случае – не рождаются вовсе. Рабочие рождаются каждые N_2 секунд с вероятностью P.

Вариант 3

Объект – аквариумная рыбка. Бывают 2 видов: золотая и гуппи. Золотые рыбки рождаются каждые N_1 секунд с вероятностью P_1. Гуппи рождаются каждые N_2 секунд с вероятностью P_2.

Вариант 4

Объект – кролик. Бывают 2 видов: обыкновенный и альбинос. Обыкновенные кролики рождаются каждые N_1 секунд с вероятностью P_1. Альбиносы рождаются каждые N_2 секунд, при условии, что их количество менее K% от общего числа кроликов, в противном случае – не рождаются вовсе.

Вариант 5

Список объектов продажи на автомобильном рынке состоит из 2-х видов машин: грузовые и легковые. Грузовые машины генерируются каждые N_1 секунд с вероятностью P_1. Легковые генерируются каждые N_2 секунд с вероятностью P_2.

Вариант 6

Рабочий коллектив компании состоит из разработчиков и менеджеров. Разработчики генерируются каждые N_1 секунд с вероятностью P_1. Менеджеры генерируются каждые N_2 секунд при условии, что их количество менее K% от общего числа разработчиков, в противном случае – не генерируются.

Вариант 7

Список жилых домов города состоит из двух типов: капитальный, деревянный. Капитальные дома генерируются каждые N_1 секунд с вероятностью P_1. Деревянные дома генерируются каждые N_2 секунд с вероятностью P_2.

Вариант 8

Список транспортных средств на дороге состоит из двух категорий: автомобили и мотоциклы. Автомобили генерируются каждые N_1 секунд с вероятностью P_1. Мотоциклы генерируются каждые N_2 секунд с вероятностью P_2.

Вариант 9

Объект обучающийся. Бывает 2 видов: студент (муж. пола) и студентка (жен. пола). Студенты генерируются каждые N_1 секунд с вероятностью P_1. Студентки генерируются каждые N_2 секунд с вероятностью P_2.

Вариант 10

Картотека налоговой инспекции состоит из записей двух типов: физические и юридические лица. Физические лица генерируются каждые N_1 секунд с вероятностью P_1. Юридические лица генерируются каждые N_2 секунд с вероятностью P_2.

Вопросы для самопроверки

1. Что такое апплет двойного назначения?
2. Над объектом какого типа выполняются все графические операции?
3. Почему рекомендуется использовать ограниченный набор цветов?
4. Почему рекомендуется использовать ограниченный набор шрифтов?
5. Как получить список всех доступных шрифтов?
6. Что такое событие? Когда возникают события?
7. Какой метод какого класса получает управление при возникновении события? Что передается ему в качестве параметра?
8. Какие методы отвечают за обработку простых событий от мыши?
9. Какие методы отвечают за обработку простых событий от клавиатуры?
10. Что такое модель делегирования событий?
11. Какие существуют способы задания обработчиков событий?
12. Как подключиться к прослушиванию события?
13. Что такое вложенные классы?
14. Что такое анонимные классы?
15. Как выполнять рисование не в методе paint?

ГЛАВА 3: РАЗРАБОТКА ГРАФИЧЕСКОГО

ИНТЕРФЕЙСА ПРОГРАММЫ

Библиотека графических компонент AWT

Для построения графического интерфейса GUI в Java имеются две отдельных, но тесно связанных между собой библиотеки графических классов: Abstract Windows Toolkit (AWT) и Java Foundation Classes (JFC), которая известна как Swing. Классы из библиотеки Swing (особенно те, что относятся к элементам управления) начинаются с символа J. Библиотека Swing более современная и предоставляет больше возможностей, чем AWT, к тому же элементы управления в ней более привлекательны визуально.

Компоненты Java. Все элементы управления основаны на классе Component. Всякий графический вывод на экран связан (прямо или косвенно) с компонентом. Например, в силу того, что апплеты фактически являются потомками класса Component, имеется возможность рисовать прямо в апплете, не прибегая для этого к специальным методам.

Класс Component содержит очень много методов для обеспечения работы с компонентом GUI.

Некоторые методы класса Component:
getParent - получает объект-владелец компонента;
isShowing - проверяет, является ли компонент видимым;
isEnabled - проверяет, является ли компонент разрешенным;
location - возвращает текущее положение компонента;
size - возвращает текущий размер компонента;
bounds - возвращает текущие границы компонента;
enable - разрешает компонент;
disable - запрещает компонент;
show - отображает компонент;
hide - скрывает компонент;
getForeground - получает текущий цвет переднего плана;
setForeground - устанавливает текущий цвет переднего плана;

getBackground - получает текущий цвет фона;
setBackground - устанавливает текущий цвет фона;
getFont - Получает текущий шрифт;
setFont - устанавливает текущий шрифт;
move - перемещает компонент в новое положение (в системе координат предка);
resize - изменяет ширину и высоту компонента;
getGraphics - получает графический контекст компонента;
paint - отвечает за рисование компонента;
update - отвечает за обновление компонента, вызывается методом repaint();
paintAll - отвечает за рисование компонента и его подкомпонентов;
repaint - вызывает перерисовку компонента. Этот метод вызывает метод update();
imageUpdate - отвечает за перерисовку компонента при изменении выводимого в компоненте изображения типа Image;
createImage - создает изображение;
prepareImage - подготавливает изображение типа Image для визуализации;
inside - проверяет, находится ли заданная точка “внутри” компонента;
locate - возвращает компонент, содержащий заданную точку;
mouseDown, mouseDrag, mouseUp, mouseMove, mouseEnter, mouseExit - отвечают за обработку соответствующих событий мыши;
keyUp, keyDown - отвечают за обработку соответствующих событий клавиатуры;
action - вызывается в том случае, если в компоненте происходит некоторое действие;
gotFocus - указывает на то, что компонент получил фокус ввода;
lostFocus - указывает на то, что компонент потерял фокус ввода;
requestFocus - запрашивает фокус ввод;
nextFocus - передает фокус следующему компоненту.

Этими методами могут пользоваться все наследуемые от класса Component классы GUI - все элементы управления и классы контейнеров.

Для того чтобы нарисовать изображение, вывести некоторый текст или разместить элемент пользовательского интерфейса на экране, должен использоваться контейнер класса Container или его подкласса. Контейнеры - это объекты, которые содержат компоненты. А компоненты - это объекты, которые наследуют от класса Component: кнопки, полосы прокрутки, другие элементы управления. Контейнеры сами являются подклассами Component, что подразумевает возможность их вложения или размещения внутри друг друга.

Так, например, класс Applet является подклассом Panel, который в свою очередь является подклассом Container, а тот - подклассом Component. Поэтому все апплеты умеют вводить и отображать компоненты. Нужно просто создать компонент и ввести его в апплет.

Элементы управления. При их помощи можно создавать программы, обладающие развитым пользовательским интерфейсом. Язык Java содержит все стандартные элементы управления, которые можно обнаружить в современных операционных системах. Элементы управления сначала создаются, затем добавляются в контейнер методом add().

Кнопки (класс **JButton**) - это элементы, которые дают возможность нажимать на них, чтобы проинициализировать некоторое действие. Для создания кнопок необходимо создать экземпляр объекта JButton, дать ему имя и добавить его в контейнер:

```
JButton btn = new JButton("Click me!");
add(btn);
```

или еще проще:

```
add(new JButton("Click me!"));
```

Следует обратить внимание на то, что при создании кнопки не передаются координаты X, Y ее положения. Это происходит потому, что элементы управления, размещаются на экране в соответствии с правилами, определяемыми менеджерами компоновки (размещения): при этом координаты не используются.

Создать кнопку можно с заданной меткой (строкой), а можно и без нее. Для этого предназначены соответствующие конструкторы **JButton(String label) и JButton()**. Для изменения и получения метки кнопки существуют методы **setText()** и **getText()** соответственно.

Обработка событий от кнопки в Java. При нажатии пользователем на кнопку генерируется событие ActionEvent. Для обработки этого события используется интерфейс ActionListener, а метод этого интерфейса – actionPerformed.

```
myButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e )    {
    ((JButton)e.getSource()).setBackground(java.awt.Color.red);
    ((JButton)e.getSource()).setText("Thanks!");
  }
});
```

Флажки (или переключатели). Язык Java поддерживает два типа флажков-переключателей: неисключающие (класс JCheckBox) и исключающие (класс JRadioButton).

Неисключающие флажки (переключатели с независимой фиксацией) могут быть выбраны независимо от состояния других флажков. Такие флажки являются самостоятельными объектами класса **JCheckBox**, для их создания используются конструкторы **JCheckBox ()** и **JCheckBox (String label)**.

Исключающие флажки (переключатели с зависимой фиксацией) позволяют одновременно выбрать только один элемент в группе флажков. Если выбирается другой элемент в группе, то выбор предыдущего флажка отменяется, а новый выбор выделяется. Для создания такой группы флажков сначала

создается объект класса **ButtonGroup**, а затем создаются объекты класса **JRadioButton**, входящие в эту группу, для чего используются конструкторы **JRadioButton (String label)** и **JRadioButton (String label, boolean state).**

```
// создание независимого флажка и добавление в контейнер
add(new JCheckBox("It is checkbox"));

// создание группы переключателей
ButtonGroup gr = new ButtonGroup();
// создание переключателей, добавление в группу и в контейнер
JRadioButton first, second, third;
first = new JRadioButton("1st radiobutton from group");
gr.add(first);   add(first);
// переключатель установлен
second = new JRadioButton("2nd radiobutton from group",true);
gr.add(second);   add(second);
third = new JRadioButton("3rd radiobutton from group");
gr.add(third);   add(third);
```

Класс ButtonGroup содержит методы для определения текущего выбора и установки нового выбора для флажков группы. Метод **getSelection()** возвращает ссылку на объект класса ButtonModel, содержащего информацию о выделенном переключателе. Метод **setSelected(ButtonModel model, boolean b)** устанавливает выделение. Объект класса ButtonModel можно получить из переключателя **JRadioButton** методом **getModel().**

При помощи методов класса **getText()** и **isSelected()** класса JRadioButton можно получить метку переключателя и текущее состояние переключателя, а при помощи методов **setText()** и **setSelected(boolean b)** изменить метку переключателя и установить состояние переключателя.

Обработка событий от флажка в Java. При изменении состояния переключателя (JCheckBox) генерируется событие **ItemEvent**. Интерфейс для обработки - **ItemListener**, в нем один метод - **itemStateChanged**().

```
myCheckbox.addItemListener(new ItemListener() {
  public void itemStateChanged(ItemEvent e )   {
    boolean i = ((JCheckBox)e.getItem()).isSelected();
    ((JCheckBox)e.getItem()).setBackground(i ?
    Color.red : Color.white);
  }
});
```

Комбобоксы (или выпадающие списки). Класс JComboBox дает возможность создавать список элементов выбора, который всплывает на экране в виде меню.

```
JComboBox cb = new JComboBox();
cb.addItem("1");     cb.addItem("2");
cb.addItem("3");     add(cb);
```

Рассмотрим некоторые методы класса JComboBox. Количество элементов выбора в выпадающем списке можно определить методом **getItemCount()**, добавляются элементы методом **addItem()**, а получить элемент по его порядковому номеру можно с помощью функции **getItemAt(int index)**.

Для того чтобы определить индекс выбранного элемента или сам выбранный элемент, имеются методы **getSelectedIndex()** и **getSelectedItem()**. Изменить выбор можно при помощи методов **setSelectedIndex(int index)** и **setSelectedItem(Object object)**.

Обработка событий от меню выбора в Java. При изменении выбора в выпадающем списке (JComboBox) генерируется событие **ItemEvent**. Интерфейс для обработки – **ItemListener**, также как при обработке событий от JCheckBox.

Списки. Для создания списка необходимо создать объект класса **JList** и ввести в него любые элементы. Затем необходимо этот объект добавить в контейнер.

```
JList list = new JList();
Vector<String> data = new Vector<String>();
```

```
data.add("1st item");
data.add("2nd item");
data.add("3rd item");
list.setListData(data);
add(list);
```

Множественный выбор устанавливается по умолчанию (элементы выделяются с зажатым Ctrl). Для того чтобы сделать единичный выбор, добавьте строчку.

```
list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

Так как список связан с коллекцией Vector<String>, то можно изменять список, изменяя коллекцию. Для добавления/удаления элементов используются методы коллекции **add(…)/remove(…)**.

Работа с выделенными элементами осуществляется через методы вида **getSelected*(…)** и **setSelected*(…)**.

Обработка событий от списка в Java. При выборе строки (или нескольких строк) в списке **JList** вырабатывается событие **ListSelectionEvent**. Для прослушивания этого события используется интерфейс **ListSelectionListener**.

```
list.addListSelectionListener(new ListSelectionListener(){
  public void valueChanged(ListSelectionEvent listSelectionEvent)  {
    JList l = (JList)listSelectionEvent.getSource();
    if (l.getSelectedIndex() == 0)      {
      // todo something
    }
    else if (l.getSelectedIndex() == 1)    {
      // todo something else
    }
  }
});
```

Метки-надписи (класс **Label**) - это простейшие устройства. Не порождают событий и используются для размещения на экране строк текста.

```
// для метки задается выравнивание по центру
add(new JLabel("Label 1!", JLabel.CENTER));
```

Для изменения установок меток служат методы: **setHorizontalAlignment(int)** - устанавливает режим выравнивания, **setText()** - устанавливает текст надписи. Для получения текущего режима выравнивания используется метод **getHorizontalAlignment()**, а для определения текста надписи - метод **getText().**

Текстовые компоненты. Текстовые компоненты рассмотрим из библиотеки AWT. Дело в том, что по какой-то причине разработчики языка лишили компоненты JTextField и JTextArea обработчиков событий на изменение текста. Кроме того, классы JTextField и TextField (JTextArea и TextArea) очень похожи вплоть до названий методов. Так что если не планируется обрабатывать ввод текста можно воспользоваться классами из библиотеки Swing. Следует также напомнить, что в одном интерфейсе можно использовать компоненты из библиотеки Swing и AWT одновременно.

Поля редактирования этого типа **TextArea** состоят из нескольких строк текста и имеют полосы прокрутки. Напротив, поля редактирования типа **TextField** состоят из одной строки и не имеют полос прокрутки. Оба этих класса являются наследниками класса **TextComponent** и аналогичны друг другу.

Вывести текст в поле редактирования или получить текст из поля можно методами **setText()** и **getText()** соответственно. Для выделения фрагмента текста применяется метод **select()** (для выделения всего текста - метод **selectAll()**), а для получения номеров первого и последнего выделенных символов - методы **getSelectionStart()** и **getSelectionEnd()**. Для получения выделенного фрагмента текста используется метод **getSelectedText()**. Запретить или разрешить редактирование можно при помощи метода **setEditable()**, а проверить, разрешено ли редактирование - методом **isEditable()**.

Например, создадим поле редактирования шириной 20 символов, инициализированной строкой "Enter text":

```
TextField tField=new TextField("Enter text",20);
add(tField);
```

Для получения информации о том, какова ширина текстового поля в символах, используется метод **getColums().**

Текстовые поля поддерживают ввод маскируемых символов (в Swing введен специальный класс **JPasswordField**), т.е. символов, ввод которых на экране отображается каким-либо одним символом (эхо-символом), а не фактически вводимыми символами. Для установки эхо-символа используется метод **setEchoCharacter()**, а для того, чтобы определить, какой символ используется в качестве эхо-символа, - метод **getEchoChar()**. Для проверки того, имеет ли поле эхо-символ, применяется метод **echoCharIsSet()**.

Обработка событий от текстовых полей в Java. При работе с текстовыми полями можно использовать события **ActionEvent** и **TextEvent**. Первое вырабатывается, когда пользователь нажал клавишу Enter, а второе - при изменении текста. Первое событие прослушивается **ActionListener**, а второе - **TextListener**. Оба интерфейса имеют по одному методу, поэтому механизм обработки событий прост. Для текстовой области используется событие **TextEvent**.

```
textField1.addTextListener(new java.awt.event.TextListener() {
    public void textValueChanged(TextEvent e) {
        button1.setLabel("Изменяем текст в поле");
    }
});
textField1.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(ActionEvent e) {
        button1.setLabel("Изменили текст в поле");
    }
});
```

```
TextArea textArea1 = new TextArea();
textArea1.setText("textArea1");
textArea1.addTextListener(new java.awt.event.TextListener() {
     public void textValueChanged(TextEvent e)  {
          button1.setLabel("Изменяем текст в области");
     }
});
```

Контейнеры

Контейнеры - это объекты (компоненты), позволяющие помещать в себя другие различные компоненты. Класс контейнеров **Container** - подкласс класса Component. Существует два вида основных вида контейнеров: панели и окна.

В случаях сложного интерфейса контейнеры позволяю объединять элементы управления в смысловые группы и по-разному размещать эти группы элементов относительно друг друга. Так, например, в окне апплета можно создать несколько панелей, разделяющих его на части. Отдельные панели могут содержать в себе такие компоненты, как кнопки, переключатели и другие компоненты.

По умолчанию каждый контейнер имеет ассоциированный с ним менеджер компоновки и предназначены они для визуальной организации элементов интерфейса.

Класс Container имеет методы, при помощи которых происходит управление введенными в него компонентами, установка менеджеров размещения и др.

Некоторые методы класса Container:

countComponents - возвращает число содержащихся в контейнере компонент;
getComponent - возвращает компонент контейнера;
getComponents - возвращает все компоненты контейнера;
add - добавляет компонент в контейнер;
remove - удаляет компонент из контейнера;
removeAll - удаляет все компоненты из контейнера;

getLayout - указывает менеджер компоновки данного контейнера;
setLayout - устанавливает менеджер компоновки данного контейнера;
layout - выполняет размещение компонент внутри контейнера;
minimumSize - возвращает минимальный размер контейнера;
paintComponents - отображает компоненты контейнера;
locate - возвращает компонент, содержащий заданную точку.

Панель (класс **JPanel**) является наиболее общим видом контейнеров в Java. Панель можно использовать как внутри другого контейнера (например, фрейма или апплета), так и непосредственно в окне Web-браузера**.** Панель может иметь свой собственный менеджер размещения (по умолчанию **FlowLayout**), независимый от менеджера размещения контейнера, в который эта панель входит.

Для создания панели, прежде всего, необходимо выбрать для окна апплета (или другого контейнера) схему размещения, соответствующую необходимому расположению панелей. Например, для создания в окне апплета двух панелей, разделяющих его по горизонтали, следует выбрать режим **GridLayout**, используя метод **setLayout()** контейнера, в данном случае апплета:

```
setLayout(new GridLayout(2,1));
```

При использовании такого менеджера размещения панели будут размещаться в ячейках таблицы, состоящей из одного столбца и двух строк. Далее нужно создать объекты класса JPanel:

```
JPanel top, bottom;          // объявление панелей
top = new JPanel();          // создание верхней панели
bottom = new JPanel();       // создание нижней панели
```

Для добавление любых компонентов (и панелей тоже) в любой контейнер используется метод add**()** контейнера:

```
add(top);                    // ввод верхней панели в контейнер
```

```
add(bottom);  // ввод нижней панели в контейнер
```

Можно добавлять панели в панели, применяя к ним аналогичные действия. Например, добавим в верхнюю панель две панели:

```
top.setLayout(new GridLayout(1,2));
JPanel left, right;                    // объявление панелей
left = new JPanel();                   // создание левой панели
right = new JPanel();   // создание правой панели
top.add(left);  // ввод левой панели в панель top
top.add(right);             // ввод правой панели в панель top
```

Для добавления компонентов в панели необходимо указывать, в какую панель вводится тот или иной компонент, например:

```
JButton btn1, btn2;                    // объявление кнопок
btn1 = new JButton("Yes"); // создание первой кнопки
btn2 = new JButton("Cancel"); // создание второй кнопки
bottom.add(btn1); // ввод первой кнопки в панель bottom
bottom.add(btn2); // ввод второй кнопки в панель bottom
```

Окна (класс **Window**), как и панели, являются общим классом контейнеров. Но в отличие от панелей окно Java представляет собой **объект операционной системы,** существующий отдельно от окна браузера или программы просмотра апплетов.

Непосредственно класс Window в большинстве случаев использовать бессмысленно, нужно использовать его подклассы **JFrame** и **JDialog**. Каждый из этих подклассов содержит все функции исходного класса Window, добавляя к ним несколько специфических свойств.

Некоторые методы класса Window:

show - отображает окно. Окно появится на переднем плане, если оно было видимым до этого;

dispose - удаляет окно. Этот метод необходимо вызвать для освобождения ресурсов, занимаемых окном;

toFront - переносит рамку на передний план окна;

toBack - переносит рамку на задний план окна.

При создании окна обязательным параметром конструктора Window**()** является объект класса **Frame** (либо его наследники, например, класс **JFrame**). Этот объект можно создать непосредственно при вызове конструктора окна. Так как окна изначально создаются невидимыми, отобразить их на экране можно, лишь вызвав метод **setVisible(true)**. Правда, перед этим иногда полезно придать окну нужные размеры, и разместить окно в нужной позиции.

```
Window win = new Window(new JFrame()); // создание окна
win.setSize(200,300);  // изменение его размеров
win.setLocation(50,50);          // перемещение окна
win.setVisible(true);    // отображение окна
```

Если окно видимо, то его можно сделать невидимым при помощи метода **setVisible(false)**. Этот метод не удаляет объект окна, просто он делает окно невидимым.

Вместо класса Window, можно использовать класс **JWindow**, результат будет аналогичным. Но имейте в виду, классы **JFrame** и **JDialog** не наследуются от класса **JWindow**.

Фрейм (класс **JFrame**) - это объект, который может существовать без всякой связи с окном браузера. С помощью класса **JFrame** можно организовать интерфейс независимого приложения.

Окно, созданное на базе класса **JFrame**, больше похоже на главное окно обычного приложения Windows. Оно автоматически может иметь главное меню, для него можно устанавливать форму курсора и пиктограмму. Внутри такого окна можно рисовать. Так как окно класса **JFrame** произошло от класса Container, то в него можно добавлять различные компоненты и панели. Нужно отметить, что по умолчанию для окон класса **JFrame** устанавливается режим размещения **BorderLayout**.

Некоторые методы класса JFrame:

getTitle - возвращает заголовок;
setTitle - устанавливает заголовок;
getIconImage - возвращает пиктограмму;

setIconImage - устанавливает пиктограмму;
getMenuBar - возвращает ссылку на объект меню класса MenuBar;
setMenuBar - устанавливает меню;
remove - удаляет компонент из фрейма;
dispose - удаляет фрейм. Этот метод необходимо вызвать для освобождения ресурсов, занимаемых фреймом;
isResizable - проверяет, может ли пользователь изменять размер фрейма;
setResizable - устанавливает флаг разрешения изменения фрейма;
setCursor - устанавливает вид курсора с использованием констант класса Frame;
getCursorType - возвращает тип курсора.

```
// объявление нового класса фрейма
class MainFrameWnd extends JFrame   {
    public MainFrameWnd(String str)  {
          // обязательный вызов конструктора суперкласса
          super(str);
          setSize(400,200);
          // здесь можно определить различные параметры фрейма,
          // например, форму курсора, пиктограмму, задать меню и др.
   }
```

После объявления класса можно создавать объекты этого класса, вызывая для отображения окна метод **setVisible()**:

```
MainFrameWnd fr = new MainFrameWnd("Title for frame");
fr.setVisible(true);
```

Чтобы правильно обработать закрытие фрейма нужно использовать следующий код.

```
fr.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Меню являются стандартным элементом интерфейса современных программ. Для того, что создать и добавить меню в контейнер, необходимо сначала создать панель меню (объект класса **JMenuBar**), затем добавить в нее меню (объект класса

JMenu). В выпадающие меню класса JMenu также можно добавлять элементы (строки или объекты класса **JMenuItem**). Подготовленную панель меню теперь можно ввести в контейнер его методом **setJMenuBar(…)**.

```
JMenuBar mainMenu = new JMenuBar();
JMenu fileMenu = new JMenu("File");     // создание меню "File"
JMenu helpMenu = new JMenu("Help"); // создание меню "Help"
```

После создания отдельных меню в них можно добавить элементы. Для этого нужно вызвать метод add**(…)** класса JMenu:

```
// строка "New" меню "File"
fileMenu.add(new JMenuItem("New"));
fileMenu.addSeparator();          // горизонтальный разделитель
// строка "Exit" меню "File"
fileMenu.add(new JMenuItem("Exit"));
// строка "Content" меню "Help"
helpMenu.add(new JMenuItem("Content"));
```

Сформированные отдельные меню добавляются затем к панели меню при помощи метода add**(…)** класса JMenuBar:

```
// добавить меню "File" в панель меню
mainMenu.add(fileMenu);
// добавить меню "Help" в панель меню
mainMenu.add(helpMenu);
```

Меню добавляются в панель слева направо в том порядке, в каком порядке происходят вызовы метода add**(…)** класса JMenuBar.

И, наконец, теперь можно установить главное меню окна фрейма.

```
setJMenuBar(mainMenu);        // установка главного меню окна
```

Класс **JMenuItem** определяет поведение элементов меню. Пользуясь методами этого класса, можно блокировать или деблокировать отдельные элементы - это нужно делать, например, если в данный момент функция, соответствующая строке меню, недоступна или не определена. Можно также

изменять текстовые строки, соответствующие элементам меню, что может пригодиться для переопределения их значения.

Некоторые методы класса JMenuItem:

setEnabled – блокирование/разблокирование элемента меню;

getText – получение текстовой строки элемента меню;

isEnabled – проверка того, является ли элемент заблокированным;

setText – установка текстовой строки элемента меню.

Специальный тип элемента меню - **JCheckBoxMenuItem**, позволяет использовать элементы меню в качестве селекторов. Если элемент класса JCheckBoxMenuItem выбран, то рядом с ним отображается пометка. Для получения состояния такого элемента используется метод **getState()**, а для установки его состояния - метод **setState()** класса **JCheckBoxMenuItem**.

Обработка событий меню в Java. Когда пользователь выбирает одну из команд меню, происходит генерация события класса **ActionEvent**. Для его обработки используется интерфейс **ActionListener** с одним методом **actionPerformed**. Для прослушивания команд меню можно добавить прослушивание к каждому пункту меню или же сразу ко всему меню.

```
fileMenu.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent e) {
  String str = e.getActionCommand();
  if ( str == "New")   {   // обработка команды New   }
  if ( str == "Exit")   {   // обработка команды Exit
            dispose();
  }
 }
});
```

Диалоговые окна (класс **JDialog**) используются для одноразового запроса информации у пользователя или для вывода небольших порций информации на экран. Диалоговые окна во всем подобны фреймам, но имеют два важных отличия: во-первых, они не реализуют интерфейс **MenuContainer**; во-

вторых, они могут иметь модальность - это значит, что можно сконструировать диалоговое окно, которое запретит пользователю обращаться к другим окнам (включая и окно браузера) до тех пор, пока пользователь не произведет требуемого действия в этом диалоговом окне.

Некоторые методы класса **JDialog**:

getTitle - возвращает заголовок;

setTitle - устанавливает заголовок;

isResizable - проверяет, может ли пользователь изменять размер фрейма;

setResizable - устанавливает флаг разрешения изменения фрейма;

isModal - определяет, является ли диалоговая панель модальной.

Для окон класса JDialog устанавливается режим размещения **BorderLayout**. Если нужен другой режим размещения, то необходимо установить его явным образом методом **setLayout()**.

Для отображения окна диалоговой панели необходимо вызвать метод **setVisible(true)**. Чтобы спрятать окно, необходимо использовать метод **setVisible(false)**. Метод **dispose**() удаляет окно диалоговой панели окончательно и освобождает все связанные с ним ресурсы.

Каждое диалоговое окно обязательно должно иметь фрейм в качестве родителя. Это значит, что диалоговое окно нельзя открыть непосредственно из апплета. Чтобы создать диалоговое окно, необходимо сначала завести фрейм, даже если его единственным значением будет служить родителем для диалогового окна. Только если апплет уже использует фреймы, можно обойтись без этой подготовительной стадии.

Для того чтобы создать свою диалоговую панель, необходимо определить новый класс, унаследовав его от класса Dialog. Например, создадим класс окон сообщений:

```
// объявление нового класса диалога
class MessageBox extends JDialog {
  // конструктор для создания диалогового окна для вывода
  // сообщения m, заголовок окна передается через t.
```

```
    // родительский фрейм - через p, модальность - через modal
    public MessageBox(String m, JFrame p, String t, boolean modal)   {
       super(parent,sTitle,modal);
       setSize(400,200);
       // здесь можно создать все необходимые компоненты для
       // размещения внутри диалоговой панели, а также
        //установить нужный режим размещения этих компонент
     }
  }
```

Теперь можно создавать диалог.

```
MessageBox msgBox = new MessageBox("String of message", new JFrame(), "Message box", true);
msgBox.setVisible(true);
```

Обработка событий диалога в Java. Для обработки событий в диалоге просто назначьте прослушивание для нужных компонентов диалога, так как было показано выше.

Менеджеры размещения компонентов

Способ размещения компонентов в контейнере зависит от менеджера компоновки, ассоциированного с этим контейнером. С помощью этих менеджеров можно легко и быстро обеспечить необходимое расположение компонентов относительно друг друга и включающего их контейнера. Этот механизм позволяет с легкостью решать такие задачи, как, например, динамическое изменение расположения компонентов в зависимости от меняющейся величины окна. JDK содержит несколько готовых менеджеров размещения, которые пригодны для построения интерфейсов в большинстве апплетов.

Типы менеджеров размещения. К простейшим менеджерам относятся FlowLayout и GridLayout, к более сложным - BorderLayout, CardLayout и GridBagLayout. Для изменения типа менеджера размещения в контейнере используется метод **setLayout()** класса контейнеров (предварительно необходимо

создать объект нового менеджера размещения). Для получения ссылки на используемый в контейнере менеджер размещения существует метод контейнера **getLayout()**.

В классы всех менеджеров размещения входят методы для обеспечения размещения компонент. Метод **layoutContainer()** предназначен для того, чтобы компоненты могли установить для себя предпочтительный размер. Определение минимального размера окна контейнера (с учетом остальных компонент в родительском контейнере), необходимого для размещения всех компонент производится методом **minimumLayoutSize()**. Для определения предпочтительного размера окна контейнера (с учетом остальных компонент в родительском контейнере), необходимого для размещения всех компонент служит метод **prefferedLayoutSize()**.

Менеджер размещения FlowLayout. Принцип действия менеджера **FlowLayout** сводится к следующему: каждый новый добавляемый компонент помещается в текущий горизонтальный ряд слева направо, если в этом ряду есть еще место, а если нет - то компонент смещается вниз и начинает следующий горизонтальный ряд.

Менеджер **FlowLayout** поддерживает три типа выравнивания: влево, вправо и по центру (режим по умолчанию). Тип выравнивания, а также величину отступов между компонентами по вертикали и горизонтали можно задать в конструкторе при создании объекта **FlowLayout**.

Менеджер размещения GridLayout. При помощи менеджера **GridLayout** можно точно указать, где именно разместить тот или иной компонент, достичь ровного, единообразного размещения компонентов. Этот менеджер создает решетку (“таблицу”), состоящую из квадратов одинакового размера, в каждом из которых располагается один компонент. При использовании этого режима компоненты размещаются слева направо и сверху вниз по одному компоненту на квадрат. При помощи конструкторов класса **GridLayout** при создании объектов этого

класса можно задать число сток и столбцов этой “таблицы”, а также величину отступов между строками и столбцами.

Менеджер размещения BorderLayout. При использовании менеджера **BorderLayout** окно контейнера разделяется на рамку и центральную часть. Каждый раз при добавлении нового компонента в контейнер необходимо указывать дополнительный параметр, который может принимать одно из следующих значений: "South" (“юг”, внизу), "North" (“север”, вверху), "East" (“восток”, вправо), "West" (“запад”, влево) и "Center" (в центре). Первые четыре параметра заставляют менеджер BorderLayout относить добавляемые компоненты к соответствующему краю контейнера - нижнему, верхнему, правому и левому. Параметр "Center" позволяет указать, что данный компонент может занять все оставшееся место. Таким образом, элементы, добавляемые с параметром "Center", будут изменять свой размер, заполняя место, не занятое другими компонентами.

Менеджер размещения CardLayout. Этот менеджер размещения позволяет изменять набор компонентов, выводимых на экран, прямо во время работы апплета. Этот менеджер отображает одновременно только один компонент (элемент управления или контейнер) по аналогии с колодой карт. Можно явно указать, какой из компонентов сделать видимым.

Обычно для управления процесса перебора в режиме **CardLayout** используются отдельные органы управления, расположенные в другой панели, например, кнопки. Такие методы как **first()**, **last(), next()** и **previous()**, позволят отображать соответственно первую, последнюю, следующую и предыдущую страницу. Если вызвать метод **next()** при отображении последней страницы, в окне появится первая страница. Аналогично при вызове метода **previous()** для первой страницы отображается последняя страница.

Для того чтобы отобразить произвольную страницу, используется метод **show()**. Однако, этот метод позволяет отображать только те страницы, при добавлении которых в

контейнер использовался метод **add()**, в параметрах которого указываются название компонента и ссылка на сам компонент.

Менеджер размещения GridBagLayout. Режим **GridBagLayout** позволяет размещать компоненты разного размера в таблице, задавая при этом для отдельных компонент размеры отступов и количество занимаемых ячеек.

Менеджер **GridBagLayout** является самым сложным менеджером компоновки. В нем используется наиболее совершенный алгоритм реагирования на изменение размеров контейнера, и позволяет реализовывать сложный интерфейс, в котором контейнер содержит много компонентов различных размеров, некоторые из которых должны находиться в одном и том же заданном положении относительно других. Когда используется этот менеджер, необходимо задавать параметры расположения для каждого компонента с помощью метода **setConstraints()**. Удобнее всего создать для каждого компонента экземпляр класса **GridBagConstraints**, что позволит изменять расположение этого компонента независимо от других.

Если в контейнере имеется несколько вложенных компонент-контейнеров, то при задании менеджеров размещения для них нужно указать, в каком контейнере он будет действовать, например:

```
inputPanel.setLayout(new BorderLayout());
```

Практические задания

1. Изучить основные классы пакета AWT/Swing и классы менеджеров компоновки.
2. Доработать программу, созданную в лабораторной работе № 2:

1) поделить рабочую область на 2 части. Визуализация переносится в левую часть окна, справа появляется панель управления;

2) добавить кнопки «Старт» и «Стоп» в панель управления. Они должны запускать и останавливать симуляцию

соответственно. Если симуляция остановлена, то кнопка «Стоп» должна блокироваться. Если симуляция идет, то блокируется кнопка «Старт». Клавиши **B** и **E** должны функционировать по-прежнему;

3) добавить переключатель «Показывать информацию», который разрешает отображение модального диалога из 7 пункта задания;

4) добавить группу из 2 исключающих переключателей: «Показывать время симуляции» и «Скрывать время симуляции». Клавиша **T** должна функционировать по-прежнему;

5) используя различные менеджеры компоновки, сформировать интерфейс пользователя согласно индивидуальному заданию;

6) добавить в программу главное в меню и панель инструментов, в которых продублировать основные команды вашего интерфейса пользователя;

7) при остановке симуляции должно появляться модальное диалоговое окно (при условии, что оно разрешено) с информацией о количестве и типе сгенерированных объектов, а также времени симуляции. Вся информация выводится в элементе TextArea, недоступном для редактирования. В диалоговом окне должно быть 2 кнопки: «ОК» и «Отмена». При нажатии на «ОК» симуляции останавливается, а при нажатии на «Отмена», соответственно продолжается;

8) предусмотреть проверку данных вводимых пользователем. При вводе неверного значения обрабатывать исключительную ситуацию: выставлять значение по умолчанию и выводить диалоговое окно с сообщением об ошибке;

9) Реализовать следующие элементы управления:

- Периоды рождения объектов – текстовые поля;
- Для задания вероятностей рождения объектов комбобокс и список (шаг значений 10%);
- Дополнить интерфейс поясняющими метками.

Вопросы для самопроверки

1. Что такое GUI?
2. Какие графические библиотеки есть в Java?
3. Какой принцип работы компонентов AWT? Недостатки библиотеки.
4. Какой принцип работы компонентов Swing?
5. Что такое элементы управления и что такое контейнеры?
6. Какие классы элементов управления существуют?
7. Что необходимо сделать, чтобы добавить компонент в контейнер?
8. Как можно перехватить и обработать события, пришедшие от компонентов?
9. Какие типы переключателей существуют?
10. Как несколько переключателей объединить в группу?
11. Чем отличаются выпадающие и невыпадающие списки? Как осуществляется в них выбор элементов?
12. Что такое текстовые поля и текстовые области? Чем они отличаются?
13. Что такое контейнеры? Какие основные виды контейнеров существует?
14. Для чего чаще всего используются панели?
15. В чем основное отличие окон и панелей?
16. Что является обязательным параметром конструктора при создании экземпляра класса окон?
17. Каковы отличительные особенности имеют фреймы?
18. Как добавить меню в контейнер? Как реализуется вложенное меню?
19. Как создать новое меню и добавить в него элементы?
20. Какими методами обрабатываются события меню?
21. Для чего в основном используются окна диалогов?
22. Каковы важные отличия окон диалогов от фреймов?
23. Объект какого класса должен обязательно быть родителем диалогового окна?
24. Что такое модальное окно?
25. Как создать диалог своего класса?
26. Для чего предназначены менеджеры компоновки? Какие существуют режимы размещения?

ГЛАВА 4. КЛАССЫ-КОЛЛЕКЦИИ

При решении практических задач, в которых количество элементов заранее неизвестно, элементы надо часто удалять и добавлять, необходим быстрый поиск данных, можно использовать библиотеки классов-коллекций.

Класс Vector. В языке Java с самых первых версий был разработан класс Vector, предназначенный для хранения переменного числа элементов. В классе Vector из пакета java.util хранятся элементы типа Object, а значит, любого типа. Количество элементов может быть любым и наперед не определяться. Элементы получают индексы 0, 1, 2, ...

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

В Java 2 класс Vector был переработан, чтобы включить его в иерархию классов-коллекций. Поэтому многие действия можно совершать старыми и новыми методами. Рекомендуется использовать новые методы.

В классе четыре конструктора:

Vector () — создает пустой объект нулевой длины;

Vector (int capacity) — создает пустой объект указанной емкости capacity;

Vector (int capacity, int increment) — создает пустой объект указанной емкости capacity и задает число increment, на которое увеличивается емкость при необходимости;

Vector (Collection c) — вектор создается по указанной коллекции. Если capacity отрицательно, создается исключительная ситуация.

Методы класса Vector:

add (Object element) — позволяет добавить элемент в конец вектора:

add (int index, Object element) — можно вставить элемент в указанное место index. Элемент, находившийся на этом месте, и все последующие элементы сдвигаются, их индексы увеличиваются на единицу;

addAll (Collection coll) — позволяет добавить в конец вектора все элементы коллекции coll;

addAll(int index, Collection coll) — вставляет в позицию index все элементы коллекции coll.

set (int index, object element) — заменяет элемент, стоявший в векторе в позиции index, на элемент element;

Количество элементов в векторе всегда можно узнать методом **size().**

capacity()— возвращает емкость вектора.

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

Обратиться к первому элементу вектора можно методом **firstEiement()**, к последнему — методом **lastEiement()**, к любому элементу — методом **get(int index)**. Эти методы возвращают объект класса Object. Перед использованием его следует привести к нужному типу.

Получить все элементы вектора в виде массива типа Object[] можно методами **toArray()**.

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

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

Четыре метода позволяют отыскать позицию указанного элемента element:

indexOf(Object element) — возвращает индекс первого появления элемента в векторе;

indexOf(Object element, int begin) — ведет поиск, начиная с индекса begin включительно;

lastIndexOf(object element) — возвращает индекс последнего появления элемента в векторе;

lastIndexOf (Object element, int start) — ведет поиск от индекса start включительно к началу вектора.

Если элемент не найден, возвращается —1.

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

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

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

Удалить из данного вектора все элементы коллекции coll возможно методом **removeAll(Collection coll)**.

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

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

Удалить все элементы вектора можно методом **clear()** или старым методом **removeAllElements()** или обнулив размер вектора методом **setSize(0).**

Пример 4.1. Работа с вектором

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

Класс Vector является примером того, как можно объекты класса Object, а значит, любые объекты, объединить в коллекцию.

Класс Stack. Класс Stack расширяет класс Vector. Стек (stack) реализует порядок работы с элементами по принципу, который называется LIFO (Last In — First Out). Перед работой создается пустой стек конструктором Stack(). Затем на стек кладутся и снимаются элементы, причем доступен только "верхний" элемент, тот, что положен на стек последним.

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

push(Object item) —помещает элемент item в стек;

pop() — извлекает верхний элемент из стека;

peek() — читает верхний элемент, не извлекая его из стека;

empty() — проверяет, не пуст ли стек;

search(Object item) — находит позицию элемента item в стеке. Верхний элемент имеет позицию 1, под ним элемент 2 и т. д. Если элемент не найден, возвращается — 1.

Пример 4.2. Пример стека

```
Stack stack = new Stack();
stack.push("aaa"); stack.push("bbb"); stack.push("ccc");
for (int i = 0; i < stack.size(); i++)
      System.out.print(stack.get(i) + " - ");
System.out.println();
stack.pop();
for (int i = 0; i < stack.size(); i++)
     System.out.print(stack.get(i) + " - ");
System.out.println();
```

Класс Hashtable. Класс Hashtable расширяет абстрактный класс Dictionary**.** В объектах этого класса хранятся пары "ключ — значение". Из таких пар "Фамилия И. О. — номер" состоит, например, телефонный справочник.

Каждый объект класса Hashtable кроме размера (size) — количества пар, имеет еще две характеристики: емкость (capacity)

— размер буфера, и показатель загруженности (load factor) — процент заполненности буфера, по достижении которого увеличивается его размер.

Для создания объектов класс Hashtable предоставляет четыре конструктора:

Hashtable() — создает пустой объект с начальной емкостью в 101 элемент и показателем загруженности 0,75;

Hashtable (int capacity) — создает пустой объект с начальной емкостью capacity и показателем загруженности 0,75;

Hashtable(int capacity, float loadFactor) — создает пустой объект с начальной емкостью capacity и показателем загруженности loadFactor;

Hashtable (Map f) — создает объект класса Hashtable, содержащий все элементы отображения f, с емкостью, равной удвоенному числу элементов отображения f, но не менее 11, и показателем загруженности 0,75.

Для заполнения объекта класса Hashtable используются два метода:

Object put(Object key, Object value) — добавляет пару "key—value", если ключа key не было в таблице, и меняет значение value ключа key, если он уже есть в таблице. Возвращает старое значение ключа или null, если его не было. Если хотя бы один параметр равен null, возникает исключительная ситуация.

void putAll(Map f) — добавляет все элементы отображения f. В объектах-ключах key должны быть реализованы методы hashCode() и equals ().

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

containsKey(Object key) — возвращает true, если в таблице есть ключ key;

containsValue(Object value) — возвращает true, если в таблице есть ключи со значением value;

isEmpty() — возвращает true, если в таблице нет элементов;

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

keySet()— предоставляет все ключи key таблицы в виде интерфейса Set. Все изменения в объекте Set корректируют таблицу, и наоборот;

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

toString () — возвращает строку, содержащую все пары;

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

clear() — удаляет все элементы, очищая таблицу.

Пример 4.3. Телефонный справочник

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

Класс Properties. Класс Properties расширяет класс Hashtable. Он предназначен в основном для ввода и вывода пар свойств системы и их значений (рис. 4.1) . Пары хранятся в виде строк типа string. В классе Properties два конструктора:

Properties() — создает пустой объект;

Properties(Properties default) — создает объект с заданными парами свойств default.

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

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

string getProperty (String key) — возвращает значение по ключу key;

String getProperty(String.key, String defaultValue) — возвращает значение по ключу key, если такого ключа нет, возвращается defaultValue.

setProperty(String key, String value) — добавляет новую пару, если ключа key нет, и меняет значение, если ключ key есть;

load(InputStream in) — загружает свойства из входного потока in;

list(PrintStream out) и **list(PrintWriter out)** — выводят свойства в выходной поток out;

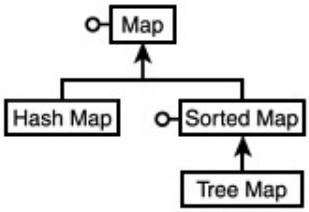
store(OutputStream out, String header) — выводит свойства в выходной поток out с заголовком header.

Пример вывода системных свойств:

```
System.getProperties().list(System.out);
```

Рис. 4.1. Системные свойства

В Java 2 разработана целая иерархия коллекций. Она приведена на рис. 4.2.



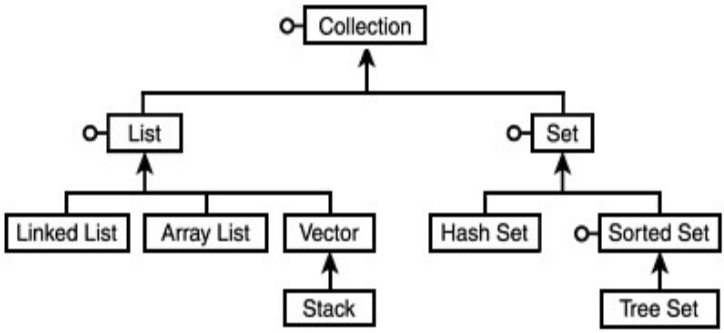


Рис. 4.2. Иерархия интерфейсов и классов-коллекций в Java.

В таблице 4.1 приведены классы коллекций, построенные на основе интерфейсов.

Таблица. 4.1.

Реализация коллекций

Интерфейс	**Реалаизация**				
	Хэш-таблицы	**Массивы**	**Деревья**	**Связанные списки**	**Хэш-таблицы +списки**
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

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

boolean add(Object obj) — добавляет элемент obj в конец коллекции; возвращает false, если такой элемент в коллекции уже есть, а коллекция не допускает повторяющиеся элементы; возвращает true, если добавление прошло удачно;

boolean addAll(Collection coll) — добавляет все элементы коллекции coll в конец данной коллекции;

void clear( ) — удаляет все элементы коллекции;

boolean contains(Object obj) — проверяет наличие элемента obj в коллекции;

boolean containsAll(Collection coll) — проверяет наличие всех элементов коллекции coll в данной коллекции;

boolean isEmpty() — проверяет, пуста ли коллекция;

Iterator iterator() — возвращает итератор данной коллекции;

boolean remove(object obj) — удаляет указанный элемент из коллекции; возвращает false, если элемент не найден, true, если удаление прошло успешно;

boolean removeAll(Collection coll) — удаляет элементы указанной коллекции, лежащие в данной коллекции;

boolean retainAll(Collection coll) — удаляет все элементы данной коллекции, кроме элементов коллекции coll ;

int size() — возвращает количество элементов в коллекции;

Object[] toArray() — возвращает все элементы коллекции в виде массива;

Object[] toArray(Object[] a) — записывает все элементы коллекции в массив a, если в нем достаточно места.

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

Класс Vector — одна из реализаций интерфейса List.

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

void add(int index, Object obj) — вставляет элемент obj в позицию index; старые элементы, начиная с позиции index, сдвигаются, их индексы увеличиваются на единицу;

boolean addAll(int index, Collection coll) — вставляет все элементы коллекции coll ;

Object get(int index) — возвращает элемент, находящийся в позиции index;

int indexOf(Object obj) — возвращает индекс первого появления элемента obj в коллекции;

int lasIindexOf(object obj) — возвращает индекс последнего появления элемента obj в коллекции;

ListIterator listIterator() — возвращает итератор коллекции;

ListIterator listIterator(int index) — возвращает итератор конца коллекции от позиции index ;

Object set (int index, object obj) — заменяет элемент, находящийся в позиции index, элементом obj ;

List subList(int from, int to) — возвращает часть коллекции от позиции from включительно до позиции to исключительно.

Интерфейс Set. Интерфейс Set из пакета java.util, расширяющий интерфейс Collection, описывает неупорядоченную коллекцию, не содержащую повторяющихся элементов. Это соответствует математическому понятию множества (set). Такие коллекции удобны для проверки наличия или отсутствия у элемента свойства, определяющего множество. Новые методы в интерфейс Set не добавлены. Этот интерфейс расширен интерфейсом SortedSet.

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

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

Дополнительные методы интерфейса отражают эти понятия:

Comparator comparator() — возвращает способ упорядочения коллекции;

Object first() — возвращает первый, меньший элемент коллекции;

SortedSet headSet(Object toElement) — возвращает начальные, меньшие элементы до элемента toElement исключительно;

Object last() — возвращает последний, больший элемент коллекции;

SortedSet subSet(Object fromElement, Object toElement) — возвращает подмножество коллекции от элемента fromElement включительно до элемента toElement исключительно;

SortedSet tailSet(Object fromElement) — возвращает последние, большие элементы коллекции от элемента fromElement включительно.

Интерфейс Map. Интерфейс Map из пакета java.util описывает коллекцию, состоящую из пар "ключ — значение". У каждого ключа только одно значение, что соответствует математическому понятию однозначной функции или отображения (map). Такую коллекцию часто называют еще словарем (dictionary) или ассоциативным массивом (associative array).

Интерфейс Map содержит методы, работающие с ключами и значениями:

boolean containsKey(Object key) — проверяет наличие ключа key ;

boolean containsValue(Object value) — проверяет наличие значения value ;

Set entrySet() — представляет коллекцию в виде множества, каждый элемент которого — пара из данного отображения, с которой можно работать методами вложенного интерфейса Map.Entry;

Object get(Object key) — возвращает значение, отвечающее ключу key;

Set keyset() — представляет ключи коллекции в виде множества;

Object put(Object key, Object value) — добавляет пару "key—value", если такой пары не было, и заменяет значение ключа key, если такой ключ уже есть в коллекции;

void putAll (Map m) — добавляет к коллекции все пары из отображения m;

Collection values() — представляет все значения в виде коллекции.

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

Вложенный интерфейс Map.Entry. Этот интерфейс описывает методы работы с парами, полученными методом entrySet(): методы getKey() и getvalue() позволяют получить ключ и значение пары; метод setValue(Object value) меняет значение в данной паре.

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

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

Comparator comparator() — возвращает способ упорядочения коллекции;

Object firstKey() — возвращает первый, меньший элемент коллекции;

SortedMap headMap(Object toKey) — возвращает начало коллекции до элемента с ключом toKey исключительно;

Object lastKey() — возвращает последний, больший ключ коллекции**;**

SortedMap subMap(Object fromKey, Object toKey) — возвращает часть коллекции от элемента с ключом fromKey включительно до элемента с ключом toKey исключительно;

SortedMap tailMap(object fromKey) — возвращает остаток коллекции от элемента fromKey включительно.

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

Интерфейс Iterator. В Java API введен интерфейс Iterator, описывающий способ обхода всех элементов коллекции. В каждой коллекции есть метод **iterator**(), возвращающий реализацию интерфейса Iterator для указанной коллекции. Получив эту реализацию, можно обходить коллекцию в порядке, определенном данным итератором, с помощью методов, описанных в интерфейсе Iterator и реализованных в этом итераторе (пример 4.4).

В интерфейсе Iterator описаны всего три метода:

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

Итератор — это указатель на элемент коллекции. При создании итератора указатель устанавливается перед первым элементом, метод next() перемещает указатель на первый элемент и показывает его. Следующее применение метода next() перемещает указатель на второй элемент коллекции и показывает его. Последнее применение метода next () выводит указатель за последний элемент коллекции.

Пример 4.4. Использование итератора вектора

```
Vector v = new Vector();
…….
for (int i = 0; i < v.size(); i++)
    System.out.print(v.get(i) + ".");
System.out.println();
Iterator it = v.iterator (); // Получаем итератор вектора
```

```
try  {
   while(it.hasNext()) // Пока в векторе есть элементы,
    System.out.println(it.next()); // выводим текущий элемент
}
catch(Exception e){}
```

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

В интерфейс ListIterator добавлены следующие методы:

void add(Object element) — добавляет элемент element перед текущим элементом;

boolean hasPrevious() — возвращает true, если в коллекции есть элементы, стоящие перед текущим элементом;

int nextIndex() — возвращает индекс текущего элемента; если текущим является последний элемент коллекции, возвращает размер коллекции;

Object previous() — возвращает предыдущий элемент и делает его текущим;

int previous index() — возвращает индекс предыдущего элемента;

void set(Object element) — заменяет текущий элемент элементом element; выполняется сразу после next() или previous().

Как видите, итераторы могут изменять коллекцию, в которой они работают, добавляя, удаляя и заменяя элементы. Чтобы это не приводило к конфликтам, предусмотрена исключительная ситуация, возникающая при попытке использования итераторов параллельно "родным" методам коллекции. Именно поэтому в примере 4.4 действия с итератором заключены в блок try-catch().

```
ListIterator lit = v.listIterator(); // Получаем итератор вектора
try {
  while(lit.hasNext())  // Пока в векторе есть элементы
    // Переходим к следующему  элементу и выводим его
```

```
        System.out.println(lit.next());
    // Теперь указатель за концом вектора. Пройдем к началу
    while (lit. hasPrevious ())
      System.out.println(lit. previous());
}
catch (Exception e){}
```

В составе Java API есть полностью реализованные классы-коллекции помимо уже рассмотренных классов Vector, Stack, Hashtable и Properties. Это классы ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap и другие.

Классы, наследующие интерфейс List. Класс **ArrayList** очень похож на класс Vector, имеет тот же набор методов и может использоваться в тех же ситуациях.

В классе ArrayList три конструктора:

ArrayList()—создает пустой объект;

ArrayList(Collection coll) — создает объект, содержащий все элементы коллекции coll;

ArrayList(int initCapacity) — создает пустой объект емкости initCapacity.

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

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

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

В классе LinkedList два конструктора:

LinkedList() - создает пустой объект

LinkedList(Collection coll) — создает объект, содержащий все элементы коллекции coll.

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

HashMap() — создает пустой объект с показателем загруженности 0,75;

HashMap(int capacity) - создает пустой объект с начальной емкостью capacity и показателем загруженности 0,75;

HashMap(int capacity, float loadFactor) — создает пустой объект С начальной емкостью capacity и показателем загруженности loadFactor ;

HashMap(Map f) — создает объект класса HashMap, содержащий все элементы отображения f, с емкостью, равной удвоенному числу элементов отображения f, но не менее 11, и показателем загруженности 0,75.

Упорядоченные отображения. Класс **TreeMap** полностью реализует интерфейс SortedMap. Он реализован как бинарное дерево поиска, следовательно, его элементы хранятся в упорядоченном виде. Это значительно ускоряет поиск нужного элемента. Порядок задается либо естественным следованием элементов, либо объектом, реализующим интерфейс сравнения Comparator.

В этом классе четыре конструктора:

TreeMap() — создает пустой объект с естественным порядком элементов;

TreeMap(Comparator c) — создает пустой объект, в котором порядок задается объектом сравнения с;

TreeMap Map f) — создает объект, содержащий все элементы отображения f, с естественным порядком его элементов;

TreeMap(SortedMap sf) — создает объект, содержащий все элементы отображения sf, в том же порядке.

Сравнение элементов коллекций.

Интерфейс Comparator описывает два метода сравнения:

int compare(Object obji, object obj2) — возвращает отрицательное число, если objl в каком-то смысле меньше obj2 ; нуль, если они считаются равными; положительное число, если objl больше obj2;

boolean equals (Object obj) — сравнивает данный объект с объектом obj, возвращая true, если объекты совпадают в каком-либо смысле, заданном этим методом.

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

Пример 4.5 показывает один из возможных способов упорядочения комплексных чисел — объектов класса Complex. Здесь описывается класс ComplexCompare, реализующий интерфейс Comparator.

Пример 4.5. Сравнение комплексных чисел

```
class Complex  {
   public double Re = 0.0;
   public double Im = 0.0;
   public Complex(){}
   public Complex(double re, double im) { Re = re; Im = im; }
}
class ComplexCompare implements Comparator  {
          public int compare(Object objl, Object obj2)  {
             Complex z1 = (Complex)objl;
             Complex z2 = (Complex)obj2;
             if (z1.Re != z2.Re) return (int)(z1.Re > z2.Re ? 1 : -1);
             else if (z1.Im != z2.Im) return (int)(z1.Im > z2.Im ? 1 : -1);
             else return 0;
          }
          public boolean equals(Object z)  {
             return compare(this, z) == 0;
          }
```

}

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

В классе HashSet четыре конструктора:

HashSet() — создает пустой объект с показателем загруженности 0,75;

HashSet(int capacity) — создает пустой объект с начальной емкостью capacity и показателем загруженности 0,75;

HashSet(int capacity, float loadFactor) — создает пустой объект с начальной емкостью capacity и показателем загруженности loadFactor;

HashSet(Collection coll) — создает объект, содержащий все элементы коллекции coll, с емкостью, равной удвоенному числу элементов коллекции coll, но не менее 11, и показателем загруженности 0,75.

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

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

В классе TreeSet четыре конструктора:

TreeSet() — создает пустой объект с естественным порядком элементов;

TreeSet(Comparator c) — создает пустой объект, в котором порядок задается объектом сравнения c;

TreeSet(Collection coll) — создает объект, содержащий все элементы коллекции coll, с естественным порядком ее элементов;

TreeSet(SortedMap sf) — создает объект, содержащий все элементы отображения sf, в том же порядке.

В пример 4.6 показано, как можно хранить комплексные числа в упорядоченном виде. Порядок задается объектом класса ComplexCompare.

Пример 4.6. Хранение комплексных чисел в упорядоченном виде

```
TreeSet ts = new TreeSet(new ComplexCompare());
ts.add(new Complex(1.2, 3.4));
ts.add(new Complex(-1.25, 33.4));
ts.add(new Complex(1.23, -3.45));
ts.add(new Complex(16.2, 23.4));
Iterator it = ts.iterator();
while(it.hasNext())  {
   Complex c = (Complex)it.next();
   System.out.println("(" + c.Re + "; " + c.Im + ")");
}
```

Практические задания

1. Изучить особенности реализации классов-коллекций в Java.

2. Доработать программу, созданную в лабораторных работах № 2 - 3:

1) добавить генерируемым объектам понятия «время рождения» и «время жизни». Время рождения устанавливается в момент генерации объекта, и по значению соответствует времени, прошедшему от начала симуляции. Время жизни – время, через которое объект должен исчезнуть, считая от времени рождения;

2) вынести параметры времен жизни объектов в пользовательский интерфейс. Для каждого типа объекта должно задаваться собственное время. Рекомендуется использовать текстовые поля, но следуют помнить о проверке на ввод некорректных данных;

3) огранизовать коллекцию сгенерированных объектов по варианту. При генерации объекта происходит добавление его в

коллекцию. При обновлении времени обойдите коллекцию и удалите все объекты, время жизни которых истекло;

4) добавить генерируемым объектам уникальные целочисленные идентификаторы (случайные числа), которые назначаются при генерации объекта. Для хранения сгенерированных идентификаторов используйте коллекцию удобную для поиска по варианту;

5) добавьте в панель управления кпопку «Текущие объекты». По нажатию на эту кнопку появляется модальное диалоговое окно, содержащее список всех «живых» объектов на момент нажатия со временем их рождения (время рождения – ключ). В класс диалогового окна должна передаваться коллекция с хранением объектов по времени рождения. Типы коллекций задаются вариантом.

Вариант 1

Коллекция для хранения объектов: Vector

Коллекция для хранения и поиска уникальных идентификаторов: HashSet

Коллекция для хранения объектов по времени рождения: TreeMap

Вариант 2

Коллекция для хранения объектов: ArrayList

Коллекция для хранения и поиска уникальных идентификаторов: HashSet

Коллекция для хранения объектов по времени рождения: TreeMap

Вариант 3

Коллекция для хранения объектов: LinkedList

Коллекция для хранения и поиска уникальных идентификаторов: HashSet

Коллекция для хранения объектов по времени рождения: TreeMap

Вариант 4

Коллекция для хранения объектов: Vector

Коллекция для хранения и поиска уникальных идентификаторов: TreeSet

Коллекция для хранения объектов по времени рождения: HashMap

Вариант 5

Коллекция для хранения объектов: ArrayList

Коллекция для хранения и поиска уникальных идентификаторов: TreeSet

Коллекция для хранения объектов по времени рождения: HashMap

Вариант 6

Коллекция для хранения объектов: LinkedList

Коллекция для хранения и поиска уникальных идентификаторов: TreeSet

Коллекция для хранения объектов по времени рождения: HashMap

Вариант 7

Коллекция для хранения объектов: Vector

Коллекция для хранения и поиска уникальных идентификаторов: HashSet

Коллекция для хранения объектов по времени рождения: HashMap

Вариант 8

Коллекция для хранения объектов: ArrayList

Коллекция для хранения и поиска уникальных идентификаторов: HashSet

Коллекция для хранения объектов по времени рождения: HashMap

Вариант 9

Коллекция для хранения объектов: Vector

Коллекция для хранения и поиска уникальных идентификаторов: TreeSet

Коллекция для хранения объектов по времени рождения: TreeMap

Вариант 10

Коллекция для хранения объектов: ArrayList

Коллекция для хранения и поиска уникальных идентификаторов: TreeSet

Коллекция для хранения объектов по времени рождения: TreeMap

Вопросы для самопроверки

1. Для чего используются классы-коллекции?
2. Что такое класс Vector? Для чего он используется? Каковы его достоинства и недостатки? Какая структура данных используется в классе Vector?
3. Назовите особенности организации класса Stack.
4. Для чего применяется класс Hashtable? Какая структура данных используется в классе Hashtable?
5. Что такое коэффициент загруженности?
6. Что такое емкость класса-коллекции?
7. Назначение и особенности применения класса Properties.
8. Расскажите об иерархии интерфейсов коллекций Java. Объясните назначение каждого интерфейса.
9. Каково назначение интерфейса Collection?
10. Опишите возможности применения интерфейсов Map, Set и List.
11. Для чего применяются интерфейсы Iterator и ListIterator?
12. Опишите классы ArrayList и LinkedList.
13. Опишите классы HashSet, TreeSet.
14. Опишите классы HashMap, TreeMap.
15. Какие структуры данных используются в этих классах?
16. Какие алгоритмы для обработки коллекций существуют в Java и как их использовать?

ГЛАВА 5. МНОГОПОТОКОВЫЕ ПРИЛОЖЕНИЯ

Процессы, потоки и приоритеты

Обычно в многозадачной операционной системе (ОС) выделяют такие объекты, как процессы и потоки.

Процесс (process) - это объект, который создается ОС при запуске приложения. Процессу выделяется отдельное адресное пространство, это пространство физически недоступно для других процессов. Процесс может работать с файлами или с каналами связи локальной или глобальной сети.

Для каждого процесса ОС создает один главный **поток** (thread), который является потоком выполняющихся по очереди команд центрального процессора. При необходимости главный поток может создавать другие потоки, пользуясь для этого программным интерфейсом ОС. Все потоки, созданные процессом, выполняются в адресном пространстве этого процесса и имеют доступ к ресурсам процесса.

Если процесс создал несколько потоков, то все они выполняются параллельно, причем время центрального процессора (или нескольких центральных процессоров в многопроцессорных системах) распределяется между этими потоками.

Распределением времени центрального процессора занимается специальный модуль операционной системы - планировщик. Планировщик по очереди передает управление отдельным потокам, так что даже в однопроцессорной системе создается полная иллюзия параллельной работы запущенных потоков.

Распределение времени выполняется для потоков, а не для процессов. Потоки, созданные разными процессами, конкурируют между собой за получение процессорного времени. Каждому потоку задается приоритет его выполнения, уровень которого определяет очередность выполнения того или иного потока.

Реализация многозадачности в Java

Для создания многозадачных приложений Java необходимо воспользоваться классом **java.lang.Thread**. В этом классе

определены все методы, необходимые для создания потоков, управления их состоянием и синхронизации.

Есть две возможности использования класса Thread.

Во-первых, можно создать собственный класс на базе класса Thread и переопределить метод run(). Новая реализация этого метода будет работать в рамках отдельного потока.

Во-вторых, создаваемый класс, может реализовать интерфейс Runnable и реализовать метод run(), который будет работать как отдельный поток.

Создание подкласса Thread. При использовании этого способа для потоков определяется отдельный класс, например:

```
class myThread extends Thread  {
  public void run()   {
    // здесь можно добавить код, который будет
    // выполняться в рамках отдельного потока
  }
  // здесь можно добавить специализированный для класса код
}
```

Метод run**()** должен быть всегда переопределен в классе, наследованном от Thread. Именно он определяет действия, выполняемые в рамках отдельного потока. Если поток используется для выполнения циклической работы, этот метод содержит внутри себя бесконечный цикл.

Метод run() получает управление при запуске потока методом **start()** класса Thread**.** В случае апплетов создание и запуск потоков обычно осуществляется в методе start() апплета.

Остановка работающего потока раньше выполнялась методом **stop()** класса Thread**.** Обычно остановка всех работающих потоков, созданных апплетом, выполняется в методе stop() апплета. Сейчас не рекомендуется использование этого метода. Завершение работы потока желательно проводить так, чтобы происходило естественное завершение метода run. Для этого используется управляющая переменная в потоке.

Пример 5.1. Многопоточное приложение с использованием наследников класса Thread

```
// Поток для расчета координат прямоугольника
class ComputeRects extends Thread  {
   boolean going = true;
// конструктор получает ссылку на создателя объекта - апплет
   public ComputeRects(MainApplet parentObj)    {
      parent = parentObj;
   }
   public void run()    {
      while(going)   {
         int w = parent.size().width-1, h = parent.size().height-1;
         parent.RectCoordinates
         ((int)(Math.random()*w),(int)(Math.random()*h));
      }
   }
   MainApplet parent;              // ссылка на создателя объекта
}
// Поток для расчета координат овала
class ComputeOvals extends Thread   {
   boolean going = true;
    public ComputeOvals(MainApplet parentObj)    {
      parent = parentObj;
   }
   public void run()    {
      while(going)   {
         int w = parent.size().width-1, h = parent.size().height-1;
         parent.OvalCoordinates
         ((int)(Math.random()*w),(int)(Math.random()*h));
      }
   }
   MainApplet parent; // ссылка на создателя объекта
}

public class MainApplet extends JApplet   {
   ComputeRects m_rects = null;
   ComputeOvals m_ovals = null;
   int m_rectX = 0;   int m_rectY = 0;
   int m_ovalX = 0;   int m_ovalY = 0;
   // Синхронный метод для установки координат
```

```
    // прямоугольника из другого потока
    public synchronized void RectCoordinates(int x, int y)    {
        m_rectX = x; m_rectY = y;
        this.repaint();
    }
    // Синхронный метод для установки координат овала
    // из другого потока
    public synchronized void OvalCoordinates(int x, int y)    {
        m_ovalX = x; m_ovalY = y;
        this.repaint();
    }
    @Override
    public void start()     {
        super.start();
        // Запускаем потоки
        if (m_rects == null) {
              m_rects = new ComputeRects(this);   m_rects.start();
        }
        if (m_ovals == null) {
             m_ovals = new ComputeOvals(this);   m_ovals.start();
        }
    }
    @Override
    public void stop()     {
        super.stop();
        // Останавливаем потоки
        if(m_rects != null) m_rects.going = false;
        if(m_ovals != null) m_ovals.going = false;
    }
   public void paint(Graphics g)     {
       int w = this.getWidth(), h = this.getHeight();
       g.clearRect(0, 0, w, h);
       g.setColor(Color.red);
       g.fillRect(m_rectX, m_rectY, 20, 20);
       g.setColor(Color.blue);
       g.fillOval(m_ovalX, m_ovalY, 20, 20);
   }
    public static void main(String[] args)     {    }
}
```

Обратите внимание на запуск и остановку потоков в методах start и stop соответственно, а также на объявление синхронных методов и использованием ключевого слова **synchronized**. Синхронизация крайне важна в многопоточных приложениях, так как потоки, работающие над одними и теми же данными одновременно, могут испортить эти данные.

Реализация интерфейса Runnable. Если нет возможности расширять класс Thread, то можно применить второй способ реализации многозадачности. Допустим, уже существует класс MyClass, функциональные возможности которого удовлетворяют разработчика. Необходимо, чтобы он выполнялся как отдельный поток.

```
class MyClass implements Runnable  {
  // код класса - объявление его элементов и методов
  // этот метод получает управление при запуске потока
  public void run()  {
    // здесь можно добавить код, который будет
    // выполняться в рамках отдельного потока
  }
}
```

Добавим новый поток в пример 5.1, реализованный через интерфейс Runnable.

Пример 5.2. Доработанное многопоточное приложение

```
// Поток для расчета координат линии
class ComputeLines implements Runnable  {
   boolean going = true;
   public ComputeLines(MainApplet parentObj)  {
      parent = parentObj;
   }
   public void compute()  {
      int w = parent.size().width-1, h = parent.size().height-1;
      parent.LineCoordinates
            ((int)(Math.random()*w),(int)(Math.random()*h),
            (int)(Math.random()*w), (int)(Math.random()*h));
```

```
    }
    MainApplet parent; // ссылка на создателя объекта
    public void run()    {
        while(going) { compute(); }
    }
}

public class MainApplet extends JApplet {
  . . .              // скопируйте из предыдущего примера
    ComputeLines m_lines = null;
    int m_lineX1 = 0, m_lineX2 = 0, m_lineY1 = 0, m_lineY2 = 0;
    // Синхронный метод для установки координат
    // прямоугольника из другого потока
    public synchronized void RectCoordinates(int x, int y)    {
        m_rectX = x; m_rectY = y;
        this.repaint();
    }
    // Синхронный метод для установки координат овала
    // из другого потока
    public synchronized void OvalCoordinates(int x, int y)    {
        m_ovalX = x; m_ovalY = y;
        this.repaint();
    }
    // Синхронный метод для установки координат линии
    // из другого потока
 public synchronized void LineCoordinates(int x1, int y1, int x2, int y2)  {
        m_lineX1 = x1; m_lineX2 = x2; m_lineY1 = y1; m_lineY2 = y2;
        this.repaint();
    }
    @Override
    public void start()    {
        super.start();
        // Запускаем потоки
        . . .        // скопируйте из предыдущего примера
        if (m_lines == null) {
            m_lines = new ComputeLines(this);
            new Thread(m_lines).start();
        }
    }
    @Override
```

```
    public void stop()    {
        super.stop();
        // Останавливаем потоки
        . . .        // скопируйте из предыдущего примера
        if(m_lines != null) m_lines.going = false;
    }
    public void paint(Graphics g)    {
        . . .        // скопируйте из предыдущего примера
        g.setColor(Color.green);
        g.drawLine(m_lineX1, m_lineX2, m_lineY1, m_lineY2);
    }
    public static void main(String[] args)    {    }
}
```

Применение анимации для мультизадачности

Одним из наиболее распространенных применений апплетов является создание анимационных эффектов типа бегущей строки, мерцающих огней и других эффектов, привлекающих внимание пользователя. Для достижения таких эффектов необходим механизм, позволяющий выполнять перерисовку всего окна апплета или его части периодически с заданным интервалом. Перерисовка окна апплета выполняется методом paint(), который вызывается виртуальной машиной Java асинхронно по отношению к выполнению другого кода апплета, если содержимое окна было перекрыто другими окнами. Для периодической перерисовки окна апплета необходимо создание потока (или нескольких потоков), которые будут выполнять рисование в окне апплета асинхронно по отношению к коду апплета. Например, можно создать поток, который периодически обновляет окно апплета, вызывая для этого метод repaint(), или рисовать из потока непосредственно в окне апплета. Также можно использовать класс таймера (этот механизм уже использовался в предыдущих работах). Таймер, по сути, тоже является потоком, выполняющимся параллельно с основным.

Рассмотрим пример апплета 5.3, который умеет сам себя перерисовывать при помощи дополнительного потока.

Пример 5.3. Самоперерисовывающийся апплет

```
public class MainApplet extends JApplet implements Runnable   {
   boolean m_isGoing = false;
   @Override
   public void run()    {
      while (m_isGoing)       {
         repaint();
         try  {    Thread.sleep(500); }
         catch(InterruptedException e) { stop(); }
      }
   }
   @Override
   public void start()    {
      super.start();
      // Запускаем поток
      m_isGoing = true;
      new Thread(this).start();
   }
   @Override
   public void stop()    {
      super.stop();
      // Останавливаем потоки
      m_isGoing = false;
      // Дадим потоку время завершиться
      try  {    Thread.sleep(500); }
      catch(InterruptedException e) {}
   }
  public void paint(Graphics g)    {
     int w = this.getWidth(), h = this.getHeight();
     g.setColor(new Color((int)(Math.random() * 255),
     (int)(Math.random() * 255), (int)(Math.random() * 255)));
     g.fillRect(0, 0, w, h);
  }
   public static void main(String[] args)    {   }
}
```

Класс Thread содержит несколько конструкторов и большое количество методов для управления потоков.

Некоторые методы класса Thread:

currentThread() − возвращает ссылку на выполняемый в настоящий момент объект класса Thread;

sleep() − переводит выполняемый в данное время поток в режим ожидания в течение указанного промежутка времени ;

start() − начинает выполнение потока. Это метод приводит к вызову соответствующего метода run();

run() − фактическое тело потока. Этот метод вызывает после запуска потока;

stop() − останавливает поток (устаревший метод);

isAlive() − определяет, является ли поток активным (запущенным и не остановленным);

suspend() − приостанавливает выполнение потока (устаревший метод);

resume() − возобновляет выполнение потока (устаревший метод). Этот метод работает только после вызова метода suspend();

setPriority() − устанавливает приоритет потока (принимает значение от MIN_PRIORITY до MAX_PRIORITY);

getPriority() − возвращает приоритет потока;

wait() − переводит поток в состояние ожидания выполнения условия, определяемого переменной условия;

join() − ожидает, пока данный поток не завершит своего существования бесконечно долго или в течении некоторого времени;

setDaemon() − отмечает данный поток как поток-демон или пользовательский поток. Когда в системе останутся только потоки-демоны, программа на языке Java завершит свою работу;

isDaemon() − возвращает признак потока-демона.

Состояние потока

Во время своего существования поток может переходить во многие состояния, находясь в одном из нижеперечисленных состояний:

- Новый поток
- Выполняемый поток

- Невыполняемый поток
- Завершенный поток

Новый поток. При создании экземпляра потока этот поток приобретает состояние “Новый поток”:

```
Thread myThread=new Thread();
```

В этот момент для данного потока распределяются системные ресурсы; это всего лишь пустой объект. В результате все, что с ним можно делать - это запустить: myThread.start**()**;

Любой другой метод потока в таком состоянии вызвать нельзя, это приведет к возникновению исключительной ситуации.

Выполняемый поток. Когда поток получает метод start**()**, он переходит в состояние “Выполняемый поток”. Процессор разделяет время между всеми выполняемыми потоками согласно их приоритетам.

Невыполняемый поток. Если поток не находится в состоянии “Выполняемый поток”, то он может оказаться в состоянии “Невыполняемый поток”. Это состояние наступает тогда, когда выполняется одно из четырех условий:

- *Поток был приостановлен*. Это условие является результатом вызова метода suspend(). После вызова этого метода поток не находится в состоянии готовности к выполнению; его сначала нужно “разбудить” с помощью метода resume(). Это полезно в том случае, когда необходимо приостановить выполнение потока, не удаляя его. Поскольку метод suspend не рекомендуется к использованию, приостановка потока должна выполняться через управляющую переменную.
- *Поток ожидает***.** Это условие является результатом вызова метода sleep(). После вызова этого метода поток переходит в состояние ожидания в течении некоторого определенного промежутка времени и не может выполняться до истечения этого промежутка. Даже если ожидающий поток имеет доступ к процессору, он его не получит. Когда указанный промежуток времени пройдет, поток переходит в состояние “Выполняемый поток”. Метод resume() не может повлиять на

процесс ожидания потока, этот метод применяется только для приостановленных потоков.

– *Поток ожидает извещения*. Это условие является результатом вызова метода wait(). С помощью этого метода потоку можно указать перейти в состояние ожидания выполнения условия, определяемого переменной условия, вынуждая его тем самым приостановить свое выполнение до тех пор, пока данное условие удовлетворяется. Какой бы объект не управлял ожидаемым условием, изменение состояния ожидающих потоков должно осуществляться посредством одного из двух методов этого потока - notify() или notifyAll(). Если поток ожидает наступление какого-либо события, он может продолжить свое выполнение только в случае вызова для него этих методов.

– *Поток заблокирован другим потоком*. Это условие является результатом блокировки операцией ввода-вывода или другим потоком. В этом случае у потока нет другого выбора, как ожидать до тех пор, пока не завершится команда ввода-вывода или действия другого потока. В этом случае поток считается невыполняемым, даже если он полностью готов к выполнению.

Завершенный поток. Когда метод run() завершается, поток переходит в состояние “Завершенный поток”.

Приоритеты потоков. В языке Java каждый поток обладает приоритетом, который оказывает влияние на порядок его выполнения. Потоки с высоким приоритетом выполняются чаще потоков с низким приоритетом. Поток наследует свой приоритет от потока, его создавшего. Если потоку не присвоен новый приоритет, он будет сохранять данный приоритет до своего завершения. Приоритет потока можно установить с помощью метода setPriority(), присваивая ему значение от MIN_PRIORITY до MAX_PRIORITY (константы класса Thread). По умолчанию потоку присваивается приоритет Thread.NORM_PRIORITY.

Потоки в языке Java планируются с использованием алгоритма планирования с фиксированными приоритетами. Этот алгоритм, по существу, управляет потоками на основе их взаимных

приоритетов, кратко его можно изложить в виде следующего правила: в любой момент времени будет выполняться “Выполняемый поток” с наивысшим приоритетом. Как выполняются потоки одного и того же приоритета, в спецификации Java не описано.

Группы потоков. Все потоки в языке Java должны входить в состав группы потоков. В классе Thread имеется три конструктора, которые дают возможность указывать, в состав какой группы должен входить данный создаваемый поток.

Группы потоков особенно полезны, поскольку внутри их можно запустить или приостановить все потоки, а это значит, что при этом не потребуется иметь дело с каждым потоком отдельно. Группы потоков предоставляют общий способ одновременной работы с рядом потоков, что позволяет значительно сэкономить время и усилия, затрачиваемые на работу с каждым потоком в отдельности.

В приведенном ниже фрагменте программы создается группа потоков под названием genericGroup (родовая группа). Когда группа создана, создаются несколько потоков, входящих в ее состав:

```
ThreadGroup genericGroup=new ThreadGroup("My generic group");
Thread t1=new Thread(genericGroup,this);
Thread t2=new Thread(genericGroup,this);
```

Если при создании нового потока не указать, к какой конкретной группе он принадлежит, этот поток войдет в состав группы потоков main (главная группа). Иногда ее еще называют текущей группой потоков. В случае апплета main может и не быть главной группой. Право присвоения имени принадлежит Web-браузеру. Для того чтобы определить имя группы потоков, можно воспользоваться методом **getName**() класса ThreadGroup.

Для того, чтобы определить к какой группе принадлежит данный поток, используется метод **getThreadGroup()**, определенный в классе Thread. Этот метод возвращает имя группы потоков, в которую можно послать множество методов, которые будут применяться к каждому члену этой группы.

Программирование движения объекта

Для того чтобы запрограммировать движение объекта из одной точки в другую, вспомним равномерное движение.

Пусть (x1;y1) – начальная координата объекта, а (x2;y2) – конечная координата.

Направление движения задается вектором (dx;dy) = Нормализация(x2-x1; y2-y1), где

Нормализация(a,b) = (a / sqrt(a^2 + b^2); b / sqrt(a^2 + b^2)).

Таким образом, координаты объекта в момент времени T от начала движения, движущегося со скоростью V по прямой, равны: (xc;yc) = (x1; y1) + (dx;dy) * V * T;

Скорость V измеряется в пиксель/сек, т.е. количество пикселей пройденных объектом на 1 сек.

Когда (xc;yc) станет равно (x2;y2), тогда объект дошел до конечной точки. На практике сравнивать текущие координаты с концом отрезка не корректно, так как промах на 1 пиксель из-за округления, например, отправит наш объект в бесконечное путешествие по прямой. Можно, например, прикинуть время, которое потребуется объекту, чтобы дойти до конца отрезка, поделив длину отрезка на скорость объекта. Условием продолжения движения будет время от начала движения T меньшее времени, которое требуется, чтобы пройти отрезок.

Движение по окружности программируется иначе.

Пусть (x0;y0) – центр окружности движения. Тогда координаты объекта в момент времени T от начала движения, движущегося со скоростью V по окружности, равны:

(xc;yc) = (x0; y0) + R * (cos(V*T); sin(V*T)),

где R – радиус окружности.

В этом случае в отличие от предыдущего V – угловая скорость, т.е. измеряется в радиан/сек.

Практические задания

1. Изучить особенности реализации и работы потоков в Java.

2. Доработать программу, созданную в лабораторных работах № 2-4:

1) создать абстрактный класс BaseAI, описывающий «интеллектуальное поведение» объектов по варианту. Класс должен быть выполнен в виде отдельного потока и работать с коллекцией объектов;

2) реализовать класс BaseAI для каждого из видов объекта, включив в него поведение, описанное в индивидуальном задании по варианту;

3) синхронизовать работу потоков расчета интеллекта объектов с их рисованием. Рисование должно остаться в основном потоке. Синхронизация осуществляется через передачу данных в основной поток;

4) добавить в панель управления кнопки для остановки и возобновления работы интеллекта каждого вида объектов. Реализовать через засыпание/пробуждение потоков;

5) добавить в панель управления выпадающие списки для выставления приоритетов каждого из потоков.

Вариант 1

1. Муравьи-рабочие двигаются в один из углов области их обитания (например, [0;0]) по прямой со скоростью V, а затем возвращаться обратно в точку своего рождения с той же скоростью.
2. Муравьи-воины двигаются по окружности с радиусом R со скоростью V.

Вариант 2

1. Пчелы-рабочие двигаются в один из углов области их обитания (например, [0;0]) по прямой со скоростью V, а затем возвращаться обратно в точку своего рождения с той же скоростью.
2. Трутни двигаются хаотично со скоростью V. Хаотичность достигается случайной сменой направления движения раз в N секунд.

Вариант 3

1. Золотые рыбки двигаются по оси X от одного края области обитания до другого со скоростью V.

2. Гуппи двигаются по оси Y от одного края области обитания до другого со скоростью V.

Вариант 4

1. Обыкновенные кролики двигаются хаотично со скоростью V. Хаотичность достигается случайной сменой направления движения раз в N секунд.

2. Альбиносы двигаются по оси X от одного края области обитания до другого со скоростью V.

Вариант 5

1. Грузовые машины двигаются в левую верхнюю четверть области симуляции (т.е. прямоугольник с верхним-левым углом в точке 0;0, шириной/длиной = (w/2;h/2), где w и h – ширина и длина области симуляции) со скоростью V по прямой. Конечная точка движения – случайная точка в пределах этой области. Если машина сгенерировалась сразу в этой области, то она никуда не движется. По прибытии в конечную точку машина больше не движется.

2. Легковые машины двигаются в нижнюю правую четверть области симуляции (т.е. прямоугольник с верхним-левым углом в точке w/2;h/2, шириной/длиной = (w/2;h/2), где w и h – ширина и длина области симуляции) со скоростью V по прямой. Конечная точка движения – случайная точка в пределах этой области. Если машина сгенерировалась сразу в этой области, то она никуда не движется. По прибытии в конечную точку машина больше не движется.

Вариант 6

1. Разработчики двигаются хаотично со скоростью V. Хаотичность достигается случайной сменой направления движения раз в N секунд.

2. Менеджеры двигаются по окружности с радиусом R со скоростью V.

Вариант 7

1. Капитальные дома двигаются (в городах будущего и не такое возможно) в левую верхнюю четверть области симуляции

(т.е. прямоугольник с верхним-левым углом в точке 0;0, шириной/длиной = (w/2;h/2), где w и h – ширина и длина области симуляции) со скоростью V по прямой. Конечная точка движения – случайная точка в пределах этой области. Если дом сгенерировался сразу в этой области, то он никуда не движется. По прибытии в конечную точку дом больше не движется.

2. Деревянные дома после генерации начинают двигаться в нижнюю правую четверть области симуляции (т.е. прямоугольник с верхним-левым углом в точке w/2;h/2, шириной/длиной = (w/2;h/2), где w и h – ширина и длина области симуляции) со скоростью V по прямой. Конечная точка движения – случайная точка в пределах этой области. Если дом сгенерировался сразу в этой области, то он никуда не движется. По прибытии в конечную точку дом больше не движется.

Вариант 8

1. Автомобили двигаются по оси X от одного края области симуляции до другого со скоростью V.
2. Мотоциклы двигаются по оси Y от одного края области симуляции до другого со скоростью V.

Вариант 9

1. Студенты двигаются хаотично со скоростью V. Хаотичность достигается случайной сменой направления движения раз в N секунд.
2. Студентки двигаются по окружности с радиусом R со скоростью V.

Вариант 10

1. Юр. лица двигаются в левую верхнюю четверть области симуляции (т.е. прямоугольник с верхним-левым углом в точке 0;0, шириной/длиной = (w/2;h/2), где w и h – ширина и длина области симуляции) со скоростью V по прямой. Конечная точка движения – случайная точка в пределах этой области. Если запись сгенерировалась сразу в этой области, то она никуда не движется. По прибытии в конечную точку запись больше не движется.
2. Физ. лица двигаются в нижнюю правую четверть области симуляции (т.е. прямоугольник с верхним-левым углом в точке

w/2;h/2, шириной/длиной = (w/2;h/2), где w и h – ширина и длина области симуляции) со скоростью V по прямой. Конечная точка движения – случайная точка в пределах этой области. Если запись сгенерировалась сразу в этой области, то она никуда не движется. По прибытии в конечную точку запись больше не движется.

Вопросы для самопроверки

1. Что такое процесс и поток (нить)?
2. Чем определяется порядок передачи управления потокам?
3. Какие есть способы реализации многозадачности в Java?
4. Что необходимо сделать для создания подкласса потоков (подкласса Thread)?
5. Когда запускается на выполнение метод run() подкласса Thread?
6. Какими методами класса Thread необходимо запускать поток на выполнение и останавливать его?
7. Что необходимо сделать для реализации классом интерфейса Runnable?
8. В каких состояниях может находиться поток?
9. Какой поток считается новым, выполняемым и завершенным?
10. В каких ситуациях поток является невыполняемым?
11. Когда возникают исключительные ситуации при работе с потоками?
12. Что такое группы потоков и чем они полезны?
13. Что такое родовая группа потоков и главная группа потоков?

ГЛАВА 6. ПОТОКИ ДАННЫХ. РАБОТА С ЛОКАЛЬНЫМИ ФАЙЛАМИ

Организация ввода-вывода в Java

Для того чтобы отвлечься от особенностей конкретных устройств ввода/вывода, в Java употребляется понятие потока (stream). Считается, что в программу идет входной поток (input stream) символов Unicode или просто байтов, воспринимаемый в программе методами read(). Из программы методами write(), print(), println() выводится выходной поток (output stream) символов или байтов. При этом неважно, куда направлен поток: на консоль, на принтер, в файл или в сеть.

Конечно, полное игнорирование особенностей устройств ввода/вывода сильно замедляет передачу информации. Поэтому в Java все-таки выделяется файловый ввод/вывод, вывод на печать, сетевой поток.

В классе system определены три потока статическими полями in, out и err. Они называются соответственно стандартным вводом (stdin), стандартным выводом (stdout) и стандартным выводом сообщений (stderr). Эти стандартные потоки могут быть соединены с разными конкретными устройствами ввода и вывода.

Потоки out и err — это экземпляры класса PrintStream, организующего выходной поток байтов. Эти экземпляры выводят информацию на консоль методами print(), println() и write(), которых в классе PrintStream имеется около двадцати для разных типов аргументов.

Поток err предназначен для вывода системных сообщений программы: трассировки, сообщений об ошибках или, просто, о выполнении каких-то этапов программы.

Поток in — это экземпляр класса InputStream. Он назначен на клавиатурный ввод с консоли методами read().

Еще один вид потока — поток байтов, составляющих объект Java. Его можно направить в файл или передать по сети, а потом восстановить в оперативной памяти. Эта операция называется сериализацией (serialization) объектов.

Методы организации потоков собраны в классы пакета **java.io**.

Кроме классов, организующих поток, в пакет java.io входят классы с методами преобразования потока, например, можно преобразовать поток байтов, образующих целые числа, в поток этих чисел.

Классы потоков ввода-вывода

Итак, в Java есть целых четыре иерархии классов для создания, преобразования и слияния потоков. Во главе иерархии четыре класса, непосредственно расширяющих класс Object:

Reader — абстрактный класс, в котором собраны самые общие методы символьного ввода;

Writer — абстрактный класс, в котором собраны самые общие методы символьного вывода;

InputStream — абстрактный класс с общими методами байтового ввода;

OutputStream — абстрактный класс с общими методами байтового вывода.

Классы входных потоков Reader и InputStream определяют по три метода ввода:

read() — возвращает один символ или байт, взятый из входного потока, в виде целого значения типа int; если поток уже закончился, возвращает -1;

read(char[] buf) — заполняет заранее определенный массив buf символами из входного потока; в классе Inputstream массив типа byte[]; метод возвращает фактическое число взятых из потока элементов или -1, если поток уже закончился;

read(char[] buf, int offset, int len) — заполняет часть символьного или байтового массива buf, начиная с индекса offset, число взятых из потока элементов равно len; метод возвращает фактическое число взятых из потока элементов или -1.

Эти методы выбрасывают IOException, если произошла ошибка ввода/вывода.

Классы выходных потоков Writer и OutputStream определяют по три почти одинаковых метода вывода:

write(char[] buf) — выводит массив в выходной поток, в классе Outputstream массив имеет тип byte[];

write(char[] buf, int offset, int len) — выводит len элементов массива buf, начиная с элемента с индексом offset;

write(int elem) в классе Writer - выводит 16, а в классе Outputstream 8 младших битов аргумента elem в выходной поток,

В классе Writer есть еще два метода:

write(string s) — выводит строку s в выходной поток;

write(String s, int offset, int len) — выводит len символов строки s, начиная с символа с номером offset.

Многие подклассы классов Writer и OutputStream осуществляют буферизованный вывод. При этом элементы сначала накапливаются в буфере, в оперативной памяти, и выводятся в выходной поток только после того, как буфер заполнится. Часто надо вывести информацию в поток еще до заполнения буфера. Для этого предусмотрен метод **flush().**

Наконец, по окончании работы с потоком его необходимо закрыть методом **close().**

Все классы пакета java.io можно разделить на две группы: классы, создающие поток (data sink), и классы, управляющие потоком (data processing).

Иерархия классов потоков ввода-вывода

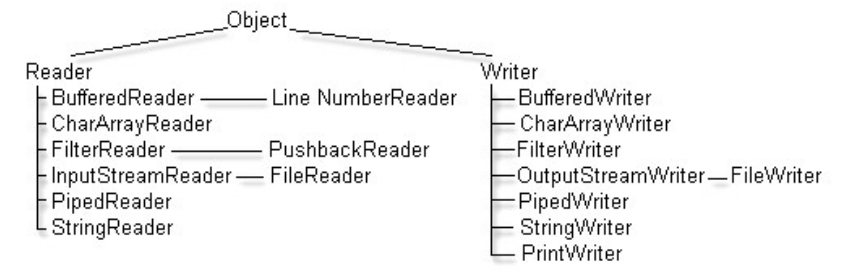


Рис. 6.1. Иерархия символьных потоков

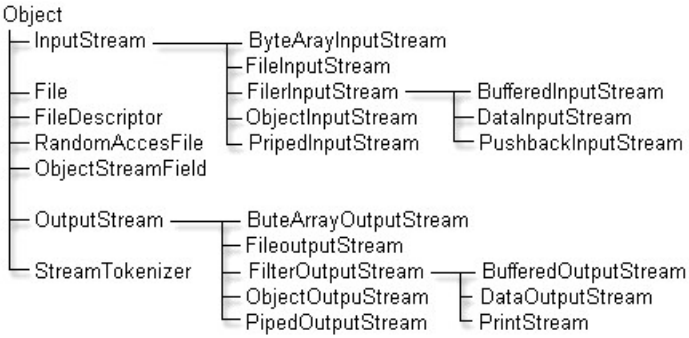


Рис. 6.2. Классы байтовых потоков

Классы, создающие потоки можно разделить на пять групп:

Классы, создающие потоки, связанные с файлами:

FileReader FileInputStream
FileWriter FileOutputStream
RandomAccessFile

Классы, создающие потоки, связанные с массивами:

CharArrayReader ByteArrayInputStream
CharArrayWriter ByteArrayOutputStream

Классы, создающие каналы обмена информацией между подпроцессами:

PipedReader PipedInputStream
PipedWriter PipedOutputStream

Классы, создающие символьные потоки, связанные со строкой:

StringReader StringWriter

Классы, создающие байтовые потоки из объектов Java:

ObjectInputStream ObjectOutputStream

Классы, управляющие потоком, получают в своих конструкторах уже имеющийся поток и создают новый, преобразованный поток.

Четыре класса выполняют буферизованный ввод/вывод:

BufferedReader BufferedInputStream
BufferedWriter BufferedOutputStream

Два класса преобразуют поток байтов в восемь простых типов Java:

DataInputStream DataOutputStream

Два класса связаные с выводом на строчные устройства — экран дисплея, принтер:

PrintWriter PrintStream

Два класса связывают байтовый и символьный потоки:

InputStreamReader OutputStreamWriter

Класс **StreamTokenizer** позволяет разобрать входной символьный поток на отдельные элементы (tokens) подобно тому, как класс StringTokenizer разбирает строку.

Из управляющих классов выделяется класс **SequenceInputStream**, сливающий несколько потоков, заданных в конструкторе, в один поток, и класс **LineNumberReader**, "умеющий" читать выходной символьный поток построчно. Строки в потоке разделяются символами '\n' и/или '\r'.

Консольный ввод/вывод. Для вывода на консоль используется метод println() класса PrintStream. Вместо System.out.println(), то вы можете определить новую ссылку на System.out, например:

PrintStream pr = System.out; и писать просто pr.println().

Консоль является байтовым устройством, и символы Unicode перед выводом на консоль должны быть преобразованы в байты.

Трудности с отображением кириллицы возникают, если вывод на консоль производится в кодировке, отличной от локали. Именно так происходит в русифицированных версиях MS Windows. Обычно в них устанавливается локаль с кодовой страницей CP1251, а вывод на консоль происходит в кодировке CP866.

В этом случае надо заменить PrintStream, который не может работать с символьным потоком, на PrintWriter и вставить "переходное кольцо" между потоком символов Unicode и потоком байтов System.out, выводимых на консоль, в виде объекта класса OutputStreamWriter. В конструкторе этого объекта следует указать нужную кодировку, в данном случае, CP866. Все это можно сделать одним оператором:

```
PrintWriter pw = new PrintWriter(new
        OutputStreamWriter( System.out, "Cp866"), true);
```

Класс PrintStream буферизует выходной поток. Второй аргумент true его конструктора вызывает принудительный сброс содержимого буфера в выходной поток после каждого выполнения метода println().

После этого можно выводить любой текст методами класса PrintWriter, которые просто дублируют методы класса PrintStream, и писать, например,

```
pw.println("Это русский текст");
```

Если вы пользуетесь окном Output в Netbeans 7 как консолью (так происходит в Netbeans 7 по умолчанию), то кодировку менять не нужно, там сразу используется локаль.

Ввод с консоли производится методами read() класса InputStream с помощью статического поля in класса System. С консоли идет поток байтов, полученных из scan-кодов клавиатуры. Эти байты должны быть преобразованы в символы Unicode такими же кодовыми таблицами, как и при выводе на консоль. Преобразование идет по той же схеме — для правильного ввода кириллицы удобнее всего определить экземпляр класса BufferedReader, используя в качестве "переходного кольца" объект класса InputStreamReader:

```
BufferedReader br = new BufferedReader(new InputStreamReader ( System.in, "Cp866"));
```

Класс BufferedReader переопределяет три метода read() своего суперкласса Reader. Кроме того, он содержит метод readLine().

Метод readLine() возвращает строку типа string, содержащую символы входного потока, начиная с текущего, и заканчивая символом '\n' и/или '\r'. Эти символы-разделители не входят в возвращаемую строку. Если во входном потоке нет символов, то возвращается null.

Пример 6.1. Консольный ввод/вывод

```
class PrWr  {
public static void main(String[] args)      {
      try       {
         boolean use866 = args.length > 0 ?
               Boolean.valueOf(args[0].toLowerCase()) : false;
         BufferedReader br = new BufferedReader(use866 ?
               new InputStreamReader(System.in, "Cp866") :
               new InputStreamReader(System.in));
         PrintWriter pw = new PrintWriter(use866 ?
               new OutputStreamWriter(System.out, "Cp866") :
               new OutputStreamWriter(System.out), true);

         String s = "Это строка с русским текстом";
         System.out.println("System.out puts: " + s);
         pw.println("PrintWriter puts: " + s) ;
         int c = 0;
         pw.println("Посимвольный ввод:");
```

```
            while((c = br.read()) != -1)      {
               if ((char)c == 'q') break;
               pw.println((char)c);
            }
            pw.println("Построчный ввод:");
            do      {
               s = br.readLine();
               pw.println(s);
            } while(!s.equals("q"));
         }
         catch(Exception e)     { System.out.println(e);  }
      }
   }
```

Файловый ввод/вывод. Поскольку файлы в большинстве современных операционных систем понимаются как последовательность байтов, для файлового ввода/вывода создаются байтовые потоки с помощью классов FileInputStream и FileOutputStream. Это особенно удобно для бинарных файлов, хранящих байт-коды, архивы, изображения, звук.

Но очень много файлов содержат тексты, составленные из символов. Несмотря на то, что символы могут храниться в кодировке Unicode, эти тексты чаще всего записаны в байтовых кодировках. Поэтому и для текстовых файлов можно использовать байтовые потоки. В таком случае со стороны программы придется организовать преобразование байтов в символы и обратно.

Чтобы облегчить это преобразование, в пакет java.io введены классы FileReader и FileWriter. Они организуют преобразование потока: со стороны программы потоки символьные, со стороны файла — байтовые. Это происходит потому, что данные классы расширяют классы InputStreamReader и OutputStreamWriter, соответственно, значит, содержат "переходное кольцо" внутри себя.

Несмотря на различие потоков, использование классов файлового ввода/вывода очень похоже.

В конструкторах всех четырех файловых потоков задается имя файла в виде строки типа String или ссылка на объект класса File. Конструкторы не только создают объект, но и отыскивают файл и открывают его. Например:

```
FileInputStream fis = new FileInputStream("PrWr.Java");
FileReader fr = new FileReader("D:\\jdkl.5\\src\\PrWr.Java");
```

При неудаче выбрасывается исключение класса FileNotFoundException, конструктор класса FileWriter выбрасывает более общее исключение IOException.

После открытия выходного потока типа FileWriter или FileQutputStream содержимое файла, если он был не пуст, стирается. Для того чтобы можно было делать запись в конец файла в классах предусмотрен конструктор с двумя аргументами. Если второй аргумент равен true, то происходит дозапись в конец файла, если false, то файл заполняется новой информацией. Например:

```
FileWriter fw = new FileWriter("ch8.txt", true);
FileOutputstream fos = new
FileOutputStream("D:\\samples\\newfile.txt");
```

Теперь можно читать файл или записывать:

```
fis.read();     fr.read();
fos.write((char)c);     fw.write((char)c);
```

Преобразование потоков в классах FileReader и FileWriter выполняется по кодовым таблицам установленной на компьютере локали. Для правильного ввода кириллицы надо применять FileReader, а не FileInputStream. Если файл содержит текст в кодировке, отличной от локальной кодировки, то придется вставлять "переходное кольцо" вручную, как это делалось для консоли, например:

```
InputStreamReader isr = new InputStreamReader(fis, "KOI8_R"));
```

Получение свойств файла. Получить сведения о файле можно от предварительно созданного экземпляра класса File. В

конструкторе этого класса File(String filename) указывается путь к файлу или каталогу, записанный по правилам операционной системы. Конструктор не проверяет, существует ли файл с таким именем, поэтому после создания объекта следует это проверить логическим методом exists().

Класс File содержит около 40 методов, позволяющих узнать различные свойства файла или каталога.

Прежде всего, логическими методами **isFile(), isDirectory()** можно выяснить, является ли путь, указанный в конструкторе, путем к файлу или каталогу.

Для каталога можно получить его содержимое — список имен файлов и подкаталогов — методом **list(),** возвращающим массив строк String[]. Можно получить такой же список в виде массива объектов класса File[] методом listFiles(). Можно выбрать из списка только некоторые файлы, реализовав интерфейс FileNameFiiter и обратившись к методу **list(FileNameFilter filter).**

Если каталог с указанным в конструкторе путем не существует, его можно создать логическим методом **mkdir().** Этот метод возвращает true, если каталог удалось создать. Логический метод **mkdirs()** создает еще и все несуществующие каталоги, указанные в пути. Пустой каталог удаляется методом **delete().**

Для файла можно получить его длину в байтах методом **length(),** время последней модификации в секундах с 1 января 1970 г. методом **lastModified().** Если файл не существует, эти методы возвращают нуль.

Логические методы **canRead(), canWrite()** показывают права доступа к файлу.

Файл можно переименовать логическим методом **renameTo(File newName)** или удалить логическим методом **delete**(). Эти методы возвращают true, если операция прошла удачно.

Если файл с указанным в конструкторе путем не существует, его можно создать логическим методом **createNewFile(),** возвращающим true, если файл не существовал, и его удалось создать, и false, если файл уже существовал.

Статическими методами:

createTempFile(String prefix, String suffix, File tmpDir) ;
createTempFile(String prefix, String suffix);

можно создать временный файл с именем prefix и расширением suffix в каталоге tmpDir или каталоге, указанном в системном свойстве java.io.tmpdir. Имя prefix должно содержать не менее трех символов. Если suffix = null, то файл получит суффикс .tmp.

Перечисленные методы возвращают ссылку типа File на созданный файл. Если обратиться к методу **deleteOnExit(),** то по завершении работы JVM временный файл будет уничтожен.

Несколько методов **getxxx**() возвращают имя файла, имя каталога и другие сведения о пути к файлу. Эти методы полезны в тех случаях, когда ссылка на объект класса File возвращается другими методами и нужны сведения о файле. Метод **toURL()** возвращает путь к файлу в форме URL.

В примере 6.2 показан пример использования класса File.

Пример 6.2. Определение свойств файла и каталога

```
class FileTest  {
   public static void main(String[] args) throws IOException    {
      File f = new File("FileTest.java");
      System.out.println();
      System.out.println("File \"" + f.getName() +
            "\" " +(f.exists()? "is " : "isn't ") + "existed");
      System.out.println("You " + (f.canRead()? "can " : "can't ") + "read
this file");
      System.out.println("You " + (f.canWrite()? "can " : "can't ") + "write
to this file");
      System.out.println("File size is " + f.length() + " B");
      System.out.println();
      File d = new File("C:");
      System.out.println("Content of " + d.getPath());
      if (d.exists() && d.isDirectory())   {
         String[] s = d.list();
         for (int i = 0; i < s.length; i++)
            System.out.println(s[i]);
      }
```

```
    }
}
```

Буферизованный ввод/вывод. Операции ввода/вывода по сравнению с операциями в оперативной памяти выполняются очень медленно. Для компенсации в оперативной памяти выделяется некоторая промежуточная область — буфер, в которой постепенно накапливается информация. Когда буфер заполнен, его содержимое быстро переносится процессором, буфер очищается и снова заполняется информацией.

Для этой цели есть четыре специальных класса **BufferedXXX**, перечисленных выше. Они присоединяются к потокам ввода/вывода как "переходное кольцо", например:

```
InputStreamReader isr = new InputStreamReader(fis, "KOI8_R"));
FileWriter fw = new FileWriter("ch8.txt", true);
BufferedReader br = new BufferedReader(isr);
BufferedWriter bw = new BufferedWriter(fw);
```

Поток примитивных типов Java. Класс **DataOutputstream** позволяет записать данные простых типов Java в выходной поток байтов методами writeBoolean(boolean b), writeByte(int b), writeShort(int h), writeChar(int c), writeInt(int n), writeLong(long 1), writeFloat(float f), writeDouble(double d).

Кроме того, метод writeBytes(string s) записывает каждый символ строки s в один байт, отбрасывая старший байт кодировки каждого символа Unicode, а метод writeChars(string s) записывает каждый символ строки s в два байта, первый байт — старший байт кодировки Unicode, так же, как это делает метод writeChar().

Класс **DataInputStream** преобразует входной поток байтов типа InputStream, составляющих данные простых типов Java, в данные этого типа. Данные из этого потока можно прочитать методами readBoolean(), readByte(), readShort(), readChar(), readlnt(), readLong(), readFloat(), readDouble(), возвращающими данные соответствующего типа.

Кроме того, методы readUnsignedByte() и readUnsignedShort () возвращают целое типа int, в котором старшие три или два байта

нулевые, а младшие один или два байта заполнены байтами из входного потока.

Программа в примера 6.3 записывает в файл fib.txt числа Фибоначчи, а затем читает этот файл и выводит его содержимое на консоль. Для контроля записываемые в файл числа тоже выводятся на консоль.

Пример 6.3. Ввод/вывод данных

```
class DataPrWr  {
   public static void main(String[] args) throws IOException    {
      DataOutputStream dos = new DataOutputStream(
new FileOutputStream("fib.txt"));
      int a = 1, b = 1, c = 1;
      for (int k = 0; k < 40; k++)       {
         System.out.print(b + " ");
         dos.writeInt(b);
         a = b; b = c; c = a + b;
      }
      dos.close();
      System.out.println("\n");
      DataInputStream dis = new DataInputStream(
new FileInputStream("fib.txt"));
      while(true)  {
         try    {
            a = dis.readInt();
            System.out.print(a + " ");
         }
         catch(IOException e) {
            dis.close();
            System.out.println("\nEnd of file");
            System.exit (0);
         }
      }
   }
}
```

Обратите внимание на то, что попытка чтения за концом файла выбрасывает исключение класса IOException, его

обработка заключается в закрытии файла и окончании программы.

Прямой доступ к файлу. Если необходимо интенсивно работать с файлом, записывая в него данные разных типов Java, изменяя их, отыскивая и читая нужную информацию, то лучше всего воспользоваться методами класса RandomAccessFile.

В конструкторах этого класса

```
RandomAccessFile(File file, String mode);
RandomAccessFile(String fileName, String mode);
```

аргументом mode задается режим открытия файла. Это может быть строка "r" — открытие файла только для чтения, или "rw" — открытие файла для чтения и записи.

Этот класс собрал все полезные методы работы с файлом. Он содержит все методы классов DataInputStream и DataOutputStream, кроме того, позволяет прочитать сразу целую строку методом readLine() и отыскать нужные данные в файле.

Байты файла нумеруются, начиная с 0, подобно элементам массива. Файл снабжен неявным указателем (file pointer) текущей позиции. Чтение и запись производится, начиная с текущей позиции файла. При открытии файла конструктором указатель стоит на начале файла, в позиции 0. Текущую позицию можно узнать методом **getFilePointer**(). Каждое чтение или запись перемещает указатель на длину прочитанного или записанного данного. Всегда можно переместить указатель в новую позицию, роз методом **seek(long pos)**.

Каналы обмена информацией. В пакете java.io есть четыре класса **Pipedxxx**, организующих обмен информацией между потоками - Thread.

В одном подпроцессе-потоке — источнике информации — создается объект класса **PipedWriter** или **PipedOutputStream**, в который записывается информация методами write() этих классов.

В другом подпроцессе-потоке — приемнике информации — формируется объект класса **PipedReader** или **PipedInputStream**.

Он связывается с объектом-источником с помощью конструктора или специальным методом **connect**(), и читает информацию методами read().

Источник и приемник можно создать и связать в обратном порядке.

Так создается однонаправленный канал (pipe) информации. На самом деле это некоторая область оперативной памяти, к которой организован совместный доступ двух или более подпроцессов. Доступ синхронизируется, записывающие процессы не могут помешать чтению.

Если надо организовать двусторонний обмен информацией, то создаются два канала.

В примере 6.4 метод run() класса Source генерирует информацию, для простоты просто целые числа k, и передает ее в канал методом pw.write(k). Метод run() класса Target читает информацию из канала методом pr.read(). Концы канала связываются с помощью конструктора класса Target.

Пример 6.4. Канал обмена информацией

```
class Target extends Thread    {
   private PipedReader pr;
   Target(PipedWriter pw)    {
      try   {  pr = new PipedReader(pw);  }
      catch(IOException e)        {
         System.err.println("From Target(): " + e);
      }
   }
   PipedReader getStream(){ return pr; }
   public void run()    {
      while(true)        {
         try    {  System.out.println("Reading: " + pr.read());  }
         catch(IOException e)            {
            System.out.println("The job's finished.");
            System.exit(0);
         }
      }
   }
}
```

```
class Source extends Thread  {
   private PipedWriter pw;
   Source()   {  pw = new PipedWriter();  }
   PipedWriter getStream()  { return pw;}
   public void run()   {
      for (int k = 0; k < 10; k++)   {
         try    {
            pw.write(k);
            System.out.println("Writing: " + k);
         }
         catch(Exception e)    {
            System.err.println("From Source.run(): " + e) ;
         }
      }
   }
}

public class Main     {
   public static void main(String[] args) throws IOException    {
      Source s = new Source();
      Target t = new Target(s.getStream());
      s.start();
      t.start();
   }
}
```

Сериализация объектов. Методы классов **ObjectInputStream** и **ObjectOutputStream** позволяют прочитать из входного байтового потока или записать в выходной байтовый поток данные сложных типов — объекты, массивы, строки.

Процесс записи объекта в выходной поток получил название сериализации (serialization), а чтения объекта из входного потока и восстановления его в оперативной памяти — десериализации (deserialization).

Сериализации можно подвергнуть объекты, которые реализует интерфейс **Serializable**. Этот интерфейс не содержит ни полей, ни методов. По сути дела это только пометка, разрешающая сериализацию класса.

```
class A implements Serializabie{...}
```

Для сериализации достаточно создать объект класса ObjectOutputStream, связав его с выходным потоком, и выводить в этот поток объекты методом writeObject().

В выходной поток выводятся все нестатические поля объекта, независимо от прав доступа к ним, а также сведения о классе этого объекта, необходимые для его правильного восстановления при десериализации. Байт-коды методов класса не сериализуются.

Если в объекте присутствуют ссылки на другие объекты, то они тоже сериализуются, а в них могут быть ссылки на другие объекты, которые опять-таки сериализуются, и получается целое множество связанных между собой сериализуемых объектов. Метод writeObject() распознает две ссылки на один объект и выводит его в выходной поток только один раз. К тому же, он распознает ссылки, замкнутые в кольцо, и избегает зацикливания.

Все классы объектов, входящих в такое сериализуемое множество, а также все их внутренние классы, должны реализовать интерфейс Serializable, иначе будет выброшено исключение NotSerializableException и процесс сериализации прервется.

Десериализация происходит так же просто, как и сериализация. Нужно только соблюдать порядок чтения элементов потока.

В примере 6.5 мы создаем объект класса GregorianCalendar с текущей датой и временем, сериализуем его в файл date.ser, через три секунды десериализуем и сравниваем с текущим временем.

Пример 6.5. Сериализация объекта

```
import java.util.*;

class SerDatef   {
   public static void main(String[] args) throws Exception   {
   GregorianCalendar d = new GregorianCalendar();
   ObjectOutputStream oos = new ObjectOutputStream(
                        new FileOutputStream("date.ser"));
   oos.writeObject(d);
   oos.flush();
```

```
        oos.close();
        Thread.sleep(3000);

        ObjectInputStream ois = new ObjectInputStream(
                            new FileInputStream("date.ser"));
       GregorianCalendar oldDate = (GregorianCalendar)ois.readObject();
        ois.close();
        GregorianCalendar newDate = new GregorianCalendar();
        System.out.println("Old time = " +
                    oldDate.get(Calendar.HOUR) +
                    ":" + oldDate.get(Calendar.MINUTE) +
                    ":" +  oldDate.get(Calendar.SECOND) +
                    "\nNew time = " + newDate.get(Calendar.HOUR) +
                    ":" + newDate.get(Calendar.MINUTE) +
                    ":" + newDate.get(Calendar.SECOND));
    }
}
```

Метод writeObject() не записывает в выходной поток поля, помеченные **static** и **transient**. Впрочем, это положение можно изменить, переопределив метод writeObject() или задав список сериализуемых полей.

Вообще процесс сериализации можно полностью настроить под свои нужды, переопределив методы ввода/вывода и воспользовавшись вспомогательными классами. Можно даже взять весь процесс на себя, реализовав не интерфейс Serializable, а интерфейс **Externalizable**, но тогда придется реализовать методы **readExternal**() и **writeExternal**(), выполняющие ввод/вывод.

Файловые диалоги. При работе с файлами часто требуются стандартные файловые диалоги. Библиотека Swing предлагает класс JFileChooser для реализации этого функционала.

```
JFileChooser fc = new JFileChooser();
fc.showOpenDialog(frame);
File selFile = fc.getSelectedFile();
fc.showSaveDialog(frame);
selFile = fc.getSelectedFile();
```

Практические задания

1. Изучить особенности реализации потоков ввода-вывода в Java.
2. Доработать программу, созданную в лабораторных работах № 2-5:

1) добавить в главное меню команду «Консоль». По этой команде должно появляться немодальное диалоговое окно с многострочным текстовым полем, занимающим всю область окна. В это окно можно вводить команды по варианту. В это же окно выводится реакция программы на команду;

2) для передачи команды в основное окно программы использовать каналы ввода-вывода;

3) создать конфигурационный файл для программы. В конфигурационный файл должны сохраняться все настройки симуляции, т.е. все данные и состояния, которые задаются в панели управления программы. Конфигурационный файл должен читаться при запуске программы и записываться при выходе. Формат файла текстовый;

4) добавить в главное меню команды «Загрузить» и «Сохранить». Команда «Сохранить» вызывает сериализацию всех «живых» объектов в ней. Команда «Загрузить» останавливает текущую симуляцию (если симуляция запущена) и загружает объекты из выбранного файла. После открытия симуляцию можно запустить, загруженные объекты должны вести себя естественно;

5) использовать стандартные файловые диалоги.

Вариант 1

Реализовать в консоли команды «Старт» и «Стоп» симуляции. Кнопки в интерфейсе должны вести себя так же, как если бы нажимали их, а не исполняли команды (блокироваться по очереди).

Вариант 2

Реализовать в консоли команду «Вернуть количество живых пчел-рабочих/трутней». Как параметр в команду должен передаваться идентификатор вида объекта.

Вариант 3

Реализовать в консоли команды «Установить вероятность рождения золотых рыбок» и «Получить вероятность рождения золотых рыбок». Как параметр в команду установки должно передаваться значение вероятности. Полученная вероятность должна выводиться на консоль.

Вариант 4

Реализовать в консоли команду «Сократить число кроликов-альбиносов на N%». Как параметр в команду должно передаваться значение N%.

Вариант 5

Реализовать в консоли команды «Остановить интеллектуальное поведение объектов» и «Продолжить интеллектуальное поведение объектов». Команда «Остановить» - останавливает поток расчета интеллекта объектов, а команда «Продолжить» - возобновляет расчет после остановки.

Вариант 6

Реализовать в консоли команду «Уволить всех менеджеров» и «Нанять N новых менеджеров». Первая команда удаляет всех менеджеров из симуляции (новые продолжают генерироваться), вторая – генерирует N новых менеджеров. N – параметр команды.

Вариант 7

Реализовать в консоли команды «Установить вероятность генерации капитальных домов» и «Получить вероятность генерации капитальных домов». Как параметр в команду установки должно передаваться значение вероятности. Полученная вероятность должна выводиться на консоль.

Вариант 8

Реализовать в консоли команду «Сократить число мотоциклов на N%». Как параметр в команду должно передаваться значение N%.

Вариант 9

Реализовать в консоли команды «Показывать время симуляции» и «Скрывать время симуляции». Чекбокс в интерфейсе должен вести себя так же, как если бы использовали его, а не исполняли команды (галочка появляется/исчезает).

Вариант 10

Реализовать в консоли команду «Вернуть количество существующих юридических/физических лиц в картотеке». Как параметр в команду должен передаваться идентификатор вида объекта.

Вопросы для самопроверки

1. Что такое потоки ввода-вывода и для чего они нужны?
2. Какие классы Java являются базовыми для работы с потоками?
3. В чем разница между байтовыми и символьными потоками?
4. Какие стандартные потоки ввода-вывода существуют в Java, каково их назначение? На базе каких классов создаются стандартные потоки?
5. Чем является поток System.in, System.out, System.err? Какими методами чаще всего пользуются при работе с этим потоком?
6. Как создать файловый поток для чтения и записи данных?
7. В чем заключается особенность создания потока, связанного с локальным файлом?
8. Как создать поток для форматированного обмена данными, связанного с локальным файлом?
9. Как добавить буферизацию для потока форматированного обмена данными, связанного с локальным файлом?
10. За счет чего буферизация ускоряет работу приложений с потоками?
11. Когда применяется принудительный сброс буферов?
12. Для выполнения каких операций применяется класс File?
13. Для чего предназначен класс RandomAccessFile? Чем он отличается от потоков ввода и вывода?

14. Как организовать передачу объектов через потоки ввода-вывода?
15. Что такое сериализация объектов? Что такое десериализация объектов?
16. Как объявить класс сериализуемым?
17. Какие поля класса не сериализуются?
18. Как передаются данные между потоками в многопоточном приложении?

ГЛАВА 7. СЕТЕВЫЕ ПРИЛОЖЕНИЯ «КЛИЕНТ-СЕРВЕР»

Сетевые средства

При увеличении числа компьютеров в организации, для повышения эффективности работы возникает необходимость объединения их в сеть. Сначала все компьютеры в сети равноправны и делают одно и то же — это **одноранговая** (peer-to-peer) **сеть**. Потом покупается компьютер с большими и быстрыми жесткими дисками, и все файлы организации начинают храниться на данных дисках — этот компьютер становится **файл-сервером**, предоставляющим услуги хранения, поиска, архивирования файлов. Остальные компьютеры становятся клиентами этих серверов. Такая архитектура сети называется архитектурой **клиент-сервер** (client-server).

Сервер постоянно находится в состоянии ожидания, он прослушивает (listen) сеть, ожидая запросов от клиентов. Клиент связывается с сервером и посылает ему запрос (request) с описанием услуги, например, имя нужного файла. Сервер обрабатывает запрос и отправляет ответ (response). После этого связь может быть разорвана или продолжиться, организуя сеанс связи, называемый сессией (session).

Запросы клиента и ответы сервера формируются по строгим правилам, совокупность которых образует протокол (protocol) связи. Всякий протокол должен, содержать правила соединения

компьютеров. Итак, все сетевые соединения основаны на трех основных понятиях: **клиент, сервер и протокол**.

Для обслуживания протокола: формирования запросов и ответов, проверок их соответствия протоколу, расшифровки сообщений, связи с сетевыми устройствами создается программа, состоящая из двух частей. Одна часть программы работает на сервере, другая — на клиенте. Эти части так и называются серверной частью программы и клиентской частью программы, или, короче, сервером и клиентом.

Обычно на одном компьютере-сервере работают несколько программ-серверов. Одна программа занимается электронной почтой, другая — пересылкой файлов, третья предоставляет Web-страницы. Для того чтобы их различать, каждой программе-серверу придается номер порта (port). Это просто целое положительное число, которое указывает клиент, обращаясь к определенной программе-серверу. Число, может быть любым, но наиболее распространенным протоколам даются стандартные номера, чтобы клиенты были твердо уверены, что обращаются к нужному серверу. Так, стандартный номер порта электронной почты 25, пересылки файлов — 21, Web-сервера — 80. Стандартные номера простираются от 0 до 1023. Числа с 1024 до 65 535, можно использовать для своих собственных номеров портов.

Чтобы равномерно распределить нагрузку на сервер, часто несколько портов прослушиваются программами-серверами одного типа. Web-сервер, кроме порта с номером 80, может прослушивать порт 8080, 8001 и еще какой-нибудь другой.

В процессе передачи сообщения используется несколько протоколов. В современных глобальных сетях принят стек из четырех протоколов, называемый стеком протоколов **TCP/IP**.

Сначала мы пишем сообщение, пользуясь программой, реализующей **прикладной** (application) протокол: HTTP (80), SMTP (25), TELNET (23), FTP (21), POP3 (100) или другой протокол. В скобках записан стандартный номер порта.

Затем сообщение обрабатывается по **транспортному** (transport) протоколу. К нему добавляются, в частности, номера

портов отправителя и получателя, контрольная сумма и длина сообщения. Наиболее распространены транспортные протоколы **TCP** (Transmission Control Protocol) и **UDP** (User Datagram Protocol). В результате работы протокола TCP получается **TCP-пакет** (packet), а протокола **UDP** — **дейтаграмма** (datagram).

Дейтаграмма невелика — всего около килобайта, поэтому сообщение делится на прикладном уровне на части, из которых создаются отдельные дейтаграммы. Дейтаграммы посылаются одна за другой. Они могут идти к получателю разными маршрутами, прийти совсем в другом порядке, некоторые дейтаграммы могут потеряться. Прикладная программа получателя должна сама позаботиться о том, чтобы собрать из полученных дейтаграмм исходное сообщение. Для этого обычно перед посылкой части сообщения нумеруются. Таким образом, протокол UDP работает как почтовая служба.

TCP-пакет тоже невелик, и пересылка также идет отдельными пакетами, но протокол TCP обеспечивает надежную связь. Сначала устанавливается соединение с получателем. Только после этого посылаются пакеты. Получение каждого пакета подтверждается получателем, при ошибке посылка пакета повторяется. Сообщение аккуратно собирается получателем. Для отправителя и получателя создается впечатление, что пересылаются не пакеты, а сплошной поток байтов, поэтому передачу сообщений по протоколу TCP часто называют передачей потоком.

Далее сообщением занимается программа, реализующая **сетевой** (network) протокол. Чаще всего это протокол IP (Internet Protocol). Он добавляет к сообщению адрес отправителя и адрес получателя, и другие сведения. В результате получается **IP-пакет**.

Наконец, IP-пакет поступает к программе, работающей по **канальному** (link) протоколу ENET, SLIP, PPP, и сообщение принимает вид, пригодный для передачи по сети.

На стороне получателя сообщение проходит через эти четыре уровня протоколов в обратном порядке, освобождаясь от

служебной информации, и доходит до программы, реализующей прикладной протокол.

Какой же адрес заносится в IP-пакет? Каждый компьютер или другое устройство, подключенное к объединению сетей Internet, так называемый **хост** (host), получает уникальный номер — четырехбайтовое целое число, называемое **IP-адресом** (IP-address). По традиции содержимое каждого байта записывается десятичным числом от 0 до 255, называемым **октетом** (octet), и эти числа пишутся через точку: 138.2.45.12 или 17.056.215.38.

IP-адрес удобен для машины, но неудобен для человека. Поэтому IP-адрес хоста дублируется **доменным именем** (domain name).

В Java IP-адрес и доменное имя объединяются в один класс **InetAddress** пакета java.net. Экземпляр этого класса создается статическим методом **getByName (string host)** данного же класса, в котором аргумент host— это доменное имя или IP-адрес.

Работа по протоколу TCP

Программы-серверы, прослушивающие свои порты, работают под управлением различных операционных систем. Чтобы сгладить различия в реализациях разных серверов, между сервером и портом введен промежуточный программный слой, названный **сокетом** (socket). Слово socket переводится как электрический разъем, розетка. Так же как к розетке при помощи вилки можно подключить любой электрический прибор, к сокету можно присоединить любой клиент, лишь бы он работал по тому же протоколу, что и сервер. Каждый сокет связан (bind) с одним портом, говорят, что сокет прослушивает (listen) порт. Соединение с помощью сокетов устанавливается так.

1. Сервер создает сокет, прослушивающий порт сервера.

2. Клиент тоже создает сокет, через который связывается с сервером, сервер начинает устанавливать (**accept**) связь с клиентом.

3. Устанавливая связь, сервер создает новый сокет с новым номером, и сообщает этот номер клиенту.

4. Клиент посылает запрос на сервер через порт с новым номером.

После этого соединение становится совершенно симметричным — два сокета обмениваются информацией, а сервер через старый сокет продолжает прослушивать прежний порт, ожидая следующего клиента.

В Java сокет — это объект класса **Socket** из пакета java.io. В классе шесть конструкторов, в которые разными способами заносится адрес хоста и номер порта. Чаще всего применяется конструктор **Socket(String host, int port) .**

Многочисленные методы доступа устанавливают и получают параметры сокета. Нам понадобятся методы, создающие потоки ввода/вывода:

getInputStream() — возвращает входной поток типа InputStream;
getOutputStream() — возвращает выходной поток типа OutputStream.

В приведенном примере 7.1 рассматриваются сервер и клиент, работающие по протоколу TCP. Клиент делает запрос на совершение действия, сервер выполняет это действие и возвращает результат. Получив запрос, сервер распознает тип операции по идентификатору, выполняет операцию и возвращает результат клиенту в виде целого числа. При малейшем подозрении на нарушение протокола сервер должен разрывать соединение с клиентом. Помните, что с сервером одновременно могут взаимодействовать тысячи клиентов, и если сервер упал, то все клиенты перестали работать.

Пример 7.1. Примитивный клиент

```
import java.net.Socket;

class Client{
  public static void main(String[] args) throws Exception {
    // Имя хоста и номер порта
    String host = "localhost";
    int port  = 3333;
    // Протокол  передачи
    // Запрос (3 целых чила): [операция][аргумент 1][аргумент 2]
```

```
    // Ответ (1 целое число): [результат]
    // Операции: 0 - сложение, 1 - умножение
    int operation = 1;
    int arg1 = 5;          int arg2 = 2;
    try   {
       System.out.println("Client is running");
       //  запрос клиента на соединение
       Socket sock = new Socket(host, port);
       // Исходящий поток
       DataOutputStream outStream = new DataOutputStream(
                              sock.getOutputStream());
       // Отправляем запрос и аргументы
       outStream.writeInt(operation);
       outStream.writeInt(arg1);
       outStream.writeInt(arg2);
       // Входящий поток
       DataInputStream inStream = new DataInputStream(
                             sock.getInputStream());
       int d = inStream.readInt();
       System.out.println("Result is " + d);
       // Завершаем потоки и закрываем сокет
       inStream.close();   outStream.close();   sock.close();
    }
    catch(Exception e)   {  System.err.println(e);    }
  }
}
```

Сокет работает как поток ввода-вывода, а значит с ним можно обращаться также как с любым потоком в Java. Сетевые соединения имеют свойство рваться, поэтому хорошая программа должна уметь их восстанавливать с минимальными неудобствами для пользователя.

Для создания серверного сокета в пакете java.net есть класс ServerSocket. В конструкторе этого класса указывается номер порта **ServerSocket(int port).** Основной метод этого класса **accept**() ожидает поступления запроса. Когда запрос получен, метод устанавливает соединение с клиентом и возвращает объект класса socket, через который сервер будет обмениваться информацией с клиентом. Обратите внимание, что для каждого

клиента сервер создает свой поток, в котором происходит их взаимодействие.

Примитивный сервер

```
class Server {
public static void main(String[] args){
   try    {
      System.out.println("Server is running");
      int port  = 3333;
      // создание серверного сокета
      ServerSocket ss = new ServerSocket(port);
      // Ждет клиентов и для каждого создает отдельный поток
      while (true)        {
         Socket s = ss.accept();
         ServerConnectionProcessor p =
                              new ServerConnectionProcessor(s);
         p.start();
      }
   }
   catch(Exception e)    { System.out.println(e);     }
}
}

class ServerConnectionProcessor extends Thread   {
   private Socket sock;
   public ServerConnectionProcessor(Socket s)     {
      sock = s;
   }
   public void run()    {
      try       {
         // Получает запрос
         DataInputStream inStream = new DataInputStream(
                                 sock.getInputStream());
         int operationId = inStream.readInt();
         int arg1 = inStream.readInt();
         int arg2 = inStream.readInt();
         // Выполняет расчет
         int result = 0;
         if (operationId == 0)            { result = arg1 + arg2;   }
         else if (operationId == 1)     { result = arg1 * arg2;   }
```

```
            // Отправляет ответ
            DataOutputStream outStream = new DataOutputStream(
                                    sock.getOutputStream());
            outStream.writeInt(result);
            // Подождем немного и завершим поток
            sleep(1000);
            inStream.close();  outStream.close();   sock.close();
        }
        catch(Exception e)    { System.out.println(e);  }
    }
}
```

Следует отметить, что сетевая программа практически всегда многопточна. Поэтому здесь, как и в любой многопоточной программе, встают вопросы потокобезопасности и синхронизации потоков.

Работа по протоколу UDP

Для посылки дейтаграмм отправитель и получатель создают сокеты дейтаграммного типа. В Java их представляет класс **DatagramSocket**. В классе три конструктора:

DatagramSocket() — создаваемый сокет присоединяется к любому свободному порту на локальной машине;

DatagramSocket(int port) — создаваемый сокет присоединяется к порту port на локальной машине;

DatagramSocket(int port, InetAddress addr) — создаваемый сокет присоединяется к порту port; аргумент addr — один из адресов локальной машины.

Класс содержит массу методов доступа к параметрам сокета и, кроме того, методы отправки и приема дейтаграмм:

send(DatagramPacket pack) — отправляет дейтаграмму, упакованную в пакет pack;

receive (DatagramPacket pack) — дожидается получения дейтаграммы и заносит ее в пакет pack.

При обмене дейтаграммами соединение обычно не устанавливается, дейтаграммы посылаются наудачу, в расчете на то, что получатель ожидает их.

При посылке дейтаграммы по протоколу UDP сначала создается сообщение в виде массива байтов, например,

```
String mes = "This is the sending message.";
byte[] data = mes.getBytes();
```

Потом записывается адрес — объект класса inetAddress:

```
InetAddress addr = InetAddress.getByName (host);
```

Затем сообщение упаковывается в пакет — объект класса DatagramPacket. При этом указывается массив данных, его длина, адрес и номер порта:

```
DatagramPacket pack = new DatagramPacket(data, data.length, addr, port);
```

Далее создается дейтаграммный сокет и дейтаграмма отправляется

```
DatagramSocket ds = new DatagramSocket();
ds.send(pack);
```

После посылки всех дейтаграмм сокет закрывается, не дожидаясь какой-либо реакции со стороны получателя:

```
ds.close ();
```

Прием и распаковка дейтаграмм производится в обратном порядке, вместо метода send() применяется метод receive (DatagramPacket pack).

В примере 7.2 представлен класс, посылающий сообщения на localhost, порт номер 3333. Класс, описанный в примере 4, принимает эти сообщения и выводит их в свой стандартный вывод.

Пример 7. 2. Посылка дейтаграмм по протоколу UDP

```
public class Main   {
    private String host;
    private int port;
```

```
    public Main(String host, int port)   {
       this.host = host;
       this.port = port;
    }
    public void sendMessage(String mes)   {
       try    {
          byte[] data = mes.getBytes();
          InetAddress addr = InetAddress.getByName(host);
          DatagramPacket pack = new DatagramPacket(data, data.length,
addr, port);
          DatagramSocket ds = new DatagramSocket();
          ds.send(pack);
          ds.close();
       }
       catch(Exception e)    { System.err.println(e);  }
    }
    public static void main(String[] args)  {
       System.out.println("Sender is running");
       Main sndr = new Main("localhost", 3333);
       for (int i = 0; i < 10; i++)
            sndr.sendMessage("test message " + i);
    }
}
```

Прием дейтаграмм по протоколу UDP

```
public class Main   {
    public static void main(String[] args)   {
       System.out.println("Receiver is running");
       try    {
          DatagramSocket ds = new DatagramSocket(3333);
          while (true)       {
    DatagramPacket pack = new DatagramPacket(new byte[1024],  1024);
             ds.receive(pack);
             System.out.println(new String(pack.getData()));
          }
       }
       catch(Exception e)   { System.out.println(e); }
    }
}
```

Практические задания

1. Изучить особенности реализации сетевых приложений в Java.
2. Доработать программу, созданную в лабораторных работах № 2-6:

1) создать отдельное консольное приложение, которое будет играть роль TCP-сервера. Клиентом будет приложение, которое было создано в предыдущих работах;

2) TCP-сервер должен ожидать подключения клиентов и выдавать вновь подключенному клиенту список уже подключенных. Необходимо также внести изменения в интерфейс клиента, так чтобы в панели управления отображался список всех подключенных к серверу клиентов. При отсоединении клиентов или при подключении новых список должен обновляться;

3) запрограммировать специальное взаимодействие по TCP с другими клиентами через сервер по варианту;

4) добавить возможность серверу управлять клиентами по протоколу UDP. Пользователь вводит команду в консольное окно сервера, и сервер отправляет команду тому или иному клиенту. Команда задается вариантом. Адреса клиентов можно получить из установленных TCP-соединений, порт UDP не должен совпадать с портом TCP;

5) связь между клиентами осуществляется через сервер. Для действия необходимо предусмотреть элементы управления в интерфейсе или команду в консоле.

Вариант 1

TCP: Реализовать выбор из одного подключенного клиента N случайных объектов и поместить в текущую симуляцию.

UDP: Команда отключения клиента.

Вариант 2

TCP: Реализовать возможность отправить N случайных объектов из текущей симуляции другому подключенному клиенту.

UDP: Команда перезагрузки (отключения, а затем повтороного подключения) клиента.

Вариант 3

TCP: Реализовать возможность получения и установки настроек симуляции таких же как у одного из подключенных клиентов.

UDP: Команда отключения клиента.

Вариант 4

TCP: Реализовать возможность передачи своих настроек симуляции одному из подключенных клиентов.

UDP: Команда перезагрузки (отключения, а затем повтороного подключения) клиента.

Вариант 5

TCP: Реализовать возможность обмена всех объектов одного вида на объекты такого же вида из другого подключенного клиента (например, все легковые машины из одной симуляции переходят в другую, а все легковые машины, что были в той другой симуляции, переходят в первую).

UDP: Команда отключения клиента.

Вариант 6

TCP: Реализовать возможность обмена всех объектов одного вида на объекты другого вида из другого подключенного клиента (например, все менеджеры из одной симуляции переходят в другую, а все разработчики, что были в той другой симуляции, переходят в первую).

UDP: Команда перезагрузки (отключения, а затем повтороного подключения) клиента.

Вариант 7

TCP: Реализовать возможность скопировать и добавить в симуляцию все объекты одного из подключенных клиентов.

UDP: Команда отключения клиента.

Вариант 8

TCP: Реализовать возможность обменяться всеми объектами с одним из подключенных клиентов.

UDP: Команда перезагрузки (отключения, а затем повтороного подключения) клиента.

Вариант 9

TCP: Реализовать возможность синхронной остановки и запуска симуляции на текущем и одном из подключенных клиентов, т.е. при остановке или запуске симуляции выбранный клиент должен также остановить или запустить симуляцию.

UDP: Команда отключения клиента.

Вариант 10

TCP: Реализовать возможность синхронной установки вероятностей появления физических и юридических лиц на текущем и одном из подключенных клиентов, т.е. при установке этих параметров на выбранном клиенте должны выставиться такие же параметры.

UDP: Команда перезагрузки (отключения, а затем повтороного подключения) клиента.

Вопросы для самопроверки

1. Что такое сокеты?
2. Какие типы сокетов существуют, чем они отличаются друг от друга?
3. Какое преимущество имеют потоковые сокеты?
4. Что такое IP-адрес и доменный адрес узла (хоста)?
5. Какой класс Java используется для представления адреса хоста?
6. Какой класс Java предназначен для работы с IP-адресами?
7. Каким образом задается адрес узла при создании объекта, отвечающего за адрес IP?
8. Что такое localhost?
9. Что такое TCP/IP ?
10. Как создается сокетное соединение «сервер-клиент»?
11. Какова последовательность действий приложений Java, необходимая для создания канала и передачи данных между клиентским и серверным приложением?
12. Можно ли оборачивать потоки ввода-вывода сокетов другими потоками ввода-вывода из java.io?

13. Почему сетевые программы в большинстве случаев являются многопоточными?
14. Каковы недостатки и преимущества дейтаграммных сокетов?
15. Что должны сделать приложения для работы с дейтаграммами?
16. Какие методы применяются для посылки и получения дейтаграмм?

ГЛАВА 8. GENERIC-КЛАССЫ В JAVA

Generics – это встроенная в язык особенность, которая позволяет сделать программирование более универсальным и надежным. Является аналогом шаблонных классов в C++. Шабло́ны в C++ - это средства, «предназначенные для кодирования обобщённых алгоритмов, без привязки к некоторым параметрам (например, типам данных, размерам буферов, значениям по умолчанию)» (Википедия). Однако имеют и ряд существенных отличий.

Рассмотрим пример класса Box**. С**оздадим простой класс Box, который управляет объектами разных типов. Он должен иметь только два метода: add, который добавляет объект в коробку, и get, который извлекает его:

```
public class Box {
   private Object object;
   public void add(Object object) {
      this.object = object;
   }
   public Object get() {  return object; }
}
```

Так как эти методы принимают и возвращают ссылку на Object, можно работать с объектами любых непримитивных типов. Однако если нужно ограничить применение нашего класса только для объектов одного типа, например, Integer, можно

только указать это в документации, а компилятор ничего не будет знать об этом:

```
public class BoxDemo1 {
   public static void main(String[] args) {
      // ONLY place Integer objects into this box!
      Box integerBox = new Box();
      integerBox.add(new Integer(10));
      Integer someInteger = (Integer)integerBox.get();
      System.out.println(someInteger);
   }
}
```

Если мы сохраним число 10 как String, то при преобразовании Object в Integer возникнет ошибка. Это совершенно очевидная ошибка, но код откомпилируется без ошибок, однако во время выполнения программыи будет выброшено исключение типа ClassCastException. Если бы класс Box был построен с учетом generics, то эта ошибка была бы обнаружена во время компиляции.

Переделаем класс Box по правилам generics. Создадим объявление типа *generic (generic type declaration),* изменив код "public class Box" на "public class Box<T>"; введя переменную типа (*type variable)* по имени T, которую можно использовать везде внутри класса. Тот же прием может быть применен и к интерфейсам. T – это специальный вид переменной, чье значение – это тип, который может быть любым: интерфейсом, классом, но не примитивным типом данных. Назовем его формальным параметром типа класса Box.

```
public class Box<T> {
   private T t;                    // T stands for "Type"
   public void add(T t) {   this.t = t;   }
   public T get()   {    return t;   }
}
```

Используя класс в программном коде, можно выполнить генерацию класса (*generic type invocation)*, которая заменит тип T на конкретное значение, например, Integer:

```
Box<Integer> integerBox;
```

Здесь не создается новый объект Box. Это простое объявление, что integerBox будет ссылкой на "Box для Integer", и теперь это записывается так: Box<Integer>.

Класс Box называется параметризованным типом (*parameterized type)*. Чтобы создать объект этого класса, используем слово new, как обычно, но вставляем <Integer> между именем класса и скобками:

```
integerBox = new Box<Integer>();
```

После инициализации integerBox можно вызвать метод get без указания приведения типов, как показано в примере BoxDemo2:

```
public class BoxDemo2 {
   public static void main(String[] args) {
      Box<Integer> integerBox = new Box<Integer>();
      integerBox.add(new Integer(10));
      Integer someInteger = integerBox.get(); // no cast!
      System.out.println(someInteger);
   }
}
```

Если Вы попробуете добавить объект типа String, компилятор выдаст сообщение об ошибке.

Важно представлять, что переменные, определяющие тип, не являются самими типами. Нет класса T.java или T.class. T – это не часть класса Box. Фактически во время компиляции вся информация generic удаляется, и остается только класс Box.class.

Заметим, что перечисляться могут несколько параметров типа, но они должны отличаться по именам Box<T,U>.

Методы и конструкторы generic. Параметры типов могут объявляться внутри методов и конструкторов для создания так

называемых generic methods и generic constructors. Это делается похожим способом, но диапазон действия параметра ограничен методом или конструктором, в котором он объявлен.

```
public class Box<T> {
   private T t;
   public void add(T t) {   this.t = t;    }
   public T get() {   return t;    }
   public <U> void inspect(U u){
      System.out.println("T: " + t.getClass().getName());
      System.out.println("U: " + u.getClass().getName());
   }
   public static void main(String[] args) {
      Box<Integer> integerBox = new Box<Integer>();
      integerBox.add(new Integer(10));
      integerBox.inspect("some text");
   }
}
```

Мы добавили один *generic* метод, с именем inspect, который определяет один параметр типа U. Этот метод получает ссылку на объект и выводит его тип, а также выводит тип T.

Более разумное использование generic методов показано ниже. Здесь во все объекты типа Box, входящие в список, добавляется ссылка на один и тот же объект:

```
public static <U> void fillBoxes(U u, List<Box<U>> boxes) {
   for (Box<U> box : boxes) {
      box.add(u);
   }
}
```

Чтобы использовать этот метод, код может иметь следующий вид:

```
Crayon red = ...;
List<Box<Crayon>> crayonBoxes = ...;
```

Вызов статического метода будет следующим:

```
Box.<Crayon>fillBoxes(red, crayonBoxes);
```

Здесь мы явно указали тип для U, но часто его опускают, а компилятор сам определяет нужный тип:

```
Box.fillBoxes(red, crayonBoxes);
```

Эта особенность называется *type inference*, она позволяет вызывать generic метод как обычный, без указания типа в угловых скобках.

Ограниченные параметры типа. Возможно, что Вы захотите ограничить типы, которые можно передавать как параметры типа. Например, метод, который оперирует с числами, захочет принимать только объекты типа Number и его подклассов. Это так называемые *bounded type parameters*.

Для объявления такого типа параметров укажите имя параметра типа, за ним ключевое слово extends Number. Заметим, что extends используется и для классов и для интерфейсов.

```
public class Box<T> {
   private T t;
   public void add(T t) {   this.t = t;    }
   public T get() {    return t;    }
   public <U extends Number> void inspect(U u){
         …
   }
         …
}
```

Теперь при компиляции будет выведено сообщение об ошибке, если inspect вызывается с параметром String.

Если надо указать дополнительные классы или интерфейсы, то используется символ &:

```
<U extends Number & MyInterface>
```

Подтипы. Как известно, возможно назначить ссылку на объект одного типа ссылке на объект другого типа, если эти типы

совместимы. Например, можно присвоить ссылку на Integer ссылке на Object, так как Object – один из базовых типов Integer.

То же самое справедливо и в отношении generics. Можно в качестве типа указать Number, а в методе add будет разрешено использование аргументов, наследующих от него:

```
Box<Number> box = new Box<Number>();
box.add(new Integer(10));                    // OK
box.add(new Double(10.1));  // OK
```

Рассмотрим следующий метод:

```
public void boxTest(Box<Number> n)  { … }
```

Какой тип аргумента он принимает? Мы видим один аргумент – тип его Box<Number>. Можно ли передавать Box<Integer> или Box<Double>? Ответ - "нет", потому что типы Box<Integer> и Box<Double> не являются наследниками Box<Number>.

Понимание проще, если представить реальные картинки, например, клетки для перевозки животных:

```
interface Cage<E> extends Collection<E>;
```

Интерфейс Collection – это корневой интерфейс иерархии классов коллекций; он представляет группу объектов. Клетка – испольуется для хранения набора животных, поэтому мы и наследуем от Collection.

Лев – это животное, поэтому Lion наследует от Animal:

```
interface Lion extends Animal {}
Lion king = ...;
Animal a = king;
```

Посадим его в клетку:

```
Cage<Lion> lionCage = ...;
lionCage.add(king);
```

так же добавим бабочку в клетку с бабочками:

```
interface Butterfly extends Animal {}
```

```
Butterfly monarch = ...;
Cage<Butterfly> butterflyCage = ...;
butterflyCage.add(monarch);
```

А теперь поговорим о клетке для животных вообще:

```
Cage<Animal> animalCage = ...;
```

Здесь можно хранить всех животных. Тогда мы можем поместить туда и льва, и бабочку:

```
animalCage.add(king);
animalCage.add(monarch);
```

Да, лев – животное (Lion – это наследник от Animal), но клетка для льва – это не наследник клетки для животного. Нельзя создать клетку, куда можно было бы поместить и льва, и бабочку. Значит, нет клетки, которую можно рассматривать как некоторую единую:

```
animalCage = lionCage;          // ошибка компиляции
animalCage = butterflyCage; // ошибка компиляции
```

Без generics такое невозможно.

Wildcards (знак вопроса). Фраза "клетка для животных - animal cage" может означать или "клетка для всех животных - all-animal cage", но может, в зависимости от контекста, означать и другое: клетка, предназначенная не для любого вида животных, а для определенного вида животного, вид которого не пока сообщается. В generics такой неоговариваемый тип помечается символом *wildcard* - **"?"**.

Тогда, чтобы указать, что клетка предназначена для определенного типа животных, можно записать:

```
Cage<? extends Animal> someCage = ...;
```

Описание "? extends Animal" определяет неизвестный заранее тип, который наследует от класса Animal или является самим Animal. Это пример *bounded wildcard*, где Animal определяет

ограничение на ожидаемый тип. Теперь мы можем создать клетку для львов или клетку для бабочек.

Можно также указать ключевое слово super вместо extends. Код <? super Animal> читается как тип, являющийся базовым для Animal или сам Animal. Можно просто указать <?>. Это то же самое, что и <? extends Object>.

Так как Cage<Lion> и Cage<Butterfly> не наследуют от Cage<Animal>, они фактически подклассы Cage<? extends Animal>:

```
someCage = lionCage;          // OK
someCage = butterflyCage; // OK
```

Ну а теперь ответим на вопрос: "Можно ли поместить и льва и бабочку в одну клетку someCage?". Ответ будет - "нет".

```
someCage.add(king);           // ошибка компиляции
someCage.add(monarch);        // ошибка компиляции
```

Если someCage – это клета для бабочки, то она позволит выполнить операции для бабочек, а льва туда не поместить. Можно создать метод для того, чтобы животных в данной клетке данного типа покормить:

```
void feedAnimals(Cage<? extends Animal> someCage) {
        for (Animal a : someCage) a.feedMe();
}
```

Тогда можно поместить львов и бабочек в отдельные клетки и покормить их в своих клетках:

```
feedAnimals(lionCage);
feedAnimals(butterflyCage);
```

Или можно поместить всех в одну клетку и кормить вместе:

```
feedAnimals(animalCage);
```

Type erasure. При создании реального типа на основе типа generic компилятор использует прием, называемый *type erasure*

— при этом компилятор удаляет всю информацию, относящуюся к типу и из самого класса и из его методов. Это позволяет поддерживать совместимость с Java – библиотеками, созданными ранее, до generics.

Например, Box<String> транслируется в класс Box, который называется сырым типом (*raw type)* — это класс или интерфейс без параметов типа. Это означает, что нельзя определить во время выполнения, какого типа объект используется в сгенерированном классе. Следующие операции невозможны:

```
public class MyClass<E> {
   public static void myMethod(Object item) {
      if (item instanceof E) {    //Ошибка компиляции
         ...
      }
      E item2 = new E();                    //Ошибка компиляции
      E[] iArray = new E[10];   /Ошибка компиляции
      E obj = (E)new Object(); //Unchecked cast warning
   }
}
```

Выделенные действия невозможны во время выполнения программы, потому что компилятор удаляет всю информацию о реальном типе аргумента, представленного параметром E на фазе компиляции.

Такая реализация позволяет осуществить совместимость со старым кодом. Вообще использование «сырого кода» - это плохая практика программирования, ее обычно следует избегать.

Практические задания

1. Изучить принципы построения классов и методов generic в Java.
2. Разработать программу, использующую классы объектов созданных в лабораторной работе № 2-6:

1) разработать Generic-класс, позволяющий работать с группами объектов по варианту (добавлять, удалять и т.д.). В

качестве параметра класс должен принимать различные объекты (использовать ограничения на тип);

2) разработать метод Generic-класса позволяющий добавить в структуру коллекцию объектов. Протестируйте метод на коллекциях разных видов и посмотрите за ошибками компиляции, которые возникают при неправильном использовании generics;

3) разработать статический generic-метод, который работает с разработанным Generic-классом. Действие, которое должен выполнять метод, задается индивидуальным вариантом. Для всех вариантов для ввода-вывода использовать консоль, созданную в лабораторной работе № 6.

Вариант 1

Структура данных generic-класса стек.

Generic-метод возвращает копию входной группы с инвертированным порядком элементов.

Вариант 2

Структура данных generic-класса циклическая очередь.

Generic-метод принимает на вход 2 группы, объединяет группы и возвращает объединенную группу.

Вариант 3

Структура данных generic-класса односвязный список.

Generic-метод возвращает группу, в которой содержатся четные элементы из входной группы.

Вариант 4

Структура данных generic-класса двусвязный циклический список.

Generic-метод принимает на вход группу, создает ее копию, перемешивает в этой копии элементы случайным образом и возвращает ее.

Вариант 5

Структура данных generic-класса очередь.

Generic-метод удаляет из входной группы все элементы.

Вариант 6

Структура данных generic-класса односвязный циклический список.

Generic-метод возвращает группу, в которой содержатся нечетные элементы из входной группы.

Вариант 7

Структура данных generic-класса двусвязный список.

Generic-метод удаляет из входной группы все четные элементы.

Вариант 8

Структура данных generic-класса односвязный список.

Generic-метод принимает на вход 2 группы, и возвращает группу, которая содержит элементы, присутствующие и в первой, и во второй группе, т.е. возвращает пересечение двух групп.

Вариант 9

Структура данных generic-класса очередь.

Generic-метод удаляет из входной группы все нечетные элементы.

Вариант 10

Структура данных generic-класса двусвязный список.

Generic-метод принимает на вход 2 группы, и возвращает группу, которая содержит уникальные неповторяющиеся элементы из обеих групп (т.е. элементы, содержащиеся в первой группе, но не содержащиеся во второй, или содержащиеся во второй, но не содержащиеся в первой).

Вопросы для самопроверки

1. Назначение generic.
2. В чем сходство и отличие шаблонов в C++ и generic в Java?
3. Что такое неопределенный тип? Приведите примеры его задания и использования.
4. Что такое ограничения на неопределенный тип? Когда их можно использовать приведите примеры.
5. Как можно задать ограничения на неопределенноый тип?

6. Что такое методы generic? Зачем они применяются? Приведите примеры.
7. Каковы сложности при создании нового API с generic?
8. Как переделывать существующее API под generic?
9. Что такое стирание информации о типе при компиляции, и почему это применяется?

СПИСОК ЛИТЕРАТУРЫ

1. И. Хабибуллин. Самоучитель JAVA. 3-е изд. перераб. и доп. – СПб.: БХВ-Петербург, 2008. – 768 с.
2. Ноутон П., Шилдт Г. Java2.: [пер. с англ.] – СПб: БХВ-Петербург, 2000. – 1072 с.
3. Портянкин И. Библиотека программиста. SWING. Эффективные пользовательские интерфейсы. Java Foundation Classes. – СПб.: Питер, 2005. – 336 с.
4. Курс лекций «Программирование на Java». Н.А. Вязовик. http://www.intuit.ru/department/pl/javapl