Федеральное агентство по образованию Российской Федерации
АМУРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Факультет математики и информатики

И.М. Акилова, Л.В. Чепак, Е.Н. Архипова

ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ.

Программирование на языке Java

Допущено учебно-методическим объединением (*УМО*) *вузов*
по университетскому политехническому образованию
в качестве учебного пособия

Благовещенск
2007

ББК

Печатается по решению
редакционно-издательского совета
факультета математики и информатики
Амурского государственного университета

Акилова И.М., Чепак Л.В., Архипова Е.Н.

Технология программирования. Программирование на языке Java: Учебное пособие. Допущено учебно-методическим объединением вузов по университетскому политехническому образованию в качестве учебного пособия для студентов высших учебных заведений, обучающихся по специальности 230102 очной формы обучения.

Благовещенск: Амурский гос. ун-т, 2007.

Пособие рекомендуется студентам, изучающим курс «Технология программирования», а также всем желающим ознакомиться с структурой Java-программы, разработкой простейших апплетов и апплетов двойного назначения, обработкой событий, графикой, анимацией, элементами управления, контейнерами компонентов, реализацией многозадачности в Java, многопотоковыми, автономными и сетевыми приложениями, сокетами и их применением, созданием и использованием сервлетов, работой с файлами и базами данных. Получить навыки программирования на основе приведенных методов.

Рассчитано на преподавателей и студентов.

Рецензенты: В.Д. Епанешников, проф. кафедры автоматики и системотехники ТОГУ, д-р физ.-мат. наук;

А.Н. Рыбалев, доцент кафедры автоматизации производственных процессов и электротехники АмГУ, к.т. наук;

А.Н. Семочкин, начальник управления информационных и телекоммуникационных технологий и информационной безопасности БГПУ, доцент кафедры информатики БГПУ, к.ф.-м. наук.

ВВЕДЕНИЕ

При изучении дисциплины «Технология программирования» акцент делается на изучение процессов разработки программных средств, в которую включены все этапы, начиная с момента зарождения идеи этого средства. Каждый этап этой совокупности базируется на использовании каких-либо методов и средств.

В настоящее время слово Java стало известно практически всем. С одной стороны, язык Java расширяет возможности разработчиков WWW-серверов, а с другой - помогает программисту превратить WWW в платформу программирования. Основным вкладом нового языка является независимый доступ к исполняемому содержимому - для Java-приложений безразлично, на какой платформе оно работает. Визуализация информации при использовании Java становится все более утонченной, позволяя кому угодно с помощью броузера, поддерживающего Java, увидеть вещи, ранее доступные только в лабораториях.

Учебное пособие «Технология программирования. Программирование на языке Java» составлено для студентов специальности «Автоматизированные системы обработки информации и управления» по курсу «Технология программирования».

В данном пособии рассмотрены основные вопросы программирования на языке Java: структура Java-программы, разработка простейших апплетов и апплетов двойного назначения, классы, создание и использование пакетов, обработка событий от мыши и от клавиатуры, графика, цвет, шрифты, анимация, элементы управления и устройства, контейнеры компонентов, реализация многозадачности в Java, многопотоковые, автономные и сетевые приложения, сокеты и их применение, создание и использование сервлетов, работа с файлами и базами данных.

Цели, которые ставит пособие:

1) изучение объектно-ориентированного языка программирования Java;
2) усвоение и закрепление основных приемов и алгоритмов языка;
3) применение навыков программирования для создания программных продуктов с использованием Java-файлов.

В каждой лабораторной работе даются методические указания, содержащие основные теоретические сведения, рассматриваются различные примеры программ, приложений и апплетов. В конце каждой лабораторной работы сформулированы задания для самостоятельного выполнения и контрольные вопросы для самопроверки. В конце методического пособия приведены приложения, содержащие Java-файлы различных апплетов.

ЛАБОРАТОРНАЯ РАБОТА № 1.

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА JAVA, ПРИЛОЖЕНИЯ JAVA (4 часа).

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ

Двумя основными формами Java-программ являются приложение и апплет. Далее рассматриваются различия между этими приложениями и их назначение на основе создания простейшего кода как для приложения, так и для апплета.

Java-программы могут выполняться под управлением специального интерпретатора (java.exe), работающего в рамках отдельного процесса, либо под управлением навигатора Интернет, такого, как Microsoft Internet Explorer или Netscape Navigator. В последнем случае программа называется апплетом.

Java-приложение работают независимо от навигатора, главное их отличие от апплетов лежит в их назначении. Приложения похожи на программы, созданные, например, с использованием языка C/C++, хотя для своей работы они требуют присутствия среды Java. Но, в отличие от апплетов, их существование никак не связано с Internet и они не выполняются как содержимое страниц WWW. Это полноправные приложения, которые существуют и выполняются в локальных компьютерных системах пользователей.

Java-апплеты же разработаны для функционирования в сети и выполняются как часть станиц WWW, поэтому к ним относятся как к исполняемому содержимому. Хотя они и встраиваются в страницы WWW подобно стандартному содержимому, созданному с использованием HTML, на самом деле это программы, которые запускаются и выполняются.

Апплеты требуют наличия соответствующего Java-броузера, так как они должны загружаться по сети с сервера WWW в обеспечивающую их работоспособность среду исполнения Java на локальном компьютере.

1. Простейшее приложение Hello

Перед созданием приложения познакомимся с последовательностью действий для создания приложений:

Использование JDK (Java Developer's Kit).

1. Создание, ввод и сохранение обычного тестового файла, содержащего код программы, имеющего расширение .java (например, Hello.java). Использовать можно любой текстовый редактор, позволяющий работать с файлами, имеющими длинные имена, например Notepad.

2. Компиляция исходного кода Java в машинный байтовый код при помощи компилятора javac. В результате трансляции создаются файлы с расширением .class (Hello.class).

3. Исполнение приложения: передача файла байтового кода интерпретатору java для выполнения приложения.

Замечание. Для выполнения компиляции и запуска приложения можно создать командный файл (с расширением .bat) следующего содержания:

javac.exe Hello.java

java.exe Hello

Использование среды разработки JBuilder.

1. Создание нового Java-проекта с именем Hello (меню “File”, пункт “New Project”)

2. В диалоговом окне “Project Wizard” задаем имя проекта Hello.java в поле “Name”. Нажимаем кнопку “Finish”. В области “Project” появится дерево с именем проекта Hello.java.jpx .

3. Для добавления файла в созданный проект в контекстном меню выбираем “Add Files/Packages”. В диалоговом окне “Add Files or Packeges to Project” в закладке “Explorer” задаем имя файла Hello.java в поле “File name”. Файл Hello.java должен быть включен в проект Hello.

4. Двойной щелчок мыши по имени файла Hello.java в области “Project” открывает рабочую область файла, где в закладке “Sourse” вводится текст программы.

5. Для запуска приложения в области “Project” из контекстного меню файла Hello.java выбираем пункт “Run”.

Приложение Hello

Приступим к созданию простейшего приложения. Java-файл должен содержать следующий код:

```
/*---------- Пример 1. Файл Hello.java ----------*/
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());
   }
} /*-------------------------------------------------------------*/
```

Так как класс Hello объявлен как public, то имя файла, в котором содержится его исходный код, должно совпадать с именем класса. Для классов, не объявленных как public, имена содержащих их исходные тексты файлов могут быть любыми (расширение обязательно .java).

Рассмотрим текст приложения примера 1. В строке

public class Hello {

объявляется новый *класс*, Hello, тело которого начинается с открывающей фигурной скобки и заканчивается закрывающей фигурной скобкой в последней строке исходного текста. Класс по своей сути является *шаблоном*, из которого реализуются объекты Hello.

Java - объектно-ориентированный язык, в котором основными строительными блоками являются объекты. В Java все выражено в объектах. Язык Java не поддерживает глобальные функции и глобальные переменные, а это означает, что все определяется из шаблонов объектов, называемых классами. Класс содержит весь код *состояния* (данные) и *поведения* (методы).

Все классы являются производными (или *подклассами*) от существующих классов. В случае класса Hello не было явно указано, от какого он класса он произошел. В таком случае -если не определен *суперкласс* - по умолчанию предполагается, что таким суперклассом является Object. Для того, чтобы явно задать суперкласс, используется ключевое слово extends, например:

public class Hello extends Object {

В классе Hello объявляется метод main() со строковым параметром args, который будет содержать аргументы командной строки, передаваемые при запуске приложения:

public static void main(String args[]) {

Подобно языку C/C++, в приложение должна быть включена функция main(). Без нее интерпретатор не сумеет понять, откуда начинать выполнение приложение (функция main() является точкой входа приложения). И точно также, как в языках C/C++, Java-приложения могут запускаться *с аргументами командной строки*. Хотя необходимо обязательно включать параметр args в определение метода main(), но использовать аргументы командной строки необязательно. Ключевые слова public и static, называемые *модификаторами доступа***,** рассматриваются ниже.

Метод main() печатает две строки, одна из них “Hello, world”, вторая содержит текущую дату. Метод System.out.println() позволяет выводить информацию на экран. Этот вывод используется обычно в приложениях для текстового вывода, в апплетах же используется графический вывод.

Если методу System.out.println() передать строку символов, заключенную в пару двойных кавычек, этот метод выведет данную строку на экран, завершив ее переводом строки. Кроме того, этот метод можно использовать для печати значений переменных - как по отдельности, так и со строками символов, например:

```
System.out.println("Symbol array");
int i=7; System.out.println(i);
int j=10; System.out.println("j="+i);
```

2. Структура Java-программы

Познакомимся с основными блоками Java-программы. Все рассматриваемые в данном разделе приложения не относятся к апплетам, но все сказанное ниже в равной степени и к апплетам.

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

Методы есть не что иное как функции или подпрограммы. В *переменных* же хранятся данные. Данные понятия присутствуют так или иначе во всех языках программирования. С другой стороны, *классы* представляют собой фундамент объектно-ориентированных свойств языка. Для простоты на данном этапе изучения языка Java можно сказать, что класс - это некое целое, содержащее переменные и методы.

Наконец, *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, пока же, опять-таки для простоты, можно рассматривать просто как наборы классов. Например, приложение Hello импортирует пакет java.util, в котором содержится класс Date.

Вышеперечисленные блоки присутствуют в любой Java-программе. Но программы могут включать в себя составные части и других видов (*интерфейсы, исключения, потоки*), которые не требуются обязательно для каждой программы, однако во многих программам без них не обойтись.

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*). В таком классе обязательно должны быть определены один (для приложений) или несколько (для апплетов) специальных методов. Для приложений первичный класс должен обязательно содержать метод main().

Рассмотрим подробнее каждый из основных блоков Java-программы: переменных, методов, классов и пакетов.

2.1 Переменные

Переменную можно представить как хранилище для единицы данных, имеющее собственное имя. Любая переменная принадлежит определенному *типу*. Тип переменной определяет, какую информацию в ней можно хранить. Переменные должны быть объявлены с использованием следующего синтаксиса (модификаторы рассматриваются ниже в разделе, посвященном классам):

<Модификаторы> ТипПеременной ИмяПеременной ;

В Java существует два вида переменных. Первый - *примитивные типы* (primitive types). К ним относятся стандартные, *встроенные в язык* типы для представления численных значений, одиночных символов и булевских (двоичных, логических) значений. Все примитивные типы имеют *предопределенный размер* занимаемой ими памяти. Ко второму виду переменных - *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.

Примитивные и ссылочные типы также различаются по тому, как переменные этих типов передаются в качестве параметров методам (то есть функциям). Переменные примитивных типов передаются *по значению*, тогда как ссылочные переменные всегда передаются *по ссылке*.

Практически самым важным различием между двумя типами переменных является то, что память для ссылочных переменных выделяется динамически, во время выполнения программы. Использование переменных ссылочных типов требует явного запрашивания требуемого количества памяти для каждой переменной прежде, чем можно будет сохранить в этой переменной какое-либо значение. Причина проста - оболочка времени выполнения сама по себе не знает, какое количество памяти требуется для того или иного ссылочного типа. Рассмотрим в качестве примера использования переменных разных типов следующую программу:

```
/*---------- Пример 2. Файл VarTypes.java ----------*/
class VarTypes
{  public static void main(String args[])
   {
            // переменная примитивного типа
            int varPrimitive;
            // сразу после объявления можно в нее записывать данные
            varPrimitive=1;
            // переменная ссылочного типа
            int varReference[]; // или: int[] varReference;
            // выделение памяти для переменной этого типа
            varReference=new int[3];
            // теперь можно сохранять данные в переменной этого типа
            varReference[0]=2;
            varReference[1]=3;
            varReference[2]=4;
```

```
        System.out.println("varPrimitive="+varPrimitive);
        System.out.println("varReference[0]="+varReference[0]);
        System.out.println("varReference[1]="+varReference[1]);
        System.out.println("varReference[2]="+varReference[2]);
    }
} /*-------------------------------------------------------------*/
```

Так как тип int относится к примитивным типам, оболочка выполнения знает, сколько места необходимо выделить такой переменной (четыре байта). Однако при объявлении *массива* переменных типа int, оболочка выполнения не может знать, сколько места потребуется для хранения этого массива. Поэтому прежде, чем разместить что-либо в переменной ссылочного типа, необходимо запросить у системы определенное количество памяти под эту переменную. Этот запрос осуществляется с помощью оператора new.

Переменные-массивы и другие ссылочные переменные лишь указывают на то место в памяти, где содержатся собственно данные, тогда как переменные примитивных типов ни на что не указывают, а просто содержат в себе соответствующие данные, имеющие определенный фиксированный размер.

Ссылочные переменные, хоть и очень похожи на указатели C/C++, имеют сильное отличие от них. Используя ссылочные типы, нельзя получить доступ к фактическим адресам данным в памяти.

2.1.1 Примитивные типы

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

Булевские (логические, двоичные) переменные могут иметь одно из двух допустимых значений: true или false. Двоичные константы являются именно константами, а не строками. Их нельзя преобразовать в строковый тип. Они также не являются целыми значениями нуль или единица, как двоичные значения в языках C/C++. Рассмотрим примеры использования булевских переменных:

```
boolean b1=true, b2, b3;
System.out.println(b1); // печать: true
b2=(25==25); // b2 равно true
b2=(25==10); // b2 равно false
b3=(b1&b2); // b3 равно false
```

System.out.println("b1&b2="+b3); // печать: b1&b2=false

b3=(b1!=b2); // b3 равно true

System.out.println("b1 != b2 - "+b3); // печать: b1 != b2 – true

Булевским переменным можно присваивать не только булевские константы, но и результаты сравнения переменных различных типов. Операции !, !=, == работают с булевскими значениями так же, как одноименные операторы работают с целочисленными значениями в языке C/C++.

2.1.2 Ссылочные типы

Ссылочные типы отличаются от примитивных тем, что они не определены в самом языке Java, и поэтому количество памяти, требуемое переменных этих типов, заранее знать невозможно. Пример одного из ссылочных типов - это тип массива. Массивы языка Java могут состоять из переменных любых типов, включая типы, определенные пользователем

Познакомимся с некоторыми терминами, относящимся к работе с переменными ссылочных типов. Когда для переменной ссылочного типа выделяется память при помощи оператора new, то тем самым этот ссылочный тип реализуется**.** Таким образом, каждая переменная ссылочного типа является реализацией, объектом или экземпляром соответствующего типа.

Язык Java не позволяет просто объявить переменную ссылочного типа и сразу же начать записывать в нее значение. Необходимо сначала запросить у оболочки времени выполнения некоторый объем памяти, а оболочка, в свою очередь, должна сделать запись в своих внутренних таблицах, что активизирована переменная данного ссылочного типа. Весь этот процесс в целом и называется *реализацией* переменной. После реализации, когда уже имеется экземпляр переменной данного типа, можно использовать этот экземпляр для хранения данных. Важно понимать, что экземпляр переменной и сам ссылочный тип, к которому эта переменная относится, являются качественно различными понятиями - для хранения переменной можно использовать только реализованный экземпляр переменной ссылочного типа.

Типы, определенные пользователем

Язык Java позволяет определять новые типы помощью новых классов, а также с помощью интерфейсов (речь о них пойдет позже). Для простоты можно сказать, что классы похожи на структуры (или записи) языка C - они тоже позволяют хранить наборы переменных разных типов. Но в отличие от структур, классы помимо переменных могут хранить и методы.

Рассмотрим пример определения и использования нового класса (нового типа) MyType:

```
/*------------- Пример 3. Файл NewClass.java -------------*/
/*------------- 3.1. Объявление нового типа -------------*/
class MyType // объявляется класс
{  public int myData=5; // переменная-элемент класса
   public void myMethod() // метод класса
   {           // печать в методе
               System.out.print("myMethod!");
               System.out.println(" myData="+myData);
   }
   MyType() // конструктор без параметров
   {           // печать в конструкторе
               System.out.println("Constructor without parameters");
   }
   MyType(int v) // конструктор с одним параметром
   {           // печать в конструкторе
               System.out.print("Constructor with one parameter");
               System.out.println(" Setting myData="+v);
               myData=v;
   }
}
/*------------- 3.2. Реализация объектов и действия с ними -------------*/
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);
   }
} /*-------------------------------------------------------------*/
```

В примере 3.1 определяется новый тип данных, а пример 3.2 иллюстрирует три основных вида действий, которые можно производить с объектом: *создание* объекта (реализация класса), доступ к *переменным-элементам* и доступ к *методам* этого объекта (через оператор “точка” (.)).

Поскольку тип myType является ссылочным типом, то для реализации класса необходимо использовать оператор new. Этот оператор запрашивает память у системы для хранения объекта. Кроме того, можно определить, какие еще действия должны выполняться в момент реализации класса, определив так называемый *конструктор* (constructor) - метод, имеющий такое же имя как и класс**.** В классе может быть не один конструктор, они должны отличаться списком аргументов.

Конструкторы можно использовать для инициализирующих действий, например для присвоения начальных значений. Можно использовать все определенные в классе конструкторы (с параметрами и без них), вызывая их при создании объектов класса. Если в классе отсутствует определение конструктора без параметров, то при использовании следующего оператора

MyType obj1=new MyType();

вызывается конструктор без параметров по умолчанию.

Нужно отметить, что название “типы, определенные пользователем” не подразумевает, что каждый пользователь сам должен определять для себя типы. В состав интерфейса прикладного программирования (Application Programming Interface) входят десятки стандартных классов, которые можно использовать в своих программах. Такие классы называются *стандартными типами, определенными пользователем***.**

Тип String (тип строковых переменных)

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

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

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;
// 1-ый способ инициализации строковой переменной
String S1=S0+"myInt"+myInt;
// 2-ый способ инициализации строковой переменной
String S2=new String(S0+"myInt"+myInt);
```

Несмотря на поддержку таких операций с примитивными типами как = и +, строковые переменные типа Strintg являются в то же время и объектами, так что для них можно вызывать методы класса String, например, узнать длину строки:

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

Итак, реализация переменных типа String не требует применения оператора new. Однако при программировании необходимо всегда помнить о том, что тип String является особым - это единственный определенный пользователем тип, 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 над любым существующим типом, включая типы, определенные пользователем. Кроме того, можно пользоваться и массивами массивов или многомерными массивами.

Иначе говоря, если можно создать переменную некоторого типа, значит можно и создать массив переменных этого типа. Создание массивов требует использования оператора new, например:

```
// объявление ссылочной переменной типа массива
int intArray[];
// реализация или создание массива переменных целого типа
intArray=new int[3];
// объявление ссылочной переменной типа массива
MyType objArray[];
// создание массива ссылочных переменных типа MyType
objArray=new MyType[3];
```

Оператор new запрашивает для массива необходимую память. При создании ссылочной переменной типа массив не нужно объявлять размер массива, это делается при реализа-

ции типа массива при помощи оператора new. Доступ же к элементам массива ничем не отличается от доступа к элементам массива в C/C++:

```
// доступ к элементам
intArray[0]=1; intArray[1]=2; intArray[2]=3;
// доступ к переменным и реализация объектов типа MyType
objArray[0]=new MyType();
objArray[1]=new MyType();
objArray[2]=new MyType();
// доступ к элементам объектов, объекты - элементы массива
objArray[0].myData=0; objArray[0].myMethod();
objArray[1].myData=0; objArray[1].myMethod();
objArray[2].myData=0; objArray[2].myMethod();
```

Массивы Java имеют три важных преимущества перед массивами других языков. Во-первых, программисту не обязательно указывать размер массива при его объявлении. Во-вторых, любой массив в языке Java является переменной - а это значит, что его можно передать как параметр методу и использовать в качестве значения, возвращаемого методом. И в-третьих, не составляет никакого труда узнать, каков размер данного массива в любой момент времени через специальную переменную length, имеющуюся в каждом массиве, например:

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

2.2 Методы

Метод представляет собой подпрограмму, аналогичную функциям языка C и Pascal. Каждый метод имеет тип возвращаемого значения и может вызываться с передачей некоторых параметров.

Объявление метода имеет следующий синтаксис (модификаторы рассматриваются ниже в разделе, посвященном классам):

```
<Модификаторы> ТипВозврЗначения ИмяМетода(<Параметры>)
{
        ТелоМетодаСодержащееОбъявлениеПеременных&Операторы
}
```

С каждым методом должен быть соотнесен тип возвращаемого им значения. Например, тип void является специальным способом указать системе, что данный метод не возвращает никакого значения. Методы, у которых возвращаемое значение принадлежит к любому другому типу, кроме void, должны содержать в своем теле оператор return. Возвращаемое значение может принадлежать к любому из типов, включая примитивные типы и ссылочные типы.

В качестве параметров в языке Java можно передавать переменные любого типа, включая типы, определенные через классы, и массивы переменных любого типа и размера. Однако в качестве параметров переменные примитивных типов ведут себя иначе, чем переменные ссылочных типов.

Все переменные примитивных типов передаются методам по значению (by value). Это означает, что в момент вызова метода делается копия переменной, передаваемой методу. Если метод в своем теле будет изменять значение переданной ему в качестве параметра переменной, то содержимое исходной переменной изменяться не будет, так как все действия будут производится с ее копией.

Напротив, значения переменных ссылочного типа, переданных в качестве параметров, можно изменить в теле метода. Когда методу в качестве параметра передается переменная ссылочного типа, то при изменении ее значения явным образом меняется содержимое того, на что указывает эта переменная.

Замечание. Несмотря на то, что тип строковых переменных (тип String) является определенным пользователем типом, он не ведет себя как ссылочный тип при передачи параметров. Переменные типа String в качестве параметров всегда передаются по значению, - то есть метод, получив строковую переменную в качестве параметра, в своем теле будет фактически работать с копией этой строковой переменной. Иначе говоря, изменение строковой переменной в теле метода не влияет на значение этой же переменной снаружи метода.

Иногда возникает необходимость создавать две или несколько функций, выполняющих по сути, одни и те же действия, но имеющие различные списки параметров. В языке Java можно присвоит одно и то же имя нескольким методам, которые различаются списками своих параметров. Этот процесс называется *совмещением методов*.

Выгоды совмещения методов особенно очевидны в том, вместо запоминания нескольких имен разных методов можно ограничиться запоминанием одного только имени, общего для всех методов с разными параметрами. В обязанности компилятора входит выяснение того, какой именно метод требуется вызвать в каждом случае.

Приведем пример, в котором показаны: методы, которые возвращают значение, и не возвращают никакого значения; методы, работающие с переменными различных видов в качестве параметром; совмещенные методы. Все методы и переменная класса объявлены как static для того, чтобы можно было ими пользоваться без создания объекта класса.

```
/*---------- Пример 4. Файл TestMethods.java ----------*/
class TestMethods
{
   static int v=0;
```

```
// функция, не возвращающая значение
static void setV(int i)
{		v=i;
		System.out.println("Void method!");
}
// фунция с возвращаемым значением
static int getV()
{		System.out.println("Returning method!");
		return v;
}
// функция для проверки работы с параметрами
static int func(int a, int b[])
{		a=a+1;
		b[0]=b[0]+1;
		System.out.println("a="+a+" b[0]"+b[0]); // a=2 b=2
		return a; // return 2
}
// сравнение двух целых
static String compare(int i,int j)
{		if(i==j)	return ""+i+" and "+j+" are equal";
		else
			if(i>j)		return ""+i+" greater than "+j;
			else		return ""+j+" greater than "+i;
}
// совмещение метода с другим числом параметров
static String compare(int i,int j,int k)
{		String S="";
		S=S+compare(i,j)+'\n';
		S=S+compare(i,k)+'\n';
		S=S+compare(j,k);
		return S;
}
// совмещение метода с параметрами другого типа
static String compare(double i,double j)
{		if(i==j)	return ""+i+" and "+j+" are equal";
```

```
            else
                        if(i>j)     return ""+i+" greater than "+j;
                        else        return ""+j+" greater than "+i;
    }
    // главный метод - точка входа
    public static void main(String args[])
    {           // вызов метода, не возвращающего значения
                setV(5);
                // вызов метода, возвращающего значение
                int vv=getV();
                // передача параметров: примитивные и ссылочные типы
                int A; int B[]=new int[1];
                A=1; B[0]=1;
                System.out.println("A="+A+" B[0]="+B[0]); // A=1 B[0]=1
                int aa=func(A,B);
                System.out.println("aa="+aa);
                System.out.println("A="+A+" B[0]="+B[0]); // A=1 B[0]=2
                // вызов совмещенных методов
                String S;
                S=compare(2,5); System.out.println(S);
                S=compare(3,1,6); System.out.println(S);
                S=compare(1.5,2.1); System.out.println(S);
    }
} /*---------------------------------------------------------------------*/
```

2.3 Классы

Классы лежат в фундаменте объектно-ориентированных свойств языка Java, рассмотрим их подробнее.

2.3.1 Статические и динамические элементы (модификатор static)

В Java переменные и методы класса могут быть либо *элементами класса*, либо *элементами экземпляра* класса, в котором они объявлены, что определяется присутствием или отсутствием модификатора static**.**

Если при определении элемента не используется ключевое слово static, то это элемент по умолчанию является *динамическим* (dynamic). Динамические методы и переменные всегда являются *элементами экземпляра класса*, и доступ к ним осуществляется через переменную-объект.

Статические методы и переменные связаны с классом, а не с экземпляром класса, и поэтому имеют название *элементов класса*. Элементы класса можно использовать без создании объекта этого класса, доступ осуществляется через имя класса. Каждая *переменная класса* и каждый *метод класса* уникальны *в своем классе*; *методы* и *переменные экземпляра* уникальны в своем экземпляре класса. Различие между членами класса и экземпляра значительно, в особенности, если дело касается переменных

Элементы класса, будучи уникальными в своем классе, используются всеми объектами, созданными из этого класса, - то есть все объекты, созданные из данного класса, разделяют статические переменные и методы, определенные в этом классе (для всех объектов существует только один экземпляр статической переменной). *Элементы экземпляра класса*, с другой стороны, создаются каждый раз, когда создается объект.

Элементы класса могут, таким образом, считаться глобальными относительно класса, несмотря на то, что настоящие глобальные переменные в Java не поддерживаются. Когда какой-нибудь объект класса изменяет значение *переменной класса*, результат становится видим всем объектам. Благодаря этому переменные класса часто используют в качестве общих данных для всех объектов, созданных из этого класса.

Чтобы обратиться к *методам и переменным экземпляра (динамическим элементам)*, объект надо сначала реализовать из класса, после чего можно получить к ним доступ, используя синтаксис (запись с точкой):

ИмяОбъекта.ИмяМетодаЭкземпляра(<Параметры>)

ИмяОбъекта.ИмяПеременнойЭкземпляра

А для использования статических методов и переменных (элементов класса) объект можно не создавать, а пользоваться записью следующего вида:

ИмяКласса.ИмяМетодаКласса(<Параметры>)

ИмяКласса.ИмяПеременнойКласса

Замечание. Нужно отметить, что *статические методы* класса могут работать только со *статическими переменными* класса, для которых также не важно наличие реализованного объекта.

Рассмотрим пример, в котором сначала определяется класс, включающий статические и динамические методы, а затем его методы используются в функции main():

```
/*---------- Пример 5. 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;
    }
}
class TestElements
{   public static void main(String args[])
    {       int ii,jj;
            // для использования элементов класса объект необязателен
            // 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);
            // перед использованием элементов экземпляра
            // требуется реализовать объект (экземпляр)
            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);
        // вызов метода, в котором используются
        // динамическая и статическая переменные
        ii=obj.summa();
        System.out.println("Main, summa="+ii);
    }
} /*---------------------------------------------------------------------*/
```

Замечание. Теперь понятно, почему при объявлении метода main всегда используется ключевое слово static. Дело в том, что при загрузке первичного класса в память он загружается в виде *типа*, а не в виде *объекта*. После этого оболочка времени выполнения просто вызывает *статический метод* main() этого первичного класса.

2.3.2 Модификаторы доступа

Модификаторы доступа используются для управления доступностью элементов класса из других частей программы (в других классах). Применение ключевых слов public (открытый), protected (защищенный), private protected и private (закрытый) к элементам помогает управлять способностью других объектов пользоваться ими.

Элемент, объявленный с ключевым словом *public*, доступен во всех классах как в том пакете (о пакетах классов см. ниже), в котором он был объявлен, так и во всех классах в любом другом пакете. Из всех модификаторов данный накладывает наименьшие ограничения на доступность элемента- он доступен всем, является открытым для всех. Этот модификатор, в отличие от всех остальных, можно использовать и *при объявлении класса* (класс может иметь этот модификатор или не иметь). Тогда этот класс также доступен для всех других классов невзирая на то, частью какого пакета классов он является. В каждом файле должен содержаться только один открытый класс.

Элемент, объявленный с модификатором *protected* в некоем классе А, доступен во всех классах в данном пакете, а также во всех классах, являющихся подклассами класса А. Иными словами, доступа к этому элементу нет в тех классах, которые не входят в данный пакет и не являются подклассами того класса, в котором этот элемент определена.

Если же элемент в классе А объявлен как *private protected*, то это означает, что к нему можно получить доступ только в подклассах класса А. В других же классах, даже входящих в этот же пакет, этот элемент недоступен.

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

Модификатор доступа (private, protected и private protected) используется для так называемого скрытия данных (data hiding). Объявление переменных или методов класса с закрывающим их модификатором делает невозможным их использование вне области доступности. Методы же их собственного класса имеют к ним доступ всегда. Поэтому, например, для доступа к закрытым переменным для установки из значений или получения значений, содержащихся в них, можно использовать специально созданные для этого незакрытые методы класса (если они конечно, существуют).

Приведем пример, в котором определяется класс, в котором есть открытая и закрытая переменные, и показываются методы доступа к этим переменным вне их класса.

```
/*---------- Пример 6. Файл TestModificators.java ----------*/
class A
{  public int k; // тип public - - доступ и в теле класса и по объекту класса
   private int n; // тип private - доступ только в теле класса
   A() { k=2; n=11; } // конструктор, инициализация переменных-элементов
   int summa() { return k+n; } // метод класса, использующий обе переменные
   // методы для доступа к защищенному элементу n класса A
   // по объекту класса A вне тела класса A
   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);
              // установить значения переменных и
              // вызвать метод, использующий
              //закрытую и открытую переменные
              obj.k=10;              obj.setN(15);
              int s=obj.summa(); System.out.println("summa="+s);
   }
```

}

2.3.3 Наследование классов

Модификаторы доступа делают классы более устойчивыми и надежными в работе, так как гарантируют, что снаружи класса можно будет получить доступ только к некоторым из методов и переменных. *Наследование* (inheritance), в свою очередь, упрощает практическое использование классов, так как позволяет *расширять* уже написанные и отлаженные классы, *добавляя* к ним новые свойства и возможности.

Например, вместо создания класса “с нуля”, можно значительно выиграть во времени, используя механизм наследования переменных и методов, определенных в другом классе, обладающем основными необходимыми свойствами. Таким образом создается то, что называется *подклассом* первоначального класса. Класс, который при этом наследуется (расширяется), называется *суперклассом***.** При расширении какого-либо класса имеется возможность использования всех написанных и отлаженных методов этого класса - их не нужно создавать заново для подкласса. Это свойство, называемое *повторным использованием кода* (code reuse), является одним из главных преимуществ объектно-ориентированного програмирования.

Для того, чтобы наследовать свойства существующего класса, класс должен быть явно определен с ключевым словом extends (подкласс расширяет суперкласс). Например, объявления класса MyClass подклассом класса SuperClass, можно записать так:

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

Если класс явно не объявляется подклассом какого-либо класса, компилятор предполагает по умолчанию его подклассом класса Object.

2.3.4 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, 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 класс имеет значение переменной null и говорят, что объект равен нулю. Если объект равен нулю, доступ к его элементам не разрешен, потому что не создан объект, с которым могли бы ассоциироваться эти элементы. Если попытаться обра-

титься к элементам до того, как создан объект, то вызовется исключение, которое остановит выполнение программы. Например, определим метод класса ClassB, использующий передаваемый ему объект класса ClassA:

```
piblic void SomeMethodClassB(ClassA ObjClassA)
{
        ObjClassA.SomeMethodClassA();
}
```

Следующий фрагмент метода класса ClassB, в котором вызывается метод SomeMethodClassB, приведет к исключению NullPointerException, потому что не создан SomeMethodClassA:

```
ClassA ObjectClassA;
SomeMethodClassB(ObjectClassA);
```

Чтобы уберечь программу от сбоев, необходимо перед использованием объектов проверять их на равенство. Изменим метод SomeMethodClassB:

```
piblic void SomeMethodClassB(ClassA ObjClassA)
{
        if(ObjClassA==null)     { System.out.println("А пусто!!!"); }
        else                    { ObjClassA.SomeMethodClassA(); }
}
```

Переменная this

Иногда бывает необходимо передать другому методу ссылку на текущий объект. Это можно сделать, просто передав переменную особую ссылочную переменную this.

Кроме того, переменная this используется для ссылок на переменную экземпляра. Например, если на уровне класса объявлена переменная-элемент с именем *ИмяПеременной*, то ничто не мешает объявить локальную переменную с тем же именем *ИмяПеременной* внутри какого-либо из методов класса, которая скроет вышестоящую переменную и сделает ее недоступной (скрытие переменной). Внутри метода идентификатор *ИмяПеременной* будет относиться именно к локальной переменной, объявленной в этом методе. Однако существует возможность получить доступ к переменной-элементу, даже если она скрыта. Для этого нужно воспользоваться переменной this. 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, а не к методу:

```
this.ИмяПеременной
```

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

this(<ПараметрыДругогоКонструктораКласса>);

Переменная super

Другая специальная переменная языка Java 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 таким же именем, что и имя переменной его суперкласса. Например, в классе объявлена переменная с именем *ИмяПеременной* и в его подклассе также объявлена переменная с тем же именем *ИмяПеременной.* Для осуществления в подклассе доступа к такой переменной суперкласса необходимо использовать переменную super:

super.ИмяПеременной

При *замещении метода* в классе создается метод, имеющий то же самое имя и список параметров, что и метод в суперклассе. Таким образом метод суперкласса замещается методом подкласса. Однако можно использовать переменную super 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.

Если возникает необходимость вызвать конструктор суперкласса из конструктора подкласса, то это можно сделать, используя ключевое слово super. Для того, чтобы вызвать из конструктора подкласса конструктор суперкласса, нужно использовать следующую запись:

super(<ПараметрыКонструктораСуперкласса>).

В том конструкторе подкласса, в котором используется подобный вызов конструктора суперкласса, этот оператор должен быть первым исполняемым оператором.

Замечание. Не следует считать, что переменная super указывает на совершенно отдельный объект. Чтобы ее использовать, не нужно реализовывать суперкласс. На самом деле это просто способ обращения к переменным-элементам суперкласса и способ выполнения методов и конструкторов, определенных в суперкласса

2.4 Пакеты и импортирование

Классы являются основными строительными блоками любой Java-программы. В сравнении с классами *пакеты* выполняют чисто утилитарную функцию. Они просто содержат в себе наборы классов (а также исключения и интерфейсы). Кроме того, пакеты позволяют определять *защищенные* (protected) элементы классов, которые доступны всем классам, входящим в один и тот же пакет, но недоступны каким бы то ни было классам за пределами такого пакетов.

2.4.1 Использование пакетов

Основное назначение пакетов - создание библиотек кода, пакеты используются для организации и категоризации классов. Стандартные пакеты Java состоят из восьми пакетов API и пакета, содержащего классы и интерфейсы для отладки.

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 строка программы. Например:

*import java.applet.**

*import java.awt.**

import java.net.URL

При этом импортируется три пакета (если точнее, то открытые - public - классы четырех пакетов) с использованием двух способов. В первых двух строках программы использовалась (*), чтобы указать на возможность доступа ко всем открытым классам в пакете (тут нужно отметить, что классы подпакетов импортируются отдельно, с указанием отдельного оператора import). В последней строке явно установлено имя открытого класса для импортирования.

Надо заметить, что несмотря на то, что в первых двух строках указывается импортировать *все* классы пакета, на самом деле импортируются только классы, которые *используются* в программе; язык Java компонует классы, только если они действительно используются.

После того, как пакет импортирован, классы можно использовать просто по именам, без указания на принадлежность тому или иному пакету, например:

ИмяКлассаПакета Obj=new ИмяКлассаПакета(<Параметры>);

При использовании класса из пакета необходимо или импортировать пакет, частью которого он является, или сослаться на него в программе, как это описано ниже. Однако нет необходимости беспокоиться об импортировании пакета java.lang, так как эти классы - базовые классы Java и автоматически импортируются независимо от того, заданы они или нет.

Явные ссылки на классы пакетов

Если пакет импортирован, то его классы могут использоваться столько раз, сколько требует программа. В случае, когда класс необходимо использовать класс только один раз, можно явно сослаться на класс, не импортируя пакет, частью которого он является. Для этого перед именем используемого класса просто указывается имя пакета, в котором находится этот класс, например:

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

Способ явной ссылки на класс из пакета более удобен для чтения и понимания, откуда появился используемый класс, чем метод предложения импорта пакета, но менее удобен для кодирования, так как ссылка получается достаточно длинной.

Конфликты именования

Одним из достоинств явной ссылки на классы пакетов, помимо ясности самого текста программы, является то, что при этом имена классов не конфликтуют (что возможно при использовании предложения импорта).

Если программа импортирует два пакета и в каждом объявлены классы с одинаковыми именами, то непонятно, к примеру, по классу из какого пакета будет создаваться экземпляр класса. Это определить невозможно, поэтому компилятор выдаст ошибку и прервет компиляцию. Однако можно предотвратить этот потенциальный конфликт с помощью явного объявления имени пакета вместе с именем классов.

Уровни пакетов

Имена пакетов упорядочены по уровням, каждый уровень отделен от следующего. Например, пакет java.applet включает два уровня. Первый уровень - java, второй - applet. Некоторые стандартные пакеты API имеют и три уровня.

При создании пакетов разработчик должен сам принимать решение относительно имен и числа уровней пакетов.

2.4.2 Создание пакетов

Для создания пакета (т.е. добавления класса в пакет), просто используется следующее предложение в качестве первой строки программы (даже раньше предложения import) в исходном тексте программы:

package ИмяПакета

Содержимое пакета может хранится 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 каталоге, соответствующем имени пакета, все точки в котором будут заменены на символ /.

Уникальные имена.

Соглашение об именовании, используемое при создании пакетов, разработано таким образом, чтобы обеспечить уникальность имен пакетов при единообразии доступа к ним и не привести к конфликту имен. Все пакеты, которые создаются программистами и которые будут использоваться вне их организации, должны использовать имена Internet-доменов организации, записанные в обратном порядке, чтобы гарантировать уникальность имен.

Например, для того чтобы пакет хранился на сервере (домене) vvsu.ru и был широко доступен, при создании его надо снабдить именем домена, записанным в обратном порядке, например:

RU.vvsu.mypackage; // этот пакет уникальный

Как видно, первая часть уникального имени должна состоять из прописных букв, в оставшейся части используются строчные буквы.

Например, определим в пакете mypackage класс MyClass, тогда ссылка на него записывается следующим образом:

RU.vvsu.mypackage.MyClass;

В соответствии с соглашением об уникальности именования пакетов можно сразу сказать, что источником пакета mypackage является домен vvsu.ru. А так как только имена классов начинаются с прописной буквы, то очевидно, что MyClass - это класс, а не очередной уровень пакета mypackage.

Задания к лабораторной работе

Задание 1. Изучить основные понятия и термины *Java.*

Задание 2. Проверить и объяснить работу всех приложений, рассматриваемых в данной главе и отмеченных курсивом. Должны быть созданы следующие приложения *Hello* (пример 1), *VarTypes* (пример 2), *NewClass* (пример 3), *TestMethods* (пример 4), *TestElements* (пример 5), *TestModificators* (пример 6).

Задание 3. Дать ответы на контрольные вопросы.

Контрольные вопросы

1. Чем отличаются Java-приложения и Java-апплеты?

2. Какие основные составные части должны присутствовать в каждой Java-программе, их функции (назначение)?

3. Что такое первичный класс приложения? Какой обязательный метод он должен содержать?

4. Какие существуют виды переменных Java, чем они отличаются друг от друга?

5. Какие примитивные типы определены в Java, особенности булевского типа?

6. Что называется процессом реализации ссылочного типа?

7. Что делает конструктор класса? Должен ли он обязательно явно присутствовать в объявлении класса?

8. Какие существуют виды ссылочных типов?

9. Что такое типы, определенные пользователем?

10. Что такое стандартные типы, определенные пользователем?

11. В чем особенности строковых переменных?

12. Чем массивы Java отличаются от массивов других языков, их преимущества?

13. Как переменные различных видов передаются в качестве параметров методам?

14. Как ведут себя строковые переменные при передачи их в качестве параметров?

15. Что такое совмещение методов?

16. 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?

17. Для чего используются модификаторы доступа? Какие существуют модификаторы доступа, как они ограничивают доступ к элементам?

18. Что позволяет делать процесс наследования?

19. Что такое суперкласс и подкласс?

20. Что такое повторное использование кода?

21. Какие заранее определенные переменные содержит каждый класс Java?

22. Что можно сделать при помощи переменной this? Что можно сделать при помощи переменной super?

23. Что такое скрытие переменной, затемнение переменной и замещение метода?

24. Как импортировать классы из пакетов?

25. Как использовать явные ссылки на классы из пакетов?

26. Как добавить класс в пакет?

ЛАБОРАТОРНАЯ РАБОТА № 2.

СОЗДАНИЕ ПРОСТЕЙШИХ АППЛЕТОВ (4 часа).

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ

Как уже говорилось раньше, простые Java-приложения выполняются с помощью интерпретатора Java, не нуждаясь при этом ни к каком Java-броузере Они подобны обычным исполняемым файлам. Апплеты же выполняются под управлением виртуальной машины Java, встроенной в WWW-навигатор. Апплет встраивается в документ HTML (HTML - язык для разметки гипертекста на WWW-страницах) и выглядит как окно заранее заданного размера. Он может рисовать в своем окне (и только в нем) произвольные изображения или текст.

Двоичный файл с исполняемым (а точнее говоря, интерпретируемым) кодом Java располагается на сервере WWW. В документ HTML с помощью специального оператора организуется ссылка на этот двоичный файл. Когда пользователь загружает в навигатор документ HTML с апплетом, файл апплета переписывается с сервера WWW на локальный компьютер пользователя. После этого навигатор начинает его выполнение.

Поскольку апплеты встраиваются в WWW-страницы, их разработка включает в себя несколько новых этапов, которых не было в рассматриваемом ранее цикле “редактирование - компиляция - запуск”, на котором строится разработка обычных программ. Перед созданием апплета познакомимся с последовательностью действий для его создания:

Использование JDK (Java Developer’s Kit).

1. Создание, ввод и сохранение обычного тестового файла, содержащего код программы, имеющего расширение .java (например, Hello.java). Использовать можно любой текстовый редактор, позволяющий работать с файлами, имеющими длинные имена, например Notepad.

2. Создание с помощью того же текстового редактора файла HTML (например, Hello.html), в который встраивается созданный апплет. Для этого в него включается следующий тег:<APPLET CODE=Hello.class WIDTH=200 HEIGHT=200></APPLET>

3. Компиляция исходного кода Java в машинный байтовый код при помощи компилятора javac. В результате трансляции создаются файлы с расширением .class (Hello.class).

4. Исполнение приложения:

1) Просмотр файла Hello.html с помощью WWW-навигатора, поддерживающего работу апплетов.

2) Либо использование программы просмотра апплетов appletviewer.exe.

Замечание. Для выполнения компиляции и запуска приложения можно создать командный файл (с расширением .bat) следующего содержания:

javac.exe Hello.java

appletviewer.exe Hello.html

Использование среды разработки JBuilder.

6. Создание нового Java-проекта с именем Hello (меню “File”, пункт “New Project”).

7. В диалоговом окне “Project Wizard” задаем имя проекта Hello.java в поле “Name”. Нажимаем кнопку “Finish”. В области “Project” появится дерево с именем проекта Hello.java.jpx .

8. Для добавления файла в созданный проект в контекстном меню выбираем “Add Files/Packages”. В диалоговом окне “Add Files or Packeges to Project” в закладке “Explorer” задаем имя файла Hello.java в поле “File name”. Файл Hello.java должен быть включен в проект Hello.

9. Двойной щелчок мыши по имени файла Hello.java в области “Project” открывает рабочую область файла, где в закладке “Sourse” вводится текст программы.

10. Для добавления HTML-файла в созданный проект в контекстном меню выбираем “Add Files/Packages”. В диалоговом окне “Add Files or Packeges to Project” в закладке “Explorer” задаем имя файла Hello.html в поле “File name”. Файл Hello.html должен быть включен в проект Hello.

11. Двойной щелчок мыши по имени файла Hello.html в области “Project” открывает рабочую область файла, где в закладке “Sourse” вводится текст HTML-файла.

12. Компиляция исходных текстов.

13. Для запуска приложения в области “Project” из контекстного меню файла Hello.html выбираем пункт “Run”.

1. Простейший апплет Hello

Исходный текст Java-файла простейшего апплета выглядит следующим образом:

```
/*---------- Пример 1. Файл Hello.java ----------*/
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);
   }
```

```
}/*-------------------------------------------------------*/
```

Файл HTML-документа со ссылкой на апплет Hello должен содержать следующий код:

```
<!-- ..... Пример 1. HTML-файл ..... -->
<html>
  <head>
        <title>Hello</title>
  </head>
  <body>
        <applet code=Hello.class width=200 height=200></applet>
  </body>
</html>
<!-- ..... -->
```

Теперь рассмотрим, из каких обязательных частей состоит апплет. Класс Hello, определенный в этом апплете, также является первичным классом, хотя он достаточно сильно отличается от первичных классов для простых приложений. Для обычной Java-программы необходимо было определить только один обязательный метод в первичном классе - метод main(). В первичном классе апплета необходимо определить как минимум два метода. Другие методы определяются в случае необходимости создания некоторых специальных эффектов.

Класс Hello определяется как public, а это значит, что он доступен для других объектов. Кроме того, явно установлен суперкласс класс Hello. Им является класс Applet (java.applet.Applet). Класс Applet должны расширять все апплеты, в том числе и класс Hello. Покажем *иерархию классов* (или дерево наследования) для апплетов:

$Hello \rightarrow Aplet \rightarrow Panel \rightarrow Container \rightarrow Component \rightarrow Object$

Класс Hello наследует данные и поведение класса Applet. Являясь подклассом класса Applet, класс Hello может считаться его более специализированным вариантом.

Большинство апплетов объявляет как минимум два метода: init() и paint(). Эти методы уже существуют в классе Applet, расширяемом всеми апплетами, включая и подкласс Hello. Объявление в подклассе метода с тем же самым именем, который есть и в суперклассе, заменяет метод суперкласса методом подкласса. Таким образом, происходит переопределение методов суперкласса с помощью специализированных версий этих методов подкласса.

Переопределяемый метод init() не имеет параметров, ничего не возвращает и объявляется открытым. Единственное, что делает в этом методе апплет Hello - это заставляет изме-

нить размеры окна апплета. Метод resize() является методом класса Applet, поэтому можно использовать его и другие методы этого класса.

Вторым переопределяемым методом является paint(), который представляет собой подпрограмму, используемую для создания на экране изображения. Подобно методу init(), он объявлен открытым и ничего не возвращает. Но у него есть параметр, а именно - объект класса Graphics. Методами этого класса можно пользоваться для вывода графической информации в окно апплета.

Рассмотрим еще раз более систематизировано основные различия между первичным классом апплета и обычной Java-программы:

- Ни один из методов в первичном классе апплета не является статическим. Из этого можно сделать вывод, что этот класс должен быть в какой-то момент явным образом реализован (в отличие от первичного класса приложения). Но в тексте апплета Hello оператора реализации класса нет. Отсюда следует, что оболочка времени выполнения апплета, встроенная в WWW-броузер, сама реализует первичный класс апплета.
- Первичный класс апплета является расширением класса Applet (или подклассом Applet). Класс Applet включает в себя те функции, которые должен иметь каждый апплет.
- Если проверить выполнение апплета Hello, то видно, что оба включенных в первичный класс метода отработали несмотря на то, что код самого апплета не содержал явных вызовов этих методов. Это объясняется тем, что также, как и оболочка времени выполнения Java сама ищет и вызывает метод main() в первичном классе программы, оболочка времени апплета самостоятельно вызывает методы, входящие в подкласс Hello класса Applet.

Для более полного понимания принципа функционирования апплетов обратим особое внимание на последний пункт. В обычных приложениях оболочка времени выполнения вызывает метод main(), который вызывал остальные методы и реализовывал алгоритм программы. В отличие от этого, когда оболочка времени выполнения броузера запускает апплет, она прежде всего ищет и вызывает метод init(). Метод init() выполняет только служебные действия и не отвечает за работу всей программы. Метод же paint() вызывается самой системой всегда, когда содержимое окна требуется обновить. Например, если при работе в Windows окно броузера перекрыть окном другого Windows-приложения, то после того, как окно броузера снова откроется, система сразу же вызовет метод paint(), чтобы восстановить содержимое окна.

За исключением того факта, что оболочка времени выполнения броузера во время работы апплета сама вызывает методы, которые являются переопределениями методов стандартного класса Applet, первичный подкласс апплета ведет себя также, как и первичные классы простых приложения. В первичном классе апплета также можно определять новые

методы (а не только переопределять методы, определенные в стандартном классе Applet), объявлять переменные и реализовывать новые классы.

1.1 Апплет Hello, управляемый мышью

Класс Applet содержит большое количество методов, которые вызываются в ответ на действия пользователя (например, перемещение курсора мыши в пределах окна или нажатие определеннх клавиш на клавиатуре). Приведем в качестве примера использование метода mouseDown(), который вызывается каждый раз, когда пользователь в пределах области, занятой апплетом, нажимает левую клавишу мыши. Усовершенствованная программа Hello перерисовывает строчку текста в той точке, где пользователь щелкнул мышью.

```
/*---------- Пример 2. Файл Hello.java ----------*/
import java.applet.*;
import java.awt.*;
public class hello extends Applet
{  int curX=50, curY=50;
   public void init()
   {          resize(640,480);
   }
   public void paint(Graphics g)
   {          g.drawString("Hello, WWW",curX,curY);
   }
   public boolean mouseDown(Event e,int x,int y)
   {          curX=x; curY=y;
              repaint();
              return true;
   }
}/*-----------------------------------------------------------*/
```

Следует обратить внимание, что в методе mouseDown() вызывается метод repaint(). Этот метод сообщает оболочке времени выполнения, что необходимо обновить изображение в окне. В ответ на это оболочка времени выполнения предает параметры экрана, содержащиеся в объекте типа Graphics, методу paint().

2. Простейший апплет HelloApplet, созданный Java Applet Wizard

Для создания шаблона апплета, на основе которого можно разрабатывать специализированные апплеты, можно воспользоваться системой автоматизированной разработки шаблонов апплета Java Applet Wizard, встроенной в JBuilder.

2.1 Создание шаблона апплета HelloApplet

Для создания шаблона апплета HelloApplet (*пример 3)* среде разработки JBuilder выбрать пункт “New” меню “File”. В появившейся диалоговой панели “Object Gallery” выбрать закладку “Web” и отметить тип “Applet”. В диалоговом окне “Applet Wizard” в поле “Class” следует ввести имя HelloApplet и нажать кнопку “Finish”.

В результате работы системы Java Applet Wizard будет создано два файла с исходными текстами: текст апплета (файл HelloApplet.java) и HTML-документ (файл HelloApplet.html).

2.2 Исходные файлы апплета HelloApplet

Рассмотрим тексты создаваемых файлов (комментарии, создаваемые Java Applet Wizard, переведены в данном примере на русский язык). Тексты подобных шаблонных файлов содержатся в Приложении 1 и их можно в дальнейшем использовать в качестве шаблонов апплета.

Пример 3. Файлы, созданные Java Applet Wizard

Листинг Java-файла:

```
//////////////////////////////////////////////////////////////////////
// HelloApplet.java:    Applet
//
//////////////////////////////////////////////////////////////////////
import java.applet.*;
import java.awt.*;
//==========================================================
// Основной класс для апплета HelloApplet
//
//==========================================================
public class HelloApplet extends Applet
{
   // Конструктор класса HelloApplet
   //----------------------------------------------------------------------
   public HelloApplet()
   {
            // Сделать: Добавьте сюда код конструктора
   }
   // ОБЕСПЕЧЕНИЕ ИНФОРМАЦИИ ОБ АППЛЕТЕ:
   // Метод getAppletInfo() возвращает строку, которая описывает
```

```
// апплет. Здесь можно привести такую информацию, как имя
// автора и дата создания, а также любую другую информацию
//----------------------------------------------------------------------
public String getAppletInfo()
{
        return "Name: HelloApplet\r\n" +
                "Created with JBuilder";
}
// Метод init() вызывается системой при первой загрузке или
// перезагрузке апплета. Можно переопределить этот метод
// для выполнения необходимой инициализации апплета, например
// инициализации структур данных, загрузки изображений и
// шрифтов, создания окон фреймов, установки системы
// управления внешним видом или добавления элементов
// пользовательского интерфейса
//----------------------------------------------------------------------
public void init()
{
        // Если для размещения в окне апплета органов управления
        // используется класс "control creator", созданный системой
        // ResourceWizard, из метода init() можно вызвать метод
        // CreateControls(). Следует удалить вызов метода resize()
        // перед добавлением вызова метода CreateControls(),
        // так как эта функция выполняет изменение размера
        // окна апплета самостоятельно
        //---------------------------------------------------------------
        resize(320, 240);
        // Сделать: Добавьте сюда код инициализации
}
// Здесь можно разместить дополнительный код, необходимый
// для “чистого” завершения работы апплета. Метод destroy()
// вызывается, когда апплет завершает работу и будет
// выгружен из памяти
//----------------------------------------------------------------------
public void destroy()
```

```
    {
        // Сделать: Добавьте сюда код завершения работы апплета
    }
    // Обработчик процедуры рисования окна апплета HelloApplet
    //-------------------------------------------------------------------------
    public void paint(Graphics g)
    {
        g.drawString("Created with JBuilder ",10, 20);
    }
    // Метод start() вызывается при первом появлении на экране
    // страницы HTML с апплетом
    //-------------------------------------------------------------------------
    public void start()
    {
        // Сделать: Добавьте сюда код, который должен
        // работать при запуске апплета
    }
    // Метод stop() вызывается, когда страница HTML с
    // с апплетом исчезает с экрана
    //-------------------------------------------------------------------------
    public void stop()
    {
        // Сделать: Добавьте сюда код, который должен
        // работать при остановке апплета
    }
    // Сделать: Добавьте сюда код, необходимый для работы
    // создаваемого специализированного апплета
}
```

Листинг HTML-файла:

```
<html>
<head>
<title>HelloApplet</title>
</head>
<body>
<hr>
```

```
<applet
   code=HelloApplet.class
   name=HelloApplet
   width=320
   height=240 >
</applet>
<hr>
<a href="HelloApplet.java">The source.</a>
</body>
</html>
```

Исходный текст апплета начинается с двух строк, в которых с помощью оператора import подключаются библиотеки классов.

Далее в исходном тексте апплета определяется класс типа public с именем HelloApplet, которое должно совпадать с именем файла, содержащего исходный текст этого класса.

Создавая файл HelloApplet.java, система Java Applet Wizard определила на в классе HelloApplet конструктор и несколько методов, заменив некоторые методы суперкласса Applet.

Конструктор HelloApplet() класса HelloApplet вызывается при создании объекта. По умолчанию тело конструктора, создаваемого системой Java Applet Wizard, не содержит никакого кода. Однако можно добавить в него строки, выполняющие инициализацию апплета при его создании как апплета.

Базовый класс Applet содержит определение метода getAppletInfo(), возвращающего null. В классе HelloApplet, который является подклассом по отношению к классу Applet, метод getAppletInfo() переопределяется так, что теперь он возвращает текстовую информацию об апплете в виде строки класса String.

Метод init() также определен в класса Applet, от которого наследуются все апплеты (определенный в суперклассе, этот метод ничего не делает). Метод init() вызывается тогда, когда WWW-навигатор загружает в свое окно документ HTML с тегом <APPLET>, ссылающимся на данный апплет. В этот момент апплет может выполнить инициализацию, например создать задачи, если он работает в мультизадачном режиме.

В переопределенном системой Java Applet Wizard методе init() по умолчанию вызывается метод resize(), определенный в суперклассе. При помощи этого оператора изменяются размеры окна апплета, установленные в параметрах тега <APPLET>. При желании редакти-

ровать размеры окна апплета через тег <APPLET> документа HTML можно удалить вызов метода resize() из метода init().

Перед удалением апплета из памяти вызывается метод destroy() (метод, обратный методу init()), определенный в классе Applet как пустая заглушка. Система Java Applet Wizard добавляет в исходный текст класса HelloApplet переопределение метода destroy(), в котором можно выполнить все необходимые операции, выполняющиеся перед удалением апплета. Например, если в методе init() создавались какие-либо задачи, то в методе destroy() их можно завершить.

Метод start() вызывается после метода init() в тот момент, когда пользователь начинает просматривать документ HTML со встроенным в него апплетом. Система Java Applet Wizard создает заглушку, переопределяющую метод start() из суперкласса. Этот метод 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(), переопределенный в классе HelloApplet 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. Так как класс Applet является подклассом класса Component (см. иерархию классов апплетов), а класса HelloApplet - подклассом Applet, то метод paint() можно переопределить в классе HelloApplet.

Метод paint() вызывается, когда необходимо перерисовать окно апплета. Перерисовка окна апплета обычно выполняется асинхронно по отношению в работе апплета (подобно перерисовки клиентской части окон Windows-приложений при поступлении сообщения WM_PAINT). В любое время апплет должен быть готов перерисовать содержимое своего окна.

Методу paint() в качестве параметра передается ссылка на объект класса Graphics. По своему смыслу этот объект напоминает контекст отображения, который используется для вывода информации в Windows-окно. Контекст отображения - это как лист бумаги, на котором можно рисовать изображение или выводит текст. Многочисленные методы класса Graphics позволяют задавать различные параметры вывода, такие, например, как цвет или шрифт.

Для вывода изображения апплеты используют координатную систему, которая соответствует режиму MM_TEXT - одному из режимов, использующихся при программирование

для Windows. Начало этой системы координат расположено в левом верхнем углу окна апплета, ось X направлена слева направо, а ось Y - сверху вниз.

2.3 Упрощенный вариант исходного текста апплета HelloApplet

Исходный текст апплета HelloApplet, предоставляемый системой Java Applet Wizard, конечно же можно упростить, оставив только следующий код:

```
//////////////////////////////////////////////////////////////////////
// HelloApplet.java:Applet
//
//////////////////////////////////////////////////////////////////////
import java.applet.*;
import java.awt.*;
//==========================================================
// Основной класс для апплета HelloApplet
//
//==========================================================
public class HelloApplet extends Applet
{
        // Обработчик процедуры рисования окна апплета HelloApplet
        //----------------------------------------------------------
        public void paint(Graphics g)
        {
                g.drawString("Created with JBuilder ",
                                10, 20);
        }
}
```

В представленном фрагменте удалены методы, не выполняющие никакой полезной работы, а также методы getAppletInfo и init(). Апплет HelloApplet будет при этом работать также как и раньше, потому как удаленные методы все равно определены в суперклассе Applet, Система Java Applet Wizard создает пустые переопределения методов только для того, чтобы при необходимости их можно было заполнить какими-нибудь полезными действиями.

Однако метод paint() переопределить нужно в любом случае, так как именно в нем выполняется рисование - специализированная часть работы апплета HelloApplet.

3. Аргументы апплета

Получение аргументов командной строки простых приложений происходит через параметр метода main() первичного класса приложения. Но как же происходит передача аргументов командной строки апплетам?

3.1 Передача параметров апплету

Аргументы командной строки передаются апплетам во время запуска и происходит при помощи специально созданных атрибутов апплетов. Эти *атрибуты параметров апплетов* (или проще - параметры апплетов) определяются в HTML-теге <APPLET> и предоставляют соответствующую информацию же самому апплету.

Параметры апплетов следуют после открывающего 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 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>
```

В этом примере параметру first присваивается значение "Hello!", а параметру second - значение "How are you?". 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 строковые объекты, то необходимо помнить, что все параметры апплетов, даже числовые, являются строками. Это очень важно, поскольку перед использованием этого параметра как числа необходимо его сначала получить как строку, а затем перевести строку в число. Например, если параметр задан в HTML-файле следующим образом:

```
<param name=loop value="5">
```

то для получения численного значения параметра необходимо использовать следующий код (или подобный ему)

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

3.2 Апплет, принимающий параметры

Создадим шаблон апплета AppletWithParam *(пример 4)*, который имеет возможность обрабатывать параметры.

Для того, чтобы апплет принимал параметры, при создании апплета с помощью системы Java Applet Wizard в четвертой диалоговой панели необходимо определить параметры, передаваемые апплету.

Первоначально список параметров пуст, для того чтобы добавить новый параметр, нужно в столбце “Name” ввести имя параметра. Для апплета AppletWithParam следует добавить параметры с именами String_1 и String_2.

В столбце “Member” при заполнении списка параметров отображаются имена полей класса AppletWithParam, в которые можно будет записать значения параметров (при помощи метода getParameter()). Значения столбца “Def-Value” используются для инициализации соответствующих полей класса. Следует задать в этих полях строки “First string” и “Second string” соответственно.

Для описания параметров служит столбец “Description”. Апплет может извлечь эту информацию методом getParameterInfo() и проанализировать.

Рассмотрим тексты созданных файлов. Тексты подобных шаблонных файлов содержатся в Приложении 2 и их можно в дальнейшем использовать в качестве шаблонов апплета, принимающего значения.

Пример 4. Файлы, созданные Java Applet Wizard

Листинг Java-файла:

```
//**************************************************************
// AppletWithParam.java:        Applet
//
//**************************************************************
import java.applet.*;
import java.awt.*;
public class AppletWithParam extends Applet
{
   // ПОДДЕРЖКА ПАРАМЕТРОВ АППЛЕТА
   //----------------------------------------------------------------------
   // Поля класса для хранения значений параметров
   // Создаются автоматически для параметров апплета
```

```
// Поля инициализируются значениями по умолчанию из таблицы
// <type> <MemberVar>        =            <Default Value>
//---------------------------------------------------------------------------
private String m_String_1 = "First string";
private String m_String_2 = "Second string";
// Имена параметров, нужны для функции getParameter
// Создаются автоматически для параметров апплета
//---------------------------------------------------------------------------
private final String PARAM_String_1 = "String_1";
private final String PARAM_String_2= " String_2";
//---------------------------------------------------------------------------
public AppletWithParam()
{
        // Сделать: Добавьте сюда код конструктора
}
//---------------------------------------------------------------------------
public String getAppletInfo()
{
        return "Name: AppletWithParam\r\n" +
                        "";
}
// ПОДДЕРЖКА ПАРАМЕТРОВ
// Метод getParameterInfo() 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()
{
        // ПОДДЕРЖКА ПАРАМЕТРОВ
        // Чтение всех параметров и запись их значений в
        // соответствующие поля класса
        //----------------------------------------------------------------
        String param;
        // Параметр с именем String_1
        //----------------------------------------------------------------
        param = getParameter(PARAM_String_1);
        if (param != null)    m_String_1 = param;
        // Параметр с именем String_2
        //----------------------------------------------------------------
        param = getParameter(PARAM_String_2);
        if (param != null)    m_String_2 = param;
        //----------------------------------------------------------------
        resize(320, 240);
        // Сделать: Добавьте сюда код инициализации
}
//----------------------------------------------------------------------
public void destroy()
{
        // Сделать: Добавьте сюда код завершения работы апплета
}
//----------------------------------------------------------------------
public void paint(Graphics g)
{
        g.drawString("Created with Microsoft Visual J++ Version 1.1", 10, 20);
}
//----------------------------------------------------------------------
public void start()
{
        // Сделать: Добавьте сюда код, который должен
        // работать при запуске апплета
```

```
    }
    //--------------------------------------------------------------------------
    public void stop()
    {
            // Сделать: Добавьте сюда код, который должен
            // работать при остановке апплета
    }
    // Сделать: Добавьте сюда код, необходимый для работы
    // создаваемого специализированного 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="First string">
            <param name=String_2 value="Second string">
</applet>
<hr>
<a href=" AppletWithParam.java">The source.</a>
</body>
</html>
```

Дополним созданный шаблон апплета. Сначала в HTML-файле изменим значения параметров апплета:

```
    <param name=String_1 value="New first string">
    <param name=String_2 value="New second string">
```

А затем выведем в методе paint() 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);
}
```

3.3 URL-адреса, загрузка и вывод графических изображений

Создадим апплет ParamUrlImage (*пример 5*), в котором через параметр аплета передается имя gif-файла, содержимое которого затем выводится в окне апплета. При создании шаблона апплета (тексты шаблонов содержатся в Приложении 2) следует задать имя параметра FileName, а значение по умолчанию - “sample.gif”. В каталог, где содержится файл исходного текста, следует поместить файл с изображением sample.gif

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), передавая ему обычный WWW-адрес, например

```
URL addUrl=new URL("http://www.vvsu.ru/");
```

Нужно отметить, что фрагмент кода, где происходит реализация класса URL, может сгенерировать исключение MalformedURLException, если вдруг объект не может по какой-либо причине быть создан (например, если передаваемая конструктору строка не является URL-адресом). Компилятор требует обработать возможность возникновения исключительной ситуации при помощи блоков try-catch, например:

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

Так как нам нужно загрузить файл с графическим изображением, то необходимо предварительно создать объект URL, передавая конструктору адрес файла. Нужно заметить, что загружать файлы, открывая сетевые соединения с удаленными главными серверами, апплетам запрещено в целях безопасности. Загружать файлы можно только с того сервера, с которого был загружен сам апплет.

Класс URL содержит конструктор, которому передается абсолютный базовый URL и строка, содержащая путь к объекту относительно этого базового указателя. Создать URL на

файлы своего сервера (или своего локального компьютера) можно при помощи именно этого конструктора. Для получения URL на каталог, содержащий класс работающего апплета, используется метод getCodeBase() 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:

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

Предварительно необходимо в классе апплета ParamUrlImage определить переменную Im:

```
Image Im;
```

и импортировать в файл класса ParamUrlImage пакет java.net:

```
import java.net.*;
```

Класс Image определяет простое двумерное графическое изображение. Для загрузки изображения из файла используется метод getImage(), принимающий URL это файла. Этот метод может импортировать изображения любого графического формата, поддерживаемого WWW-броузером (чаще всего используются форматы GIF и JPEG).

Для вывода изображения в окно необходимо в методе paint() апплета ParamUrlImage вызвать метод drawImage() класса Graphics, передавая ему изображение, координаты расположения изображения и ссылку текущий объект класса ParamUrlImage:

```
public void paint(Graphics g)
{       g.drawImage(Im,0,0,this);
}
```

3.4 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(), которую 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.

Создадим шаблон апплета QuickPicture *(пример 6)* на основе шаблонов из Приложения 1 (или при помощи системы Java Applet Wizard). В каталог, где содержится файл исходного текста, следует поместить файл с изображением sample.gif.

В классе апплета объявить переменные:

```
Image pic; // изображение из файла
boolean picLoaded=false; // было ли полностью загружено изображение
```

В методе init() класса апплета загрузить графическое изображение pic из файла, затем создать пустое изображение offScreenImage -нечто вроде виртуального экрана в памяти, получить графический контекст этого “экрана”, в который затем вывести изображение pic:

```
try
{       pic=getImage(new java.net.URL(getCodeBase(),"sample.gif"));
}
catch(java.net.MalformedURLException e)
{       // код здесь выполняется, если строка URL неправильна
        pic=createImage(0,0); // создание пустого изображения
}
// создание виртуального экрана
Image offScreenImage=createImage(size().width,size().height);
// получение его контекста
Graphics offScreenGraphics= offScreenImage.getGraphics();
// вывод изображения на виртуальный экран
offScreenGraphics.drawImage(pic,0,0,this);
```

В данном случае, так как программа выводит постепенно загружающееся изображение не на реальный экран, а в память, в окне апплета ничего не появляется.

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. Поток, созданный метод drawImage(), вызывает метод imageUpdate() до тех пор, пока он возвращает true.

Переопределим в классе апплета метод imageUpdate():

```
public boolean imageUpdate(Image img,int infoflags,int x,int y,int w,int h)
{       if(infoflags==ALLBITS)
        {       // изображение загружено полностью
                picLoaded=true;
```

```
                repaint();// перерисовать окно апплета
                // больше метод imageUpdate не вызывать
                return false;
        }
        return true; // изображение загружено в память не полностью
    }
```

Параметр infoflags этого метода позволяет отследить, какая часть изображения загружена в память (есть несколько определенных состояний этого флага). Когда этот параметр равен ALLBITS, это означает, что это изображение полностью находится в памяти.

Переменная picLoaded используется в методе paint(). Когда она принимает значение true, то изображение pic (загруженное уже полностью), выводится в окно апплета:

```
    public void paint (Graphics g)
    {       if(picLoaded)
            {       // четвертым параметром передается null,
                    // он не позволяет функции drawImage() вызывать
                    // метод imageUpdate() в процессе вывода
                    g.drawImage(pic,0,0,null);
                    // сообщение в панель состояния
                    showStatus("Done");
            }
            else
            {       // сообщение в панель состояния
                    showStatus("Loading image");
            }
    }
```

Так как изображение теперь находится в памяти, оно тотчас появляется в окне апплета.

4. События и их обработка

Для того, чтобы взаимодействовать с пользователем в реальном масштабе времени и отслеживать изменения в оболочке времени выполнения Java, апплеты используют понятие событие (event). *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. Оболочка времени выполнения следит за происходящими событиями и передает информацию о событии особому методу, называемому *обра-*

ботчиком событий (event handler).

4.1 Обработчики событий от мыши и клавиатуры

Многие обычно используемые обработчики событий определены для класса Applet и по умолчанию не делают ничего - чтобы их использовать, надо переопределить метод суперкласса апплетов в классе создаваемого апплета, например:

```
public boolean mouseMove(Event evt,int x,int y)
{       // здесь можно разместить код, выполняющийся
        // если мышь перемещается
        return true; // событие обработано
}
```

Метод mouseMove() вызывается всякий раз при перемещении мыши. После обработки события метод возвращает true.

Перечислим часто используемые обработчики событий:

- boolean mouseDown(Event evt, int x, int y) - нажата кнопка мыши, параметры x и y указывают расположение мыши.
- boolean mouseUp(Event evt, int x, int y) - кнопка мыши отпущена.
- boolean mouseMove(Event evt, int x, int y) - перемещение мыши.
- boolean mouseDrag(Event evt, int x, int y) - перемещение мыши с нажатой кнопкой.
- boolean mouseEnter(Event evt, int x, int y) - перемещение мыши на окно апплета.
- boolean mouseExit(Event evt, int x, int y) - мышь покинула окно апплета.
- boolean keyDown(Event evt, int key) - нажата клавиша перемещения курсора или функциональная клавиша. Параметр key указывает эту клавишу.
- boolean keyUp(Event evt, int key) - клавиша перемещения курсора или функциональная клавиша отпущена.

Переопределяя эти методы в подклассе класса Applet, можно обрабатывать различные действия пользователя.

Обработчикам событий клавиатуры передается в качестве одного из параметров идентификатор клавиши. Идентификатор клавиши - это целое число, которое соответствует нажатой клавише. Обычные клавиши символов имеют значения, соответствующие их ASCII-кодам. Кроме того, Java поддерживает множество специальных клавиш (UP, DOWN, LEFT, RIGHT, PGUP, PGDOWN, HOME, END, F1 - F12). Эти специальные клавиши фактически определены как статические целые переменные (константы) класса Event. Следующий фрагмент иллюстрирует проверку нажатия клавиш управления курсором:

```
public boolean keyDown(Event evt, ink key)
{       switch(key)
```

```
{       case Event.UP:      .....; break;
        case Event.DOWN: .....; break;
        case Event.LEFT:   .....; break;
        case Event.RIGHT: .....; break;
        default: break;
}
return true;
}
```

4.2 Обработка событий

Рассмотрим, что происходит при возникновении события в окне апплета. При этом создается объект класса Event, в котором сохраняется информация для идентификации события и который передается затем методу handleEvent(). Этот метод проверяет тип события и вызывает соответствующий обработчик, передавая ему относящиеся к событию параметры. Например, при перемещении мыши возникает событие типа MOUSE_MOVE, которое и передается методу handleEvent(). Последний вызывает соответствующий этому событию метод mouseMove(), передавая ему положение мыши.

Класс Event содержит переменные, в которых хранится информация о событии. Перечислим переменные, описывающие событие:

- Object target - компонент (объект), в котором произошло событие
- int id - тип события, одна из констант, определенная в классе Event.
- int key - идентификатор клавиши
- int modifiers - состояние маски наложения клавиш
- long when - время, в которое произошло событие.
- int x - коодината x события
- int y - коодината y события
- int clickCount - равна 1 или 2 в зависимости от того был сделан одинарный или двойной щелчок.

Обрабатывать события мыши и клавиатуры можно не только при помощи специальных обработчиков. Если необходимо обработать большое количество различных событий без замены каждого конкретного метода-обработчика, то можно переопределить метод handleEvent(). Нужно только помнить, что методы-обработчики событий не будут вызываться без явного их вызова в переопределенном методе handleEvent(). Приведем пример, в котором некоторые сообщения от мыши обрабатываются в переопределенном методе handleEvent(), а все остальные передаются на обработку методу суперкласса:

```
public boolean handleEvent(Event evt)
```

```
{       switch(evt.id)
        {       default: // передача сообщения на обработку
                         // методу базового класса
                         return super.handleEvent(evt);
                case Event.MOUSE_MOVE:  ....; break;
                case Event.MOUSE_DRAG:  ....; break;
                case Event.MOUSE_UP:    ....; break;
                case Event.MOUSE_DOWN:  ....; break;
        }
        return true;
}
```

События можно генерировать прямо в программе. Для этого нужно просто создать новый экземпляр класса Event, используя один из конструкторов, и послать его соответствующему объекту (можно даже самому себе):

```
// создание сообщения текущему объекту класса (самому себе)
Event Evt=new Event(this, // пункт назначения
        System.currentTimeMillis(), // время события
        MOUSE_MOVE, // тип события
        new_x,new_y, // координаты x и y события
        0,0); // идентификатор клавиши и модификаторы
```

4.3 Апплет, обрабатывающий события

Создадим апплет MouseEvent (*пример 7)*, отображающий положение курсора мыши с помощью вывода графического изображения в точке. Включение и выключение режим слежения происходит при нажатии на клавишу мыши.

При создании шаблона апплета с помощью системы Java Applet Wizard необходимо в третьей диалоговой панели дать возможность системе сгенерировать заготовки методов-обработчиков нажатия на клавишу и перемещение мыши - в исходный текст апплета будут включены заготовки соответствующих методов-обработчиков. Тексты подобных шаблонных файлов содержатся в Приложении 3 и их можно в дальнейшем использовать в качестве шаблонов апплета, обрабатывающего простые сообщения от мыши.

В каталог, где содержится файл исходного текста, следует поместить файл с изображением arrow.gif

Для хранения состояния класса MouseEvent объявить в классе переменные:

```
private int m_x=0, m_y=0; // текущее положения курсора
private boolean follow=true; // режим отслеживания включен/выключен
```

```
Image imageCur; // переменная для граф. изображения
```

Для загрузки графического изображения из файла того же каталога, где располагается файл класса MouseEvent, можно использовать следующий фрагмент, помещаемый в метод init() класса MouseEvent:

```
try
{       imageCur=getImage(new java.net.URL(getCodeBase(),"arrow.gif"));
}
catch(java.net.MalformedURLException e)
{       // код здесь выполняется, если строка URL неправильна
        imCur=createImage(0,0); // создание пустого изображения
}
```

Для прорисовки изображения апплета вызывается метод paint(), его следует переопределить в классе MouseEvent следующим образом:

```
public void paint(Graphics g)
{       // рамка вокруг окна апплета
        Dimension size=size(); // размер компонента (окна апплета)
        g.drawRect(0,0,size.width-1,size.height-1);
        // прорисовка изображения
        g.drawImage(imageCur,m_x,m_y,this)
}
```

За обработку такого события, как перемещение мыши, отвечает метод mouseMove(). В этом методе необходимо сохранить текущие координаты мыши и перерисовать окно апплета:

```
public boolean mouseMove(Event evt, int x, int y)
{       if(follow) // если режим отслеживания
        {       m_x=x; m_y=y; // сохранение координат
                repaint(); // перерисовка окна
        }
        return true;
}
```

Режим слежения за курсором включается и выключается при нажатии, поэтому следует изменить метод mouseDown() следующим образом:

```
public boolean mouseDown(Event evt, int x, int y)
{       follow=!follow;
        return true;
```

}

4.4 Устранение мерцания при выводе, двойная буферизация

При просмотре апплета MouseEvent заметен факт мерцания окна апплета. Этот эффект является результатом усилий апплета перерисовать экран быстрее, чем это технически возможно. Классическим решением такой проблемы является двойная буферизация графического изображения - не осуществлять вывод изображения сразу на экран, а сначала сформировать его полностью в памяти (на виртуальном экране), а затем сразу полностью вывести его в окно.

Когда оболочка времени выполнения должна повторно перерисовывать графические окна - например, когда окно было затенено или когда апплету требуется повторно перерисовать свое окно, - вызывается метод модификации апплета update() и передается объект Graphics графического экрана. Этот метод по умолчанию просто передает полученный объект Graphics методу paint().

Апплет запрашивает перерисовку окна методом repaint(),который в свою очередь как можно скорее вызывает метод модификации update(). Если апплет запрашивает другую перерисовку, прежде чем метод модификации вызвался из первой, апплет модифицируется только в первый раз. Воспользуемся именно эти свойством методом update() для формирования изображения в памяти и вывода его в окно вместо простого разрешения выводить изображение в апплет прямо в окно.

Модифицируем апплет MouseEvent *(пример 8)*, так чтобы эффект мерцания был устранен. В апплете нужно определить ссылки на виртуальный экран и его контекст отображения:

```
// изображение (виртуальный экран)
private Image OffScreenImage;
// контекст отображения (интерфейс) изображения
private Graphics OffscreenGraphics;
```

Для создания объектов этих типов добавим следующий фрагмент в метод init() после кода загрузки картинки из файла arrow.gif:

```
// пустое изображение (виртуальный экран)
OffScreenImage=createImage(size().width,size().height);
// получение контекста отображения для OffScreenImage
OffscreenGraphics=OffScreenImage.getGraphics();
```

В этом фрагменте создается пустое изображение размером с окно апплета (виртуальный экран). Объект OffscreenGraphics - это контекст для вывода в OffScreenImage. Апплет будет при помощи этого контекста рисовать на виртуальном экране с такой скоростью, с ка-

кой только сможет. Метод модификации update(), переопределяемый в апплете выводит изображение на объект Graphics с такой скоростью, с какой только сможет:

```
public void update(Graphics g)
{       // перерисовать виртуальный экран,
        // то есть изображение OffScreenImage
        paint(OffscreenGraphics);
        // вывести изображение OffScreenImage в окно
        g.drawImage(OffScreenImage,0,0,this);
}
```

Переопределим также и метод paint(), теперь он может, используя передаваемый ему контекст, выводить изображение на виртуальный экран при вызове его из метода update(), когда сам апплет требует перерисовки методом repaint(), так и при вызове его системой тогда, когда для окна требуется перерисовка в связи с его перекрытиями другими окнами:

```
public void paint(Graphics g)
{
        g.setColor(Color.gray);
        g.fillRect(0,0,size().width-1,size().height-1);
        g.setColor(Color.black);
        g.drawRect(0,0,size().width-1,size().height-1);
        g.drawImage(imageCur,m_x,m_y,this);
}
```

5. Апплеты двойного назначения

Апплеты двойного назначения - это программа, которая может работать и под управлением WWW-навигатора, и автономно, как самостоятельное приложение. Создать апплеты двойного назначения, объединяющие функциональные возможности апплета и приложения в одном программном коде, 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 (для создания рамки используется подкласс класса Frame, в котором как минимум переопределен метод-обработчик событий). Затем создать экземпляр класса апплета и связать его рамкой (фреймом). С помощью экземпляра апплета можно вызвать методы init() и start(), запуская апплет в методе main() так, как обычно это делает WWW-навигатор.

Нужно напомнить, что приложения не принимают параметры через HTML-файл, как это делает апплет. И следовательно, придется передавать приложению все параметры в одной командной строке, как это обычно делается с аргументами стандартной командой строки. Т.е. передача параметров такому апплету двойного назначения должна дублироваться при помощи командной строки и при помощи тега <PARAM> HTML-файла.

Апплеты двойного назначения полезны в случае, если есть программа и необходимо, чтобы она работала и как апплет, и как приложение, без написания двух отдельных ее версий.

Создадим апплет двойного назначения Combi (*пример 9)* при помощи системы Java Applet Wizard. Для подготовки шаблона апплета двойного назначения в первой диалоговой панели нужно установить переключатель “As on applet and as on application”.

Рассмотрим тексты файлов, созданные системой Java Applet Wizard. Тексты подобных шаблонных файлов содержатся в Приложении 4 и их можно в дальнейшем использовать в качестве шаблонов апплета двойного назначения.

Пример 9. Файла, созданный Java Applet Wizard

Листинг Java-файла первичного класса:

```
//**************************************************************
// Combi.java:          Applet
//
//**************************************************************
import java.applet.*;
import java.awt.*;
import CombiFrame; // импорт класса рамки (фрейма)
public class Combi extends Applet
{
   // ПОДДЕРЖКА РАБОТЫ ПРОГРАММЫ КАК ПРИЛОЖЕНИЯ:
   // Признак режима работы программы:
   // true - работа в качестве приложения
   // false - работа в качестве апплета
   //---------------------------------------------------------
   private boolean m_fStandAlone = false;
   // ПОДДЕРЖКА РАБОТЫ ПРОГРАММЫ КАК ПРИЛОЖЕНИЯ:
   // Метод main() - точка входа в случае работы программы как
   // приложения. Если программа загружается при помощи
   // документа HTML, то этот метод игнорируется
```

```
//------------------------------------------------------
public static void main(String args[])
{       // Здесь можно обработать аргументы
        // командной строки, если они есть
        // Создать рамку (фрейм) для апплета. Класс CombiFrame
        // наследуется от класса Frame, в нем как минимум
        // переопределяется метод завершение работы приложения
        //-----------------------------------------------
        CombiFrame frame = new CombiFrame("Combi");
        // До изменения размеров фрейма отобразить его.
        // Это необходимо для того, чтобы метод insert()
        // выдавал правильные значения
        //-----------------------------------------------
        frame.show(); frame.hide();
        frame.resize(frame.insets().left + frame.insets().right + 320,
                frame.insets().top + frame.insets().bottom + 240);
        // Создание объекта апплета, связывание апплета и фрейма
        //-----------------------------------------------
        Combi applet_Combi = new Combi();
        frame.add("Center", applet_Combi);
        // Установление признака режима работы - приложение
        //-----------------------------------------------
        applet_Combi.m_fStandAlone = true;
        // Вызов методов ааплета для его запуска
        //-----------------------------------------------
        applet_Combi.init();
        applet_Combi.start();
        // Отображение окна фрейма
        //-----------------------------------------------
        frame.show();
}
//------------------------------------------------------
public Combi()
{
        // Сделать: Добавьте сюда код конструктора
```

```
}
//---------------------------------------------------
public String getAppletInfo()
{
        return "Name: Combi\r\n" +
                "";
}
//---------------------------------------------------
public void init()
{       //-----------------------------------------------
        resize(320, 240);
        // Сделать: Добавьте сюда код инициализации
}
//---------------------------------------------------
public void destroy()
{
        // Сделать: Добавьте сюда код завершения работы апплета
}
//---------------------------------------------------
public void paint(Graphics g)
{
        g.drawString("Created with Microsoft Visual J++ Version 1.1",10, 20);
}
//---------------------------------------------------
public void start()
{
        // Сделать: Добавьте сюда код, который должен
        // работать при запуске апплета
}
//---------------------------------------------------
public void stop()
{
        // Сделать: Добавьте сюда код, который должен
        // работать при остановке апплета
}
```

```
    // Сделать: Добавьте сюда код, необходимый для работы
    // создаваемого специализированного апплета
}
```

Листинг Java-файла класса фрейма для апплета:

```
//*****************************************************************
// CombiFrame.java:
//
//*****************************************************************
import java.awt.*;
// ПОДДЕРЖКА РАБОТЫ ПРОГРАММЫ КАК ПРИЛОЖЕНИЯ:
// Этот класс действует как окно, в котором отображается апплет,
// когда он запускается как обычное приложение
//=================================================
class CombiFrame extends Frame
{
    // Конструктор класса CombiFrame
    //-------------------------------------------------------
    public CombiFrame(String str)
    {
            // Сделать: Добавьте сюда код конструктора
            // Вызов конструктора базового класса должен быть
            // первым оператором конструктора подкласса
            super (str);
    }
    // Метод-обработчик handleEvent() обрабатывает все события,
    // получаемые окном фрейма. В этом методе можно обрабатывать
    // все оконные события. Для обработки событий, генерируемых
    // меню, кнопками и другими элементами управления
    // используется метод action()
    //-------------------------------------------------------
    public boolean handleEvent(Event evt)
    {
            switch (evt.id)
            {
                    // при закрытии окна завершается работа приложения
```

```
                    //---------------------------------------------
                    case Event.WINDOW_DESTROY:
                            // Сделать: Добавьте сюда код, который должен
                            // работать при остановке приложения
                            dispose(); // удаление окна
                            System.exit(0); // завершение приложение
                            return true;
                    default:
                            // передача сообщения на обработку
                            // методу базового класса
                            return super.handleEvent(evt);
            }
    }
}
```

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

Задания к лабораторной работе

Задание 1. Проверить и объяснить работу всех апплетов, рассматриваемых в данной главе и отмеченных курсивом. Должны быть созданы следующие апплеты: *Hello* (пример 1), **модифицированный апплет** *Hello* (пример 2), *HelloApplet* (пример 3), *AppletWithParam* (пример 4), *ParamUrlImage* (пример 5), *QuickPicture* (пример 6), *MouseEvent* (пример 7), **модифицированный апплет** *MouseEvent* (пример 8), *Combi* (пример 9).

Задание 2. Дать ответы на контрольные вопросы.

Контрольные вопросы

1. Чем выполнение апплета отличается от выполнения простого Java-приложения?
2. Чем отличаются первичные классы приложения и апплета?
3. Какие методы должен переопределять первичный класс апплета?
4. Каковы принципы функционирования апплета?
5. Какие методы переопределяются в шаблоне апплета, предоставляемом системой Java Applet Wizard? Каково их назначение?
6. Как передаются параметры апплету?
7. Как загрузить графическое изображение из файла?
8. Как ускорить вывод графических изображений, загружаемых из файла?

9. Какие фрагменты кода для обработки параметров создает система Java Applet Wizard?
10. Что называется событием, когда они генерируются?
11. Какие обработчики событий используются чаще всего?
12. Как переменные содержит класс Event для идентификации события?
13. Как обработать любое событие?
14. Как приложение может само сгенерировать сообщение?
15. Как устранить мерцание при выводе изображений?
16. Что такое апплеты двойного назначения? Как они работают?

ЛАБОРАТОРНАЯ РАБОТА № 3.
РИСОВАНИЕ В ОКНЕ, ОБРАБОТКА СОБЫТИЙ МЫШИ И КЛАВИАТУРЫ
(4 часа).

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ

1. Рисование в окне

Язык Java предоставляет набор графических примитивов, которые позволяют рисовать текст, линии, прямоугольники и др. прямо на экране. При этом можно рисовать эти примитивы сами по себе и с заполнением каким угодно цветом. Для вывода текста можно использовать различные доступные шрифты.

1.1 Графика

В процессе рисования на экране прежде всего требуется наличие объекта класса Graphics (контекст отображения). Графические операции всегда выполняются над объектом Graphics. Например, в апплетах для вывода в окно используется метод paint(), которому передается единственный параметр - объект класса Graphics.

Некоторые методы класса Graphics:

- Graphics - Конструктор нового объекта - контекста отображения
- clearRect - Очищает указанный прямоугольник, заполняя цветом фона
- clipRect - Задает область ограничения вывода
- getClipRect - Возвращает ограничивающий прямоугольник области отсечения
- 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
- GetColor - Получить текущий цвет
- getFont - Получить текущий шрифт
- getFontMetrics - Получить размеры шрифта
- setColor - Устанавливает текущий цвет
- setFont - Устанавливает текущий шрифт
- setPaintMode - Устанавливает режим заполнения текущим цветом
- setXORMode - Устанавливает режим заполнения
- translate - Сдвиг начала системы координат в контексте отображения

1.2 Цвет

Для задания текущего цвета используется метод setColor() класса Graphics:

```
// пусть g - объект Graphics, создадим случайный цвет и установим его
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), определяемой в виде составляющих красного, зеленого и синего цветов.

Для использования цвета необходимо сначала создать объект Color и передать в него значения красного, зеленого и синего цвета (причем существуют два конструктора - для за-

дания 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(0,255,0); // создать зеленый цвет
Color clr3=new Color(0,0,255); // создать красный цвет
Color clr4=new Color(255,255,255); // создать белый цвет
Color clr5=new Color(0,0,0); // создать черный цвет
Color clr6=new Color(100, 100, 100); // создать оттенок серого цвета
```

Язык Java предоставляет ряд заранее установленных цветов, которые можно использовать, не задавая их значения (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:

```
// пусть g - объект Graphics, установим красный цвет
g.setColor(Color.red);
```

Класс Color содержит ряд методов, которые позволяют осуществить запрос для нахождения различных составляющих цвета (методы getRed(), getGreen(), getBlue(), getRGB()), получения разновидностей текущего цвета (например, методы brighter() и darker() возвращают соответственно более светлый или более темный оттенок текущего цвета), а также другие методы.

Несмотря на то, что можно создать миллионы различных цветов, для обеспечения наилучшего из возможных результатов, охватывающих широкий спектр применяемых дисплеев, рекомендуется работать с 256 или меньше цветами. И применять там, где это возможно, стандартные цвета, определяемыми переменными, а не создавать собственные цвета.

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

1.3 Шрифты

Класс Graphics позволяет размещать на экране текст с использованием установленного шрифта. Для того чтобы воспользоваться шрифтом, необходимо прежде всего создать объект класса Font. Для этого необходимо лишь предоставить 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 по всей вероятности имеются на всех системах, на которых будут работать апплеты.

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:

```
// пусть g - объект Graphics, создадим шрифт и установим его
g.setFont(new Font("Times Roman",Font.BOLD,72));
```

Некоторые методы класса Font:

- getFamily - Получает название шрифта, зависящее от платформы. Для получение логического имени испольуется метод getName
- 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

Способы получения метрик шрифта.

Для получения специфических типографических параметров шрифта Font можно воспользоваться методами класса FontMetrics. Для это необходимо просто предоставить объект Font конструктору FontMetrics, в результате чего получится объект FontMetrics, для которого можно применять методы его класса.

Метрики шрифта Font, можно получить и при помощи метода getFontMetrics() класса Graphics.

Замечание. Если попытаться выбрать шрифт, который не установлен на конкретной машине, Java заменит его стандартным шрифтом (например, Courier). Для того, чтобы узнать какие шрифты - из установленных на компьютере в данный момент - доступны апплету, можно воспользоваться методом getFontList(), определенным в классе Toolkit. Кстати, этот класс содержит еще два интересных метода: getScreenResolution() и getScreenSize().

Для того, чтобы воспользоваться методами класса Toolkit, необходимо сначала при помощи статического метода getDefaultToolkit**()** получить ссылку на объект этого класса, а затем, пользуясь полученной ссылкой, извлекать различную информацию.

1.4 Приложение FontsList

Задание. Создать апплет двойного назначения FontsList, в окне которого отображается список всех доступных апплету шрифтов.

Методические указания. Апплет должен быть создан на основе шаблонов, содержащихся в Приложении 4 (или при помощи системы Java Applet Wizard).

Перерисовка окна апплета (метод paint()).

Для перерисовки изображения в окне используется метод paint(), которому передается объект g типа Graphics (g - контекст отображения для окна). Методами этого класса пользуются для вывода графической информации в окно апплета.

Сначала получим объект, который содержит метрики текущего шрифта для контекста g:

```
FontMetrics fm=g.getFontMetrics();
```

Сохраним высоту символов, выдаваемую методом fm.getHeight(), в переменной yStep типа int. Эта величина нужна для определения шага вывода строк по вертикали.

Затем получим список доступных апплету шрифтов:

```
// получение списка доступных шрифтов
Toolkit toolkit=Toolkit.getDefaultToolkit();
String fntList[]=toolkit.getFontList(); // массив строк - список шрифтов
```

Методом resize() апплета изменим размер окна апплета так, чтобы ширина окна была 240, а высота равнялась 20+yStep*fntList.length.

Далее отобразим список всех доступных апплету шрифтов. Для этого в цикле по i (i от 0 до fntList.length) методом g.drawString() выведем строку fntList[i] в точке с координатами (10, 20+yStep*i).

2. Обработка событий

Когда пользователь выполняет операции с мышью или клавиатурой в окне апплета, возникают события, которые передаются соответствующим методам класса Applet. Переопределяя эти методы, можно организовать обработку событий, возникающих от мыши и клавиатуры.

2.1 Как обрабатываются события

Когда возникает событие, управление получает метод **handleEvent()** из класса Component (класс Applet является подклассом класса Component). Прототип метода handleEvent():

```
public boolean handleEvent(Event evt);
```

В качестве параметра методу передается объект класса **Event**, который содержит всю информацию о событии. По содержимому элементов класса Event можно определить коор-

динаты курсора мыши в момент, когда пользователь нажал клавишу, отличить одинарный щелчок мыши от двойного и т.д.

Перечислим элементы класса Event, которые можно проанализировать:

- Object target - компонент (объект), в котором произошло событие.
- Object arg - произвольный аргумент события, значение которого зависит от типа события.
- int id - тип события, одна из констант, определенная в классе Event.
- int key - идентификатор клавиши.
- int modifiers - состояние маски наложения клавиш.
- long when - время, в которое произошло событие.
- int x - коодината x события.
- int y - коодината y события.
- int clickCount - равна 1 или 2 в зависимости от того был сделан одинарный или двойной щелчок.

Элемент id (тип события) может содержать следующие значения:

- ACTION_EVENT - пользователь хочет, чтобы произошло некоторое событие, например, он нажал на кнопку, изменил состояние переключателя, выбрал элемент из меню выбора (выпадающего списка), выбрал элемент раскрывающегося списка, для поля ввода нажал клавишу «Enter», выбрал пункт из меню окна.
- GOT_FOCUS - компонента (например, окно апплета) получил фокус ввода.
- KEY_ACTION - пользователь нажал функциональную клавишу F1 - F12 или клавишу перемещения курсора.
- KEY_ACTION_RELEASE - пользователь отпустил функциональную клавишу F1 - F12 или клавишу перемещения курсора.
- KEY_PRESS - пользователь нажал обычную клавишу.
- KEY_RELEASE - пользователь отпустил обычную клавишу.
- LIST_DESELECT - отмена выделения элемента в списке.
- LIST_SELECT - выделение элемента в списке.
- LOAD_FILE - загрузка файла (не используется в апплетах).
- LOST_FOCUS - компонента потеряла фокус ввода.
- MOUSE_DOWN - пользователь нажал клавишу мыши.
- MOUSE_DRAG - пользователь нажал клавишу мыши и начал выполнять перемещение курсора мыши.
- MOUSE_ENTER - курсор мыши вошел в область окна апплета.

- MOUSE_EXIT - курсор покинул область окна апплета.
- MOUSE_MOVE - перемещение курсора мыши без нажатой клавиши.
- MOUSE_UP - пользователь отпустил клавишу мыши.
- SAVE_FILE - сохранение в файле (не используется в апплетах).
- SCROLL_ABSOLUTE - пользователь переместил бегунок полосы просмотра в новую позицию.
- SCROLL_LINE_DOWN - сдвиг на одну строку вниз.
- SCROLL_LINE_UP - сдвиг на одну строку вверх.
- SCROLL_PAGE_DOWN - сдвиг на одну страницу вниз.
- SCROLL_PAGE_UP - сдвиг на одну страницу вверх.
- WINDOW_DEICONIFY - пользователь сделал запрос операции восстановления нормального размера окна после его минимизации.
- WINDOW_DESTROY - пользователь собирается удалить окно.
- WINDOW_EXPOSE - окно будет отображено.
- WINDOW_ICONIFY - окно будет минимизировано.
- WINDOW_MOVED - окно будет перемещено.

Если событие связано с клавиатурой, в элементе key может находится одно из следующих значений:

- код обычной клавиши.
- UP, DOWN, HOME, END, LEFT, RIGHT, PGUP, PGDN.
- F1 - F12.

Для состояния маски наложения клавиш модификаторов modifiers могут быть указаны следующие:

- ALT_MASK - была нажата клавиши <Alt>.
- META_MASK - была нажата метаклавиша.
- CTRL_MASK - была нажата клавиша <Ctrl>.
- SHIFT_MASK - была нажата клавиша <Shift>.

Приложение может переопределить метод handleEvent() и обрабатывать события самостоятельно, однако есть более простой путь. Обработчик handleEvent(), который используется по умолчанию, вызывает несколько методов, которые более удобны в использовании при обработки простых событий от мыши и клавиатуры. Можно переопределить именно эти специальные методы для обработки этих событий.

2.2 События от мыши

В результате таких операций как нажатие и отпускание клавиши мыши, перемещение курсора мыши в окне апплета с нажатой или отпущенной клавишей, перемещение курсора мыши в окно и выход курсора из окна, возникают события, которые можно обработать специальными методами.

Все перечисляемые ниже методы должны вернуть значение true, если обработка события выполнена успешно и дальнейшая обработка не требуется. Если же методы вернут значение false, то событие будет обработано методом-обработчиком суперкласса, то есть для него будет выполнена обработка, принятая по умолчанию.

Нажатие клавиши мыши

Переопределение метода

public boolean mouseDown(Event evt, int x, int y);

позволяет отслеживать нажатия клавиши мыши. Через параметр evt передается объект Event, с помощью которого можно получить полную информацию о событии. Анализируя параметры x и y, можно определить координаты курсора на момент возникновения события.

Для отслеживания двойного щелчка мыши не предусмотрено никакого отдельного метода. Однако, анализ содержимого элемента clickCount переменной evt, можно определить кратность щелчка мыши:

```
if(evt.clickCount>1)        showStatus("Mouse Double Click");
else                        showStatus("Mouse Down");
```

Отпускание клавиши мыши

При отпускании клавиши мыши управление получает метод

public boolean mouseUp(Event evt, int x, int y);

Анализируя параметры x и y, можно определить координаты точки, в которой пользователь отпустил клавишу.

Перемещение курсора мыши

Когда пользователь перемещает курсор мыши над окном апплета, в процессе перемещения происходит вызов метода

public boolean mouseMove(Event evt, int x, int y);

Через переменные x и y передаются текущие координаты курсора мыши.

Перемещение курсора мыши с нажатой клавишей

Операция «Drag and Drop» выполняется следующим образом: пользователь нажимает клавишу мыши и начинает перемещать курсор мыши, не отпуская клавиши. При этом происходит вызов метода

public boolean mouseDrag(Event evt, int x, int y);

Через переменные x и y передаются текущие координаты курсора мыши. Метод mouseDrag() вызывается даже в том случае, если в процессе перемещения курсор вышел за пределы окна апплета.

Вход курсора мыши в область окна

Когда курсор мыши в процессе перемещения по экрану попадает в область окна, управление передается методу

public boolean mouseEnter(Event evt, int x, int y);

Этот метод можно использовать для активизации апплета, на который указывает курсор мыши.

Выход курсора мыши в области окна

Если курсор покидает область окна апплета, вызывается метод

public boolean mouseExit(Event evt, int x, int y);

Если пользователь убрал из окна апплета, активизированного методом mouseEnter(), то метод mouseExit() может переключить апплет в пассивное состояние.

2.3 Приложение LinesDraw

Задание. Создать 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. При перерисовке апплет будет пользоваться координатами всех нарисованных ранее линий. Для того, чтобы стереть содержимое окна апплета, необходимо сделать двойной щелчок в окне. При этом из массива координат должны будут удалены все элементы.

Методические указания. Апплет должен быть создан на основе шаблонов, содержащихся в Приложении 5 (или при помощи системы Java Applet Wizard)**.**

Объявление элементов класса апплета.

В классе апплета объявим следующие элементы:

```
int xDown, yDown;          // координаты нажатия клавиши
int xPrev, yPrev;    // предыдущие координаты конца линии
boolean bDrawing;          // признак включения/выключения режима рисования
Vector lines;              // массив координат линий
```

В массив lines типа Vector будем добавлять элементы типа Rectangle, которые будут содержать координаты линии (класс Rectangle содержит элементы x, y, width, height, сохраненная линия будет идти от (x, y) до (x+width, y+height)). Для того, чтобы использовать класс Vector, в файл класса апплета импортируем пакет java.util:

import java.util.;*

Инициализация апплета (метод init()).

При инициализации апплета в методе init() установим признак bDrawing равным false (установлен *признак выключения режима рисования*) и создадим массив lines при помощи конструктора без параметров Vector().

Перерисовка окна апплета (метод paint()).

Для перерисовки изображения в окне используется метод paint(), которому передается объект g типа Graphics (g - контекст отображения для окна). Методами этого класса пользуются для вывода графической информации в окно апплета.

Сначала раскрасим фон окна и нарисуем рамку окна. Для этого сделаем следующие действия: при помощи метода size() класса апплета получим текущий размер окна - объект appSize типа Dimension; методом g.setColor() выберем цвет фона Color.yellow; зарисуем всю внутреннюю область окна с помощью метода g.fillRect() и объекта appSize; методом g.setColor() выберем цвет рамки Color.black; нарисуем рамку вокруг окна методом g.drawRect(), используя объект appSize.

Дальше следует вывод всех уже нарисованных линий. Для этого: получим размер массива lines с помощью метода lines.size(); затем в цикле получим координаты каждой (i-той) линии и нарисуем ее

Rectangle p=(Rectangle)lines.elementAt(i);

g.drawLine(p.x, p.y, p.x+p.width, p.y+p.height);

После вывода всего изображения установим *признак выключения режима рисования.*

Обработка нажатия клавиши мыши (метод mouseDown()).

Если сделан двойной щелчок мышью, то очистим массив lines методом lines.removeAllElements() и перерисуем содержимое окна вызовом метода repaint().

Иначе, если произошло просто нажатие на клавишу, то необходимо сохранить начальную координату линии. Для этого запомним в элементах xDown и yDown координаты начала линии, которые содержатся в параметрах x и y, передаваемых методу mouseDown(). Предыдущие координаты конца линии xPrev и yPrev для процесса рисования сначала установим равными началу линии. Затем установим *признак включения режима рисования.*

Обработка отпускании клавиши мыши (метод mouseUp()).

Если установлен *признак включения режима рисования*, то это значит, что данная координата указателя является конечной точкой линии, и что координаты линии необходимо запомнить в массиве lines. Поэтому создадим новый объект Rectangle и добавим его в массив lines:

Rectangle p=new Rectangle(xDown,yDown,x-xDown,y-yDown);

lines.addElement(p);

Затем перерисуем содержимое окна методом repaint() и установим *признак выключения режима рисования.*

Обработка перемещения указателя мыши при нажатой клавише (метод mouseDrag()).

Для того, чтобы что-то нарисовать в окне апплета вне метода paint(), необходимо получить контекст отображения (методу paint() этот контекст передается через параметр). Получим этот контекст методом апплета getGraphics():

Graphics g=getGraphics();

Затем удалим изображение линии, нарисованное ранее. Для этого методом g.setColor() выберем цвет фона Color.yellow и нарисуем линию от (xDown, yDown) до (xPrev, yPrev) методом g.drawLine().

Нарисуем новую линию, установив методом g.setColor() цвет Color.black и проведя методом g.drawLine() линию от (xDown, yDown) до (x, y). Параметры x и y передаются методу mouseDrag().

Предыдущие координаты конца линии xPrev и yPrev для процесса рисования установим теперь равными x и y соответственно. После этого установим *признак включения режима рисования.*

Обработка перемещения указателя мыши без нажатой клавиши (метод mouseMove()).

В этом методе просто установим *признак выключения режима рисования.*

2.4 События от клавиатуры

Апплет может обрабатывать события, создаваемые клавиатурой. Он может реагировать на нажатие и отпускание функциональных клавиш F1 - F12, клавиш перемещения курсора, обычных клавиш.

Для того, чтобы обработать события от клавиатуры, апплет должен переопределить методы keyDown() и keyUp():

public boolean keyDown(Event evt,int nkey);

public boolean keyUp(Event evt,int nkey);

Первый параметр этих функция - объект класса Event, который несет всю информацию о событии. Наибольший интерес представляют элементы key и modifiers переменной evt. Через них передается соответственно код нажатой клавиши и код модификации.

Второй параметр key методов keyDown() и keyUp() дублирует элемент key объекта evt.

Замечание. Прежде чем апплет будет способен принимать события от клавиатуры, ему нужно передать фокус ввода. Это можно сделать, щелкнув в окне левой клавишей мыши. Фокус ввода - это атрибут, который присваивается окну, обрабатывающему события от кла-

виатуры. Так как клавиатура одна, а апплетов и других активных мнокон на экране может быть много, необходим механизм, позволяющий определить, в какое окно направляется событие, создаваемое клавиатурой. Такой механизм и обеспечивается атрибутом фокуса ввода.

2.5 Приложение KeyCodes

Задание. Создать 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).

Методические указания. Апплет должен быть создан на основе шаблонов, содержащихся в Приложении 4 (или при помощи системы Java Applet Wizard)**.**

Объявление элементов класса апплета.

В классе апплета объявим следующие элементы:

int yHeight; // высота символов шрифта, выбранного по умолчанию

Dimension appSize; // текущий размер окна апплета

Инициализация апплета (метод init()).

При инициализации апплета в методе init() получим высоту символов шрифта для вывода текста в окно.

Для того, чтобы получить информацию о контексте окна апплета вне метода paint(), необходимо получить контекст отображения (методу paint() этот контекст передается через параметр). Получим этот контекст g класса Graphics методом апплета getGraphics() (см. приложение LinesDraw).

Затем получим объект fm класса FontMetrics, который содержит метрики текущего шрифта, методом g.getFontMetrics() (приложение FontsList).

Сохраним высоту символов, выдаваемую методом fm.getHeight(), в переменной yHeight. Эта величина нужна для определения шага свертки.

Перерисовка окна апплета (метод paint()).

Для перерисовки изображения в окне используется метод paint(), которому передается объект g типа Graphics (g - контекст отображения для окна). Методами этого класса пользуются для вывода графической информации в окно апплета.

В данном апплете этот метод используется для определения текущего размера appSize окна и для закраски фона окна и рисовании рамки окна (см. приложение LinesDraw).

Обработка нажатия клавиши (метод keyDown()).

Переопределим в классе апплета метод-обработчик нажатия клавиши клавиатуры keyDown(). В этом методе будем выводить строку информации о нажатой клавише в верхней части окна. Сначала сформируем объект s класса String, равный

"Character "+(char)key+" -> Key "+key+" -> Modifiers "+evt.modifiers

затем получим методом getGraphics() контекст окна g класса Graphics и выведем в окно строку s в точке с координатами (10, yHeight) методом g.drawString().

Метод keyDown() должен возвратить true, так как обработка завершена.

Обработка отпускания клавиши (метод keyUp()).

Переопределим в классе апплета метод-обработчик отпускания клавиши клавиатуры keyUp().

Получим методом getGraphics() контекст окна g класса Graphics и с помощью метода g.copyArea() скопируем часть экрана, ограниченную прямоугольником с началом в точке (0, 1), шириной appSize.width-1 и высотой appSize.height-yHeight-5, в область с началом в точке (0, yHeight+1).

Затем методом g.setColor() выберем цвет фона Color.yellow и методом g.fillRect() закрасим область вывода очередной строки - прямоугольник с началом в точке (1, 1), шириной appSize.width-2 и высотой yHeight+1.

Метод keyUp() должен возвратить true, так как обработка завершена.

Задания к лабораторной работе

Задание 1. Создать апплеты *FontsList*, *LinesDraw*, *KeyCodes* и объяснить их работу. Апплеты должны иметь возможность работать как независимые приложения.

Задание 2. Дать ответы на контрольные вопросы.

Контрольные вопросы

17. Над объектом какого типа выполняются все графические операции?
18. Почему рекомендуется использовать ограниченный набор цветов?
19. Почему рекомендуется использовать ограниченный набор шрифтов?
20. Как получить метрики шрифтов?
21. Когда возникают события?
22. Какой метод какого класса получает управление при возникновении события? Что передается ему в качестве параметра?
23. Какие методы отвечают за обработку простых событий от мыши?
24. Какое значение (true или false) должен вернуть метод-обработчик для того, чтобы событие считалось обработанным и не передавалось на дальнейшую обработку методу суперк-

ласса?

25. Какие методы отвечают за обработку простых событий от клавиатуры?

26. Для чего нужен фокус ввода?

ЛАБОРАТОРНАЯ РАБОТА № 4

КОМПОНЕНТЫ И ИСПОЛЬЗОВАНИЕ ЭЛЕМЕНТОВ УПРАВЛЕНИЯ

(4 часа)

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ

Для создания *графического интерфейса пользователя* (GUI) применяются различные элементы управления (кнопки, списки, поля редактирования и т.д.) и контейнеры, содержащие в себе эти элементы и вложенные контейнеры. Предком всех этих компонент пользовательского интерфейса является класс Component.

1. Компоненты GUI

Графическое окружение языка Java основано 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-предок 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
- reshape - Изменяет форму компонента (положение, высоту и ширину)
- prefferedSize - Возвращает предпочтительный размер компонента
- minimumSize - Возвращает минимальный размер компонента
- getGraphics - Получает графический контекст компонента
- getFontMetrics - Получает типографские параметры текущего шрифта
- 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()
- print - Отвечает за печать компонента
- imageUpdate - Отвечает за перерисовку компонента при изменении выводимого в компоненте изображения типа Image
- createImage - Создает изображение
- prepareImage - Подготавливает изображение типа Image для визуализации
- checkImage - Возвращает состояние создания экранного представления изображения
- inside - Проверяет, находится ли заданная точка «внутри» компонента
- locate - Возвращает компонент, содержащий заданную точку
- deliverEvent - Доставляет событие компоненту
- postEvent - Пересылает событие компоненту, в результате чего вызывается метод handleEvent()
- handleEvent - Отвечает за обработку события. Если метод возвращает false, то событие передается предку компонента
- 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 класс контейнеров. Эти компоненты GUI имеют следующие деревья наследования:

Object → Component → Button

Object → Component → CheckBox

Object → Component → Choice

Object → Component → List

Object → Component → Scrollbar

Object → Component → Label

Object → Component → Canvas

Object → Component → TextComponent

Object → Component → TextComponent → TextField

Object → Component → TextComponent → TextArea

Object → Component → Container

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 попадают под класс 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.

2. Устройства или элементы управления

Рассмотрим такие компоненты, как элементы управления. При их помощи можно создавать программы, обладающие развитым пользовательским интерфейсом. Язык Java содержит все стандартные элементы управления, которые можно обнаружить в современных компьютерных операционных системах. Элементы управления сначала создаются, затем добавляются в контейнер (объект класса Container) его методом add().

2.1 Кнопки

Кнопки (класс Button) - это устройства, которые дают возможность нажимать на них, чтобы проинициализировать некоторое действие.

Для создания кнопок необходимо в методе класса Container (или его подкласса) просто создать экземпляр объекта Button, дать ему имя и добавить его в контейнер:

```
Button myButtom=new Button("Click me!");
add(myButton);
```

или еще проще:

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

Следует обратить внимание на то, что при создании кнопки не передаются координаты X, Y ее положения. Это происходит потому, что элементы управления, как и все устройства, размещаются на экране в соответствии с правилами, определяемыми *менеджерами компоновки* (*размещения*): при этом координаты не используются. Подробно менеджеры размещения будут рассматриваться в следующей главе.

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

Обработка событий от кнопки (и от других компонент)

Для обработки событий, создаваемых кнопками, и другими компонентами, можно переопределить метод handleEvent() контейнера, содержащего эти компоненты. Однако существует более простой способ, основанный на переопределении метода action(), который получает управление, когда пользователь совершает какое-либо действие с компонентом (нажатие на кнопку мышью; завершение ввода строки в текстовом поле - нажатие клавиши «Enter»; выбор элемента из списка - двойной щелчок мышью; изменение состояния переключателя щелком мыши или клавишей пробела; выбор элемента из меню выбора - одинарный щелчок мышью).

В качестве первого параметра методу action() передается ссылка на объект класса Event, содержащий всю информацию о событии, второй параметр представляет собой ссылку на объект, вызвавший появление события.

События в методе action() обрабатываются следующим образом. Прежде всего необходимо проверить, объект какого типа создал событие. Это делается при помощи операции instanceof. Рассмотрим пример для обработки событий от кнопок (щелчок мышью на кнопке), предполагая, что контейнер включает в себя кнопки btn1, btn2 и btn3:

```
public boolean action(Event evt, Object obj)
{
        // обработка событий от кнопок
if(evt.target instanceof Button) // объект является кнопкой?
{// получаем ссылку на кнопку, вызвавшую событие
```

```
        Button btn=(Button)evt.target;
        // получаем метку кнопки
        String lbl=btn.getLabel();
   // проверка, от какой именно кнопки пришло событие
        if(btn.equals(btn1)) // от кнопки с номером 1?
   {    // обработка события от кнопки 1
                              .....                          }
        else if(btn.equals(btn2)) // от кнопки с номером 2?
   {    // обработка события от кнопки 2
                                   .....                     }
        else if(btn.equals(btn3)) // от кнопки с номером 3?
   {    // обработка события от кнопки 3
                                        .....                }
        else
   {    // если событие вызвано кнопкой,
   // обработчик которой не
   // предусмотрен в данном методе
        return false;                                    }
   // если событие обработано, иначе return false
              return true;                               }
   // аналогичный процесс обработки событий ACTION_EVENT
   // от других компонент
       if(evt.target instanceof КлассЭлементаGUI)
{  // получаем ссылку на компонент, вызвавший событие
              КлассЭлементаGUI element=
                        (КлассЭлементаGUI)evt.target;
// проверка, от какого именно компонента этого класса
// пришло событие и обработка этого события от
// конкретного компонента
              .....
return true; // если событие обработано, иначе return false
        }
// необработанные события передаем на дальнейшую обработку
        return false;        }
```

Нужно отметить, что не все действия пользователя обрабатываются методом action(), например, события линейки прокрутки можно обработать только методом handleEvent(). Методом action() обрабатываются только те события, для которых элемент id объекта evt класса Event равен ACTION_EVENT (см. предыдущую главу). Другие события обрабатываются более универсальным методом handleEvent().

Рекомендация. Если контейнер содержит элементы управления, события от которых не обрабатываются методом action(), то рекомендуется обработку всех событий от всех компонентов производить в методе handleEvent(), чтобы не делить процесс обработки на части, а обрабатывать события централизовано. Процесс обработки событий в методе handleEvent() аналогичен процессу обработки в методе action():

```
public boolean handleEvent(Event evt)
{       switch(evt.id)
{       default:// передача сообщения на обработку
// методу базового класса
        return super.handleEvent(evt);
        case Event.ACTION_EVENT:
// обработка событий от кнопок
// объект является кнопкой?
        if(evt.target instanceof Button)
{// ссылка на кнопку, вызвавшую событие
        Button btn=(Button)evt.target;
// получаем метку кнопки
        String lbl=btn.getLabel();
        if(btn.equals(btn1)) // событие от кнопки 1?
{// обработка события
                                                .....
}
        else    if(btn.equals(btn2))
{       // обработка события
                                                .....
}
        else    if(btn.equals(btn3))
{       // обработка события
                                                .....
}
```

```
            else return false;
    // если событие обработано, иначе return false
            return true;                                    }
    // аналогичный процесс обработки событий
    // Event.ACTION_EVENT от других компонент
            if(evt.target instanceof КлассЭлементаGUI)
    {       // ссылка на компонент, вызвавший событие
            КлассЭлементаGUI element=
            (КлассЭлементаGUI)evt.target;
    // проверка, от какого именно компонента этого
    // класса пришло событие и обработка этого
    // события от конкретного компонента
    ..................................................
    // если событие обработано, иначе return false
                                            return true;        }
                    break;      }
            return true;        }
```

2.2 Флажки (или переключатели)

Язык Java поддерживает два типа флажков-переключателей (класс Checkbox): исключающие и неисключающие.

Неисключающие флажки (*переключатели с независимой фиксацией)* могут быть выбраны независимо от состояния других флажков. Такие флажки являются самостоятельными объектами класса Checkbox, для их создания используются конструкторы Checkbox() и Checkbox(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 CheckboxGroup, а затем создаются объекты класса Checkbox, входящие в эту группу, для чего используется конструктор Checkbox(String label, CheckboxGroup group, boolean state).

Приведем фрагмент программного кода, в котором создаются один независимый флажок и группа из трех исключающих флажков:

```
// создание одного независимого флажка и добавление его в контейнер
   add(new Checkbox("It is checkbox"));
// создание группы флажков
```

```
    CheckboxGroup gr=new CheckboxGroup();
// создание флажков группы gr и добавление их в контейнер
    Checkbox first, second, third;
// флажок сброшен
    first=new Checkbox("1st checkbox from group",gr,false);
    add(first);
// флажок установлен
    second=new Checkbox("2nd checkbox from group",gr,true);
    add(second);
// флажок сброшен
    third=new Checkbox("3rd checkbox from group",gr,false);
    add(third);
```

Класс CheckboxGroup содержит методы для определения текущего выбора и установки нового выбора для флажков группы. Метод getCurrent() возвращает ссылку на выбранный флажок группы, а метод setCurrent() переносит выбор на задаваемый флажок группы.

При помощи методов класса getLabel() и getState() класса *Checkbox* можно получить метку флажка и текущее состояние переключателя, а при помощи методов setLabel() и setState() изменить метку флажка установить состояние флажка.

Для проверки того, какой группе флажков принадлежит флажок, класса Checkbox содержит метод getCheckboxGroup(), а для того, чтобы поместить ранее созданный независимый флажок в группу, или для изменения группы, в которую теперь будет входить флажок, используется метод setCheckboxGroup().

Обработка событий от флажка

Для обработки событий от флажков (щелчок мышью на флажке свидетельствует об изменении состояния переключателя) следует внести в метод action() контейнера, содержащего эти флажки, следующий фрагмент:

```
// процесс обработки событий от флажка
    if(evt.target instanceof Checkbox)
{  // получаем ссылку на флажок, вызвавший событие
            Checkbox check=(Checkbox)evt.target;
// проверка, от какого именно флажка пришло событие и
// обработка этого события от конкретного флажка
            .....
return true; // если событие обработано, иначе return false}
```

2.3 Меню выбора (или выпадающие списки)

Меню выбора (выпадающие списки) фактически являются всплывающими меню, они дают возможность создавать список элементов выбора, который всплывает на экране в виде меню.

Применяя меню выбора (класс Choice), можно выбирать только один элемент - множественный выбор не разрешается. При необходимости множественного выбора придется воспользоваться раскрывающимся списком (см. следующий пункт).

Приведем пример создания выпадающего списка:

```
// создать меню выбора
Choice choice=new Choice();
// добавить в него элементы
choice.addItem("1st item");
choice.addItem("2nd item");
// добавить меню choice в контейнер
add(choice);
// добавить в меню элемент
choice.addItem("3rd item");
```

Здесь сначала создается объект Choice, а затем в него задаются различные элементы выбора. Далее меню вводится в контейнер. Нужно отметить, что совсем не обязательно задавать все элементы выбора в меню до его ввода в контейнер. В контейнер можно вводить и пустое меню, добавляя элементы выбора по мере необходимости.

Рассмотрим некоторые методы класса Choice. Количество элементов выбора в меню можно определить при помощи метода countItem(), добавляются элементы в меню методом addItem(), а получить название какого-либо элемента (его строковое представление) по его порядковому номеру можно с помощью функции getItem().

Для того, чтобы определить индекс и строковое представление выбранного в меню элемента, предназначены методы getSelectIndex() и getSelectItem(). Изменить выбор в меню можно при помощи метода select().

Обработка событий от меню выбора

Для обработки событий от меню выбора (одинарный щелчок мышью на элементе меню свидетельствует о выборе элемента) следует внести в метод action() контейнера, содержащего эти меню выбора, следующий фрагмент:

```
// процесс обработки событий от меню выбора
   if(evt.target instanceof Choice)
{/// получаем ссылку на меню выбора, вызвавшее событие
```

```
    Choice choice=(Choice)evt.target;
// проверка, от какого именно меню выбора пришло событие
// и обработка этого события от конкретного меню выбора
                .....
return true; // если событие обработано, иначе return false}
```

2.4 Раскрывающиеся списки

Раскрывающиеся списки особенно полезны, если необходимо объединить функциональные возможности меню с возможностью множественного выбора элементов.

Для создания списка необходимо создать объект класса List и ввести в него любые элементы. Затем необходимо этот объект добавить в контейнер:

```
// создать список с тремя видимыми элементами и с возможностью
// множественного выбора (передача true во втором параметре)
    List list=new List(3, true);
// добавить в него элементы
    list.addItem("1st item");
    list.addItem("2nd item");
    list.addItem("3rd item");
// добавить список list в контейнер
    add(list);
```

Рассмотрим некоторые методы класса List. Количество элементов выбора в списке можно определить при помощи метода countItem(). Добавляются элементы в список методом addItem(), удаляются из него методом delItem(). Для очистки всего списка используется метод clear(). Для того, чтобы заменить элемент с заданным номером, используется метод replaceItem(). Получить название какого-либо элемента (его строковое представление) по его порядковому номеру можно с помощью функции getItem().

Для того, чтобы определить индекс(ы) и строковое(ые) представление(ия) выбранного(ных) в списке элемента(ов), предназначены методы getSelectIndex() (getSelectIndexes()) и getSelectItem() (getSelectItems()). Элемент можно выделить как выбранный в списке при помощи метода select(). Снять выделение с помеченного элемента можно методом deselect(). Проверить, является ли элемент выделенным, можно при помощи метода isSelected().

Методом getRows() можно определить количество одновременно видимых элементов списка (его высоту). Методом makeVisible() можно осуществить прокрутку в списке так, чтобы элемент с заданным индексом стал видим.

Метод setMultipleSelections() разрешает или запрещает выбор нескольких элементов одновременно, а метод allowsMultipleSelections() позволяет определить, разрешен ли для списка множественный выбор.

Обработка событий от списка

Для выбора строки (или нескольких строк) из списка класса List, пользователь должен сделать двойной щелчок клавишей мыши по выделенному элементу (или элементам). Этим он отличается от меню выбора: в меню выбора элемент выбирается одинарным щелчком мыши. Для обработки таких событий от списков следует внести в метод action() контейнера, содержащего эти списки, следующий фрагмент:

```
// процесс обработки событий выбора элементов списка
   if(evt.target instanceof List)
{// получаем ссылку на список, вызвавший событие
   List lst=(List)evt.target;
// проверка, от какого именно списка пришло событие и
// обработка этого события от конкретного списка
                .....
return true; // если событие обработано, иначе return false}
```

Однако список класса List создает события не только при двойном щелчке, но и при выделении или отмене выделения нескольких элементов, сделанных одинарным щелчком клавиши мыши. Такие события можно перехватить и обработать, переопределив метод handleEvent(). Процесс обработки событий в методе handleEvent() аналогичен процессу обработки в методе action(), например:

```
public boolean handleEvent(Event evt)
{  switch(evt.id)
{  default:// передача сообщения на обработку
   // методу базового класса
   return super.handleEvent(evt);
         case Event.ACTION_EVENT:
// процесс обработки событий выбора элементов списка
   if(evt.target instanceof List)
{// получаем ссылку на список, вызвавший событие
   List lst=(List)evt.target;
// проверка, от какого именно списка пришло
// событие и обработка этого события от
// конкретного списка
```

```
    .....
// если событие обработано, иначе return false
    return true;                              }
    break;
    case Event.LIST_SELECT:
// процесс обработки событий выделения элементов списка
    if(evt.target instanceof List)
{// получаем ссылку на список, вызвавший событие
    List lst=(List)evt.target;
// проверка, от какого именно списка пришло
// событие и обработка этого события от
// конкретного списка
        .....
// если событие обработано, иначе return false
    return true;   }
    break;
    case Event.LIST_DESELECT:
// процесс обработки событий отмены выделения
// элементов списка
    if(evt.target instanceof List)
{// получаем ссылку на список, вызвавший событие
    List lst=(List)evt.target;
// проверка, от какого именно списка пришло
// событие и обработка этого события от
// конкретного списка
    .....
// если событие обработано, иначе return false
    return true;                                    }
    break;              }
    return true;        }
```

2.5 Полосы прокрутки

Полосы прокрутки к спискам, меню выбора и полям редактирования при необходимости добавляются автоматически. Но можно использовать их и как независимые, отдельные компоненты для обеспечения прокрутки необходимой информации. Создание линейки про-

крутки (класс Scrollbar**)** и добавление ее в контейнер осуществляется, например, следующим образом:

```
    Scrollbar
scrlbar=new Scrollbar.HORIZONTAL,25,15,1,100);
    add(scrlbar);
```

Здесь создается горизонтальная линейка прокрутки, ширина которой на экране равна 15. Начальное значение (текущее положение бегунка линейки прокрутки) установлено равным 25, минимальное возможное значение равно 1, а максимальное - 100.

При помощи методов изменять установки линейки прокрутки. Метод setValue() изменяет текущее положение бегунка, а метод setValues() изменяет параметры полосы прокрутки, задаваемые при ее создании. Для установки строчного и страничного инкремента полосы прокрутки используются методы setLineIncrement() и setPageIncrement().

Для получении информации о линейке используются следующие методы. Метод getValue() возвращает текущее положение бегунка, методы getMaximum() и getMinimum() - возвращают максимальное и минимальное значение для полосы прокрутки. Для получения строчного и страничного инкремента полосы прокрутки используются методы getLineIncrement() и getPageIncrement().

Ширину полосы прокрутки на экране (величину ее видимой части) можно узнать методом getVisible(), а ориентация полосы прокрутки определяется методом getOrientation().

Обработка событий от линейки прокрутки

Для обработки событий от линеек прокрутки (прокрутка вниз и вверх на одну строку, прокрутка вниз и вверх на одну страницу) требуется внести в метод handleEvent() контейнера, содержащего эти линейки, следующий фрагмент:

```
public boolean handleEvent(Event evt)
   {          switch(evt.id)
   {          default:// передача сообщения на обработку
// методу базового класса
              return super.handleEvent(evt);
              case Event.SCROLL_LINE_UP:
// процесс обработки событий линейки прокрутки
              if(evt.target instanceof Scrollbar)
{// получаем ссылку на линейку, вызвавшую событие
              Scrollbar scrlbar=(Scrollbar)evt.target;
// проверка, от какой именно линейки пришло
// событие и обработка этого события от
```

```
// конкретной линейки
                .....................................
// если событие обработано, иначе return false
                return true;                                    }
                break;
                case Event.SCROLL_LINE_DOWN:
// процесс обработки событий линейки прокрутки
                if(evt.target instanceof Scrollbar)
{// получаем ссылку на линейку, вызвавшую событие
                Scrollbar scrlbar=(Scrollbar)evt.target;
// проверка, от какой именно линейки пришло
// событие и обработка этого события от
// конкретной линейки
                .....................................
// если событие обработано, иначе return false
                return true;                                    }
                break;
                case Event.SCROLL_PAGE_UP:
// процесс обработки событий линейки прокрутки
                if(evt.target instanceof Scrollbar)
{// получаем ссылку на линейку, вызвавшую событие
                Scrollbar scrlbar=(Scrollbar)evt.target;
// проверка, от какой именно линейки пришло
// событие и обработка этого события от
// конкретной линейки
                .....................................
// если событие обработано, иначе return false
                return true;    }
                break;
                case Event.SCROLL_PAGE_DOWN:
// процесс обработки событий линейки прокрутки
                if(evt.target instanceof Scrollbar)
{// получаем ссылку на линейку, вызвавшую событие
                Scrollbar scrlbar=(Scrollbar)evt.target;
// проверка, от какой именно линейки пришло
```

```
// событие и обработка этого события от
// конкретной линейки
                .....
// если событие обработано, иначе return false
                return true;                                    }
                break;          }
                return true;    }
```

2.6 Метки

Метки-надписи (класс Label) - это простейшие устройства. Они не порождают событий, они используются просто для размещения на экране строк текста. Преимущество использования этих элементов управления для вывода текста перед выводом текста методом drawString() класса Graphics заключается в том, что эти элементы управления (вместе с их содержимым) имеют способность перемещаться по экрану всякий раз, когда изменяется размер контейнера, содержащего эти метки.

Приведем примеры создания меток (или надписей):

```
// для метки задается выравнивание по левому краю
add(Label lbl1=new Label("1st string", Label.LEFT);
// для метки задается выравнивание по правому краю
add(Label lbl1=new Label("1st string", Label.RIGHT);
// для метки задается выравнивание по центру
add(Label lbl1=new Label("1st string", Label.CENTER);
```

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

2.7 Текстовые компоненты

Поля редактирования типа TextArea может использоваться как для вывода, так и для ввода и редактирования текста. Поля редактирования этого типа 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, за исключением упомянутых различий, во всем аналогичны друг другу.

Текстовые компоненты TextComponent

Так как классы TextArea и TextField являются подклассами TextComponent, то они могут пользоваться его методами. Рассмотрим некоторые методы класса TextComponent. Вывести текст в поле редактирования или получить текст из поля можно методами setText() и

getText() соответственно. Для выделения фрагмента текста, расположенного между указанными начальным и конечными символами, применяется метод select() (для выделения всего текста используется метод selectAll()), а для получения номеров первого и последнего выделенных символов - методы getSelectionStart() и getSelectionEnd(). Для получения выделенного в содержимом поля фрагмента текста используется метод getSelectedText(). Запретить или разрешить редактирование в поле можно при помощи метода setEditable(), а проверить, разрешено ли редактирование - методом isEditable().

Поле редактирования TextField

Для создания поля, в котором пользователь может ввести небольшой объем информации (одну строку текста), удобно пользоваться однострочным полем редактирования (текстовым полем) TextField. 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, т.е. символов, ввод которых на экране отображается каким-либо одним символом (эхо-символом), а не фактически вводимыми символами. Для установки такого 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

Для обработки событий от текстовых полей (нажатие кнопки «Enter» свидетельствует о прекращении ввода информации) следует внести в метод action() контейнера, содержащего эти текстовые поля, следующий фрагмент:

```
// процесс обработки событий от текстовых полей
   if(evt.target instanceof TextField)
{// получаем ссылку на текстовое поле, вызвавшее событие
            TextField field=(TextField)evt.target;
// проверка, от какого именно поля пришло событие и
// обработка этого события от конкретного поля
            .....
return true; // если событие обработано, иначе return false}
```

Поле редактирования TextArea

Для ввода большого объема информации применяются многострочные поля редактирования (текстовые области) TextArea. Действия пользователя с данными компонентами не генерируют сообщений ACTION_EVENT. Для создания этих полей существует большой набор конструкторов, при их помощи можно создавать объекты с указанным числом строк и числом символов в строке, просто поле с указанным текстом, а также поля, обладающие всеми этими параметрами, например:

```
// поле высотой 5 строк и шириной 20 символов
add(TextArea tArea1=new TextArea(5,20));
// поле, инициализированное строкой "Enter text"
add(TextArea tArea2=new TextArea("Enter text"));
// поле высотой 10 строк и шириной 15 символов,
// инициализированное строкой "Enter text"
add(TextArea tArea3=new TextArea("Enter text",10,15));
```

Получить количество строк и число символов в строке текстовой области можно методами getRows() и getColums().

Для добавления некоторого текста в конец содержимого текстовой области используется метод appendText(), для замены текста, расположенного между символами с указанными номерами, на другой текст - метод replaceText(), а для вставки некоторого текста после символа с казанным номером - метод insertText().

3. Приложение AllElements

Задание. Создать апплет двойного назначения AllElements, в окне которого размещены различные элементы управления. При действия пользователя с этими компонентами в панели состояния должны отображаться соответствующие надписи.

Методические указания. Апплет должен быть создан на основе шаблонов, содержащихся в Приложении 4 (или при помощи системы Java Applet Wizard)**.**

Объявление элементов класса апплета.

В классе апплета объявим следующие элементы - ссылки на объекты классов элементов управления:

```
Button btn; // кнопка
Checkbox chck; // независимый переключатель
Checkbox chck1,chck2; // радио-переключатели
Choice mn; // меню выбора - выпадающий список
List lst; // раскрывающийся список
Scrollbar scrllbr; // линейка прокрутки
Label lbl; // метка
```

TextField txt; // текстовое поле

TextArea area; // текстовая область

Инициализация апплета (метод init()).

При инициализации апплета в методе init() создадим компоненты и введем их в апплет:

- Создать новую кнопку с надписью "Click!". Ссылку нее присвоить переменной btn. Ввести btn в апплет.
- Создать новый независимый переключатель с надписью "Undependent checkbox". Ссылку него присвоить переменной chck. Ввести chck в апплет.
- Создать группу переключателей:

CheckboxGroup gr=new CheckboxGroup();

Создать новый переключатель с надписью "1st checkbox from group", включенный в группу gr, установленный во включенное состояние. Ссылку него присвоить переменной chck1. Ввести chck1 в апплет.

Создать новый переключатель с надписью "2nd checkbox from group", включенный в группу gr, установленный во выключенное состояние. Ссылку него присвоить переменной chck2. Ввести chck2 в апплет.

- Создать новое пустое меню выбора (выпадающий список). Ссылку него присвоить переменной mn. Ввести mn в апплет. Добавить в меню mn пункты "1st item", "2nd item", "3rd item".
- Создать новый пустой раскрывающийся список, имеющий видимую часть в 3 пункта и не имеющий возможности множественного выбора. Ссылку него присвоить переменной lst. Ввести lst в апплет. Добавить в список lst пункты "1st item", "2nd item", "3rd item", "4th item".
- Создать новую горизонтальную линейку прокрутки шириной 100, минимальное значение 1, максимальное 100, текущее положение бегунка 10. Ссылку нее присвоить переменной scrllbr. Ввести scrllbr в апплет.
- Создать новую метку с надписью "Label" и выравниванием по центру. Ссылку нее присвоить переменной lbl. Ввести lbl в апплет.
- Создать новое текстовое поле шириной 30 символов, инициализированное строкой "Enter text into textfield". Ссылку него присвоить переменной txt. Ввести txt в апплет.
- Создать новую текстовую область шириной 40 символов и высотой 5 строк, инициализированное строкой "Enter text into textarea". Ссылку него присвоить переменной area. Ввести area в апплет.

Отрисовка содержимого окна (метод paint()).

Так как апплет не выводит в окно никаких графических изображений, то в этом методе не делается никаких действий.

Обработка событий (метод handleEvent()).

Рекомендация. Если контейнер содержит элементы управления, события от которых не обрабатываются методом action() (методом action() нельзя обработать события от линейки прокрутки и события выделения и отмены выделения пунктов раскрывающегося списка), то рекомендуется обработку всех событий от всех компонентов производить в методе handleEvent(), чтобы не делить процесс обработки на части, а обрабатывать события централизовано. Процесс обработки событий в методе handleEvent() аналогичен процессу обработки в методе action().

Переопределим в класса апплета метод handleEvent(), создав заготовку следующего вида (эту заготовку можно использовать в качестве шаблона метода для обработки событий от элементов управления в других приложениях):

```
public boolean handleEvent(Event evt)
{  switch(evt.id)
   {        default:// передача сообщения на обработку
                    // методу базового класса
                        return super.handleEvent(evt);
                    case Event.ACTION_EVENT:
                    // обработка событий, которые можно
                        // обработать и методом action()
                        break;
                    case Event.LIST_SELECT:
                    // обработка выделения пунктов списка
   break;
                    case Event.LIST_DESELECT:
              // обработка отмены выделения пунктов списка
                        break;
                    case Event.SCROLL_LINE_UP:
              // обработка изменения положения бегунка
              // линейки прокрутки (полосы просмотра)
                        break;
   case Event.SCROLL_LINE_DOWN:
              // обработка изменения положения бегунка
              // линейки прокрутки (полосы просмотра)
```

```
                    break;
                case Event.SCROLL_PAGE_UP:
            // обработка изменения положения бегунка
            // линейки прокрутки (полосы просмотра)
                    break;
                case Event.SCROLL_PAGE_DOWN:
            // обработка изменения положения бегунка
            // линейки прокрутки (полосы просмотра)
                    break;          }
            return true;        }
```

Теперь приступим к обработке событий от компонент (модификации заготовки метода handleEvent()).

Событие Event.ACTION_EVENT (данное событие приходит при нажатии на кнопку мышью; при завершение ввода строки в текстовом поле нажатием клавиши «Enter»; при выборе элемента из списка двойным щелчком мыши; при изменении состояния переключателя щелком мыши или клавишей пробела; при выборе элемента из меню выбора одинарным щелчком мыши).

- Если событие вызвал компонент класса Button (*результат операции evt.target instanceof Button равен true*), то: 1) получить текст кнопки и присвоить ссылку на него переменной text класса String; 2) если сообщение пришло от кнопки btn (*метод evt.target.equals(btn) возвращает значение true*), то выдать сообщение *showStatus("Pressed button : <<"+text+">>")*, иначе возвратить false (сообщение пришло от неизвестной кнопки).

Например:

```
if(evt.target instanceof Button) // событие вызвал компонент Button?
    {//1.получить текст кнопки
                String
    text=((Button)evt.target).getLabel();
if(evt.target.equals(btn))
    // 2. сообщение от btn?
    //сообщение в панели состояния
    showStatus("Pressed button :
     <<"+text+">>");
                else
return      false;      //      сообщение      от      неизвестной      кнопки
            }
```

- Если событие вызвал компонент класса Choice, то: 1) получить текст переключателя и присвоить ссылку на него переменной text класса String; 2) получить состояние переключателя и присвоить его переменной ch типа boolean; 3) если сообщение пришло от переключателя chck, то выдать сообщение *showStatus("State of "+text+" is <<"+ch+">>")*; иначе, если сообщение пришло от переключателя chck1, то выдать сообщение *"State of "+text+" is <<"+ch+">>"+" but "+"State of "+chck2.getLabel()+" is <<"+chck2.getState()+">>"*; иначе, если сообщение пришло от переключателя chck2, то выдать сообщение *"State of "+text+" is <<"+ch+">>"+" but "+"State of "+chck1.getLabel()+" is <<"+chck1.getState()+">>"*; иначе возвратить false (сообщение пришло от неизвестного переключателя).

Например:

```
if(evt.target instanceof Checkbox) // событие вызвал компонент Checkbox?
{       оператор; // 1. получить текст переключателя
        оператор; // 2. получить состояние переключателя
        if(evt.target.equals(chck)) // 3. сообщение от chck?
        оператор;
        else    if(evt.target.equals(chck1)) // сообщение от chck1?
        оператор;
        else    if(evt.target.equals(chck2)) // сообщение от chck2?
        оператор;
return false; // сообщение от неизвестного флажка
                }
```

- Если событие вызвал компонент класса Choice, то: 1) получить текст выбранного пункта и присвоить ссылку на него переменной text класса String; 2) если сообщение пришло от меню mn, то выдать сообщение *showStatus("Choice from menu: <<"+text+">>")*, иначе возвратить false (сообщение пришло от неизвестного меню выбора).

- Если событие вызвал компонент класса List, то: 1) получить текст выбранного пункта и присвоить ссылку на него переменной text класса String; 2) если сообщение пришло от списка lst, то выдать сообщение *showStatus("Choice from list: <<"+text+">>")*, иначе возвратить false (сообщение пришло от неизвестного списка).

- Если событие вызвал компонент класса TextField, то: 1) получить содержимое текстового поля и присвоить ссылку на него переменной text класса String; 2) если сообщение пришло от текстового поля txt, то выдать сообщение *showStatus("In textfield entered : <<"+text+">>")*, иначе возвратить false (сообщение пришло от неизвестного текстового поля).

Событие Event.LIST_SELECT (данное событие приходит при выделении пункта списка). Если событие вызвал компонент класса List, то: 1) получить текст выделенного пункта и присвоить ссылку на него переменной text класса String; 2) если сообщение пришло от списка lst, то выдать сообщение *showStatus("In list selected: <<"+text+">>")*, иначе возвратить false (сообщение пришло от неизвестного списка).

Событие Event.LIST_DESELECT (данное событие приходит при отмене выделении пункта списка). Если событие вызвал компонент класса List, то если сообщение пришло от списка lst, то выдать сообщение *showStatus("Deselected item from list")*, иначе возвратить false (сообщение пришло от неизвестного списка).

Событие Event.SCROLL_LINE_UP (данное событие приходит при изменении положения бегунка полосы просмотра). Если событие вызвал компонент класса Scrollbar, то: 1) получить текущее положение бегунка и присвоить его переменной pos типа int; 2) если сообщение пришло от полосы просмотра scrllbr, то выдать сообщение *showStatus("Event - LineUp, current position of scrollbar: <<"+pos+">>")*, иначе возвратить false (сообщение пришло от неизвестной полосы просмотра).

Событие Event.SCROLL_LINE_DOWN (данное событие приходит при изменении положения бегунка полосы просмотра). Если событие вызвал компонент класса Scrollbar, то: 1) получить текущее положение бегунка и присвоить его переменной pos типа int; 2) если сообщение пришло от полосы просмотра scrllbr, то выдать сообщение *showStatus("Event - LineDown, current position of scrollbar: <<"+pos+">>")*, иначе возвратить false (сообщение пришло от неизвестной полосы просмотра).

Событие Event.SCROLL_PAGE_UP (данное событие приходит при изменении положения бегунка полосы росмотра). Если событие вызвал компонент класса Scrollbar, то: 1) получить текущее положение бегунка и присвоить его переменной pos типа int; 2) если сообщение пришло от полосы просмотра scrllbr, то выдать сообщение *showStatus("Event - PageUp, current position of scrollbar: <<"+pos+">>")*, иначе возвратить false (сообщение пришло от неизвестной полосы просмотра).

Событие Event.SCROLL_PAGE_DOWN (данное событие приходит при изменении положения бегунка полосы просмотра). Если событие вызвал компонент класса Scrollbar, то: 1) получить текущее положение бегунка и присвоить его переменной pos типа int; 2) если сообщение пришло от полосы просмотра scrllbr, то выдать сообщение *showStatus("Event - PageDown, current position of scrollbar: <<"+pos+">>")*, иначе возвратить false (сообщение пришло от неизвестной полосы просмотра).

Задания к лабораторной работе

Задание 1. Создать апплет *AllElements* и объяснить его работу. Апплет должен иметь возможность работать как независимое приложение.

Задание 2. Дать ответы на контрольные вопросы.

Контрольные вопросы

27. Что такое GUI?
28. Какие два типа компонентов GUI существует?
29. Что такое элементы управления и что такое контейнеры?
30. Какие классы элементов управления существуют?
31. Что необходимо сделать, чтобы ввести компонент в контейнер?
32. Как можно перехватить и обработать события, пришедшие от компонентов?
33. Какие события от компонентов можно обработать при помощи метода action()?
34. Если событие не обрабатывается методом action(), то какой метод используется для обработки события?
35. Какие типы переключателей существуют?
36. Как несколько переключателей объединить в группу?
37. Чем отличаются выпадающие и раскрывающиеся списки? Как осуществляется в них выбор элементов?
38. При помощи какого метода обрабатываются сообщения от линейки прокрутки?
39. Порождают ли события компоненты класса Label и для чего используются эти компоненты? В чем преимущества их использования перед обычным выводом текста методами класса Graphics?
40. Что такое текстовые поля и текстовые области? Чем они отличаются?

ЛАБОРАТОРНАЯ РАБОТА № 5.
КОНТЕЙНЕРЫ КОМПОНЕНТОВ GUI И МЕНЕДЖЕРЫ РАЗМЕЩЕНИЯ
(4 часа)

Методические указания к лабораторной работе

Контейнеры - это объекты (компоненты), позволяющие помещать в себя другие различные компоненты.

1. Контейнеры

Класс контейнеров Container - подкласс класса Component. Существует два вида основных вида контейнеров: *панели* (класс Panel, подклассом которого является класс Applet) и

окна (класс Window, подклассами которого являются Frame и Dialog). Контейнеры имеют следующие деревья наследования:

Object → Component → Container → Panel

Object → Component → Container → Panel → Applet

Object → Component → Container → Window

Object → Component → Container → Window → Frame

Object → Component → Container → Window → Dialog

Object → Component → Container → Window → Dialog → FileDialog

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.

Пространство, занимаемое контейнерами, может быть разделено с использованием одного из менеджеров компоновки (менеджеров размещения). 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 на менеджер компоновки данного контейнера
- setLayout - Устанавливает менеджер компоновки данного контейнера
- layout - Выполняет размещение компонент внутри контейнера
- prefferedSize - Возвращает предпочтительный размер контейнера
- minimumSize - Возвращает минимальный размер контейнера
- paintComponents - Отображает компоненты контейнера
- deliverEvent - Отыскивает нужный компонент и доставляет ему событие
- locate - Возвращает компонент, содержащий заданную точку

- insets - Возвращает вкладки контейнера. Они показывают размер границ контейнера. Например, у фрейма будет верхняя вкладка, высота которой соответствует высоте строки заголовка.

1.1 Панели

Панель (класс Panel) является наиболее общим видом контейнеров в Java. Панель можно использовать как внутри другого контейнера (например, фрейма или апплета), так и непосредственно в окне WWW-броузера**.** Когда интерфейс состоит из большого количества элементов, почти всегда есть смысл объединить группы связанных по смыслу элементов с помощью панелей. Панель может иметь свой собственный менеджер размещения (по умолчанию это FlowLayout), независимый от менеджера размещения контейнера, в который эта панель входит.

Класс Panel имеет подкласс Applet, так что апплеты рассматриваются как расширение класса панелей. Панели очень часто используются в апплетах, имеющий сложный графический интерфейс пользователя.

Создание панелей

Панель создается достаточно просто. 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)); // установка менеджера размещения
```

При использовании такого менеджера размещения панели будут размещаться в ячейках таблицы, состоящей из одного столбца и двух строк.

Далее нужно создать объекты класса Panel:

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

Добавление панелей в контейнеры

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

```
add(top); // ввод верхней панели в контейнер (например, в апплет)
add(bottom); // ввод нижней панели в контейнер (например, в апплет)
```

Нужно заметить, что можно добавлять панели в панели, применяя к ним аналогичные действия. Например, добавим в верхнюю панель две панели:

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

Добавление компонент в панели

Для добавления компонент в панели (элементов управления или других контейнеров) необходимо указывать, в какую панель вводится тот или иной компонент, например:

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

Рисование в окне панели

Для того, чтобы что-то нарисовать, необходимо сначала получить контекст отображения окна. Методу paint() компонент передается необходимый контекст отображения через параметр.

Первый способ рисования в панелях заключается в том, что в методе paint() контейнера, содержащего эту панель, можно получить контекст отображения панели, а затем приступить к рисованию. Например, рассмотрим процесс рисования в панели DrawPanel в методе paint() апплета:

```
public void paint(Graphic g) // метод апплета, g - контекст апплета
{       // gDrawPanel контекст панели DrawPanel
        Graphics gDrawPanel=DrawPanel.getGraphics();
        // теперь можно рисовать в окне панели
        gDrawPanel.drawString("Text inside panel",10,10);
}
```

Второй способ рисования в панелях основан на создании собственного класса на базе класса Panel и переопределения в этом классе метода paint(). Естественно, этому методу paint() передается в качестве параметра контекст отображения панели.

Обработка и передача событий

Когда компонент, расположенный внутри панели, получает событие, этому компоненту первому предоставляется возможность его обработать. Если компонент не обработает событие до конца (и не запретит другим компонентам продолжать обработку), то событие

передается другим компонентам панели и, в конце концов, - панели. Если панель находится внутри другой панели и не запретит дальнейшую обработку события, то после завершения обработки внутренней панелью событие перейдет ко внешней панели. Однако, если существует множество панелей, вложенных одна в другую, передача событий может усложниться. Например, существует пять панелей, одна внутри другой: panel5 находится внутри panel4, panel4 - внутри panel3, panel3 - внутри panel2, panel2 - внутри panel1, а panel1 - внутри апплета. При нажатии на кнопку на panel5 первой событие нажатия кнопки получит сама кнопка, Затем, если обработка события не была прервана, то оно перейдет к panel5, и дальнейшую обработку события делает panel5. Если panel5 не прервет дальнейшую обработку, то событие затем перейдет к panel4 и так до тех пор, пока не дойдет до апплета.

При использовании вложенных панелей важно понять, что порядок их добавления определяет их иерархию. Иерархия обработки событий обратна последовательности добавления компонентов. Если событие не было до конца обработано ни одним компонентом и даже апплетом, то оно передается системе управления, в которой выполняется данная программа. Эта система отвечает за конечную обработку события и его завершение.

1.2 Окна

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

Непосредственно класс Window никогда не используется, а используются три его подкласса Frame, Dialog и FileDialog. Каждый их этих подклассов содержит все функции исходного класса Window, добавляя к ним несколько специфических свойств (см. описание методов этих подклассов ниже).

Некоторые методы класса Window:

- Window - Конструктор, который создает окно, первоначально не видимое. Окно ведет себя как модальное диалоговое окно, т.е. пока это окно отображается, оно блокирует ввод в другие окна. В качестве параметра конструктору передается владелец окна - объект класса Frame
- 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

Точно также, как и в случае с панелями, можно использовать любые методы контейнера применительно к окнам. Например, созданное окно или диалог можно переместить с помощью метода move() или изменить его размеры методом resize().

При создании окна обязательным параметром конструктора Window() является объект класса Frame. Этот объект можно создать непосредственно при вызове конструктора окна. Так как окна изначально создаются невидимыми, отобразить их на экране можно, лишь вызвав метод show(). Правда, перед этим иногда полезно придать окну нужные размеры и разместить окно в нужной позиции. Если перемещать окно в видимом состоянии, то это может вызвать неприятное мерцание на экране. Приведем пример создания окна и отображения окна:

```
Window win=new Window(new Frame)); // создание окна
win.resize(200,300); // изменение его размеров
win.move(50,50); // перемещение окна
win.show(); // отображение окна
```

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

1.3 Рамки, фреймы

Фрейм (класс Frame) - это объект, который может существовать без всякой связи с окном WWW-броузера. С помощью класса Frame можно организовать интерфейс независимого апплета. Окно, созданное на базе класса Frame, больше всего похоже на главное окно обычного приложения Windows, и может пользоваться большим количеством методов класса Frame. Оно автоматически может иметь главное меню (так как класс Frame реализует интерфейс MenuContainer), для него можно устанавливать форму курсора и пиктограмму. Внутри такого окна можно рисовать. Так как окно класса Frame произошло от класса Container, то в него можно добавлять различные компоненты и панели, так же как в делается для апплетов и панелей. Нужно отметить, что по умолчанию для окон класса Frame устанавливается режим размещения BorderLayout.

Некоторые методы класса Frame:

- 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 указанную строку меню

- 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

Если необходимо создать фрейм только ради меню, то стоит воспользоваться стандартным объектом класса Frame:

```
Frame fr=new Frame("Title for frame");
```

Однако наибольшую пользу от применения фреймов можно извлечь, после создания подклассов этого класса, инкапсулирующих требуемое поведение и состояние. Тогда всякий раз, когда понадобятся рамки и окна, можно просто создать объекты новых классов.

Создание подкласса Frame.

Для создания подкласса класса Frame необходимо определить новый класс, унаследовав его от класса Frame:

```
class MainFrameWnd extends Frame // объявление нового класса фрейма
{
// конструктор для создания фрейма с заголовком
        public MainFrameWnd(String str)
        {
// обязательный вызов конструктора суперкласса для
        // правильного создания фрейма
super(str); // этот вызов должен быть самым первым оператором
// задание необходимых размеров окна фрейма
                resize(400,200);
// здесь можно определить различные параметры фрейма,
// например, форму курсора, пиктограмму, задать меню и др.
                .....
        }
// обработка событий фреймом MainFrameWnd, например
// таких, как закрытие окна, события меню и т.д.
        public boolean handleEvent(Event evt)
        {       switch (evt.id)
{       case Event.WINDOW_DESTROY:
    // --------------------
```

```
// различные завершающие действия
                                        .....
// удаление окна
            dispose();
// если это класс главных окон, то завершение
// всей программы методом System.exit(0);
              return true;
              default:
        // ---------------------------------------
// передача необработанных действий
              return super.handleEvent(evt);
                        }
           return true;
             }
}
```

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

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

Меню

Меню являются стандартным элементом интерфейса современных программ. Меню Java генерируют события, для которых программист создает обработчики. Классы MenuBar и Menu позволяют конструировать удобные меню и интегрировать их в структуру апплета. Все фреймы автоматически реализуют интерфейс MenuContainer, поэтому, пожалуй, они являются единственным классом контейнеров, в которых стоит использовать меню. Теоретически же меню может быть добавлено в любой контейнер. Для того, что создать и добавить меню в контейнер, необходимо сначала создать панель меню (объект класса MenuBar), затем добавить в нее меню (объект класса Menu). В выпадающие меню класса Menu также можно добавлять элементы (строки или объекты класса MenuItem). Подготовленную панель меню теперь можно ввести в контейнер его методом add().

Создание панели меню и добавление ее в контейнер.

Контейнер может иметь главное меню или, как говорят, панель или строку меню. Главное меню создается на основе класса MenuBar.

Некоторые методы класса MenuBar:

- add - Добавляет меню (объект класса Menu) в данную панель меню

- countMenus - Определение количества меню в данной панели меню
- getMenu - Получение ссылки на меню
- remove - Удаление меню
- setHelpMenu - Установка меню Help
- getHelpMenu - Получение ссылки на меню Help

Для формирования главного меню создается объект класса MenuBar, а затем в него добавляются отдельные меню:

```
MenuBar mainMenu=new MenuBar(); // создание панели меню
Menu fileMenu=new Menu("File"); // создание меню "File"
Menu helpMenu=new Menu("Help"); // создание меню "Help"
```

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

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

Сформированные отдельные меню добавляются затем к панели меню при помощи метода add() класса MenuBar:

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

Меню добавляются в панель слева направо в том порядке, в каком порядке происходят вызовы метода add() класса MenuBar. Класс MenuBar предоставляет интересную возможность присвоения одному из меню статуса «меню помощи», которая осуществляется следующим образом:

```
// присвоение меню "Help" статуса «меню помощи»
mainMenu.setHelpMenu(helpMenu);
```

Метод setHelpMenu() превращает меню helpMenu в меню помощи, перенося его в правый конец панели, отдельно от других меню. Следует обратить внимание, что, прежде чем вызывать метод setHelpMenu(), необходимо добавить это меню в панель меню.

И наконец, теперь можно установить главное меню окна фрейма:

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

Создание меню и его элементов.

При создании меню необходимо реализовать класс Menu для каждого выпадающего меню и класс MenuItem (или его подкласс) для каждого элемента в этом меню.

Некоторые методы класса Menu:

- add - Добавляет элемент (объект класса MenuItem (строку или вложенное меню) или просто строку) в данное меню
- addSeparator - Добавляет горизонтальный разделитель
- countItems - Определение количества элементов в данном меню
- getItem - Получение ссылки на элемент меню
- remove - Удаление элемента меню
- isTearOff() - Проверка того, остается ли меню на экране после того, как пользователь отпустил клавишу мыши

Вот как выглядит объявление и создание нового меню:

```
Menu mn=new Menu("Operation");
```

Здесь создается меню с заголовком "Operation" и сохраняется ссылка на него в переменной mn. Есть еще одна интересная возможность - объявление «западающего меню». Это значит, что такое меню останется открытым даже после того, как кнопка мыши будет отпущена. «Западающее меню» создается с помощью следующего конструктора

```
new Menu("Operation",true);
```

Теперь создадим элемент cmd меню mn (команды включаются в меню в направлении сверху вниз):

```
MenuItem cmd=new MenuItem("Command"); // создание элемента cmd
mn.add(cmd); // добавление элемента cmd в меню mn
```

Так для доступа к пункту меню достаточно его заголовка, то эти два оператора можно объединить:

```
// добавление элемента "Command" в меню mn
mn.add(new MenuItem("Command"));
```

В меню можно добавлять не только команды, но и *вложенные меню* благодаря тому факту, что класс Menu является всего лишь расширением класса MenuItem:

```
// добавление вложенного меню в меню mn
mn.add(new Menu("Other menu inside mn"));
```

Классы элементов меню.

Класс MenuItem 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 MenuItem:

- disable - Блокирование элемента меню
- enable - Разблокирование элемента меню. Другая версия этого метода осуществляет блокировку и разблокировку по условию
- getLabel - Получение текстовой строки элемента меню
- isEnable - Проверка того, является ли элемент заблокированным
- setLabel - Установка текстовой строки элемента меню

Так как класс Menu является всего лишь расширением класса MenuItem, то объекты класса Menu также могут пользоваться методами класса MenuItem.

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

Обработка событий меню.

Когда пользователь выбирает одну из команд меню, происходит генерация соответствующего события. При этом вызывается метод action() контейнера. Каждый элемент меню создает свое индивидуально событие, так что, переопределив метод action(), всегда можно однозначно сказать, какой элемент меню в каком меню был выбран. Можно проверить, является ли элемент, создавший событие экземпляром класса MenuItem, и если так, то можно использовать аргумент arg типа Object события, чтобы извлечь имя выбранного элемента меню. Допустим, что панель меню состоит из меню "File", в которое входят элементы "New" и "Exit". Тогда метод action() контейнера, содержащего эту панель меню, можно переопределить следующим образом:

```
   public boolean action(Event evt, Object obj)
   {
// обработка событий от элементов меню
            if(evt.target instanceof MenuItem) // объект является элементом меню?
{  // получаем название/метку элемента меню
            String label=(String)evt.arg;
// проверка, от какого именно элемента пришло событие
            if(label.equals("New")) // от элемента с меткой "New"?
{  // обработка события от элемента с меткой "New"
            .....                                    }
```

```
        else     if(label.equals("Exit")) // от элемента с меткой "Exit"?
{  // обработка события от элемента с меткой "Exit"
        .....                                   }
                        else
{  // если событие вызвано элементом, обработчик
// которого не предусмотрен в данном методе
        return false;                           }
// если событие обработано, иначе return false
        return true;                        }
// необработанные события передаем на дальнейшую обработку
        return false;               }
```

Для обработки событий от меню можно переопределять не только метод action(). Вместо переопределения метода action() можно переопределить более универсальный обработчик событий handleEvent():

```
   public boolean handleEvent(Event evt)
   {        switch(evt.id)
        {       default:// передача сообщения на обработку
// методу базового класса
        return super.handleEvent(evt);
        case Event.ACTION_EVENT:
// обработка событий от элементов меню
        if(evt.target instanceof MenuItem)
{  // получаем название/метку элемента меню
        String label=(String)evt.arg;
// проверка, от какого элемента пришло событие
        if(label.equals("New"))
{  // обработка события
        .....                           }
        else if(label.equals("Exit"))
{  // обработка события
      .....                             }
        else
{  // если событие вызвано элементом,
// обработчик которого не
// предусмотрен в данном методе
```

```
            return false;                    }
    // если событие обработано, иначе return false
            return true;                     }
                    break;                   }
            return true;     }
```

Диалоги

Кроме окон и фреймов есть особый подкласс окон, называемый диалоговыми окнами или просто диалоги. В отличие от фреймов или обычных окон диалог предоставляет пользователю окно, в котором нужно выполнить те или иные действия, лишь по завершении которых пользователь сможет продолжить работу с программой.

Диалоговые окна.

Диалоговые окна (класс Dialog) используются в основном 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 являются реализацией интерфейса MenuContainer (а следовательно не реализуют панель меню автоматически); во-вторых, они могут иметь *модальность* - это значит, что можно сконструировать диалоговое окно, которое запретит пользователю обращаться к другим окнам (включая и окно WWW-навигатора) до тех пор, пока пользователь не произведет требуемого действия в этом диалоговом окне.

Некоторые методы класса Dialog:

- 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

При создании диалогового окна его модальность задается указанием особого параметра вызова конструктора. Таким образом, диалоговое окно дает выбор: позволить ли пользователю продолжить работу, не обращая внимания га диалоговое окно, или же потребовать от него особой реакции на сообщение в этом окне.

Для окон класса Dialog 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 show(). Чтобы спрятать окно, необходимо использовать метод hide(). Метод dispose удаляет окно диалоговой панели окончательно и освобождает все связанные с ним ресурсы.

Когда пользователь пытается уничтожить окно диалоговой панели при помощи органов управления, расположенных в заголовке такого окна, возникает событие Event.WINDOW_DESTROY. Необходимо обработать это сообщение, обеспечив удаление окна диалоговой панели вызовом метода dispose(), если это, конечно, соответствует логике работы этого диалога.

Диалоговые окна удобны для приглашения пользователя к выполнению какого-либо действия и для подтверждения того, что пользователь ознакомился с сообщением программы. Диалоговые окна в отличие от фреймов не реализуют панель меню автоматически. Поскольку диалоговые окна обычно меньше по размерам, чем фреймы, они бывают удобны для быстрого диалога с пользователем.

При работе с диалоговыми окнами важно помнить одно правило: 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.

Создание подкласса Dialog.

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

```
class MessageBox extends Dialog // объявление нового класса диалога
{
// конструктор для создания диалогового окна для вывода
// сообщения sMsg, заголовок окна передается через sTitle.
// родительский фрейм - через parent, модальность - через modal
public MessageBox(String sMsg,Frame parent,String sTitle,boolean modal)
        {
// обязательный вызов конструктора суперкласса для
// правильного создания окна диалога
// этот вызов должен быть самым первым оператором
super(parent,sTitle,modal);
// задание необходимых размеров окна диалога
resize(400,200);
// здесь можно создать все необходимые компоненты для
// размещения внутри диалоговой панели, а также установить
// нужный режим размещения этих компонент
```

```
    .....                    }
// обработка событий диалогом, например закрытие окна
    public boolean handleEvent(Event evt)
    {       switch (evt.id)
    {       case Event.WINDOW_DESTROY:
//  --------------------
// различные завершающие действия
    .....
// удаление окна
    dispose();
    return true;
    default:
//  ------------------------------
// передача необработанных действий
    return super.handleEvent(evt);
                    }
    return true;        }       }
```

Теперь можно создавать диалоги этого класса следующим оператором:

```
    MessageBox msgBox=new MessageBox("String of message",
    new Frame(),"Message box",true)
```

<u>Файловые диалоги.</u>

Кроме стандартного обобщенного класса диалога существует подкласс класса Dialog - специальный класс FileDialog, позволяющий открывать и сохранять файлы. Эти диалоги функционируют в соответствии с требованиями той платформы, на которых выполняется программа и с их помощью можно получить дескрипторы файлов для загрузки или сохранения через потоковый класс.

Рассмотрим примеры создания файловых диалогов:

```
    FileDialog getFile=new FileDialog(this,         // ссылка на окно родителя
    "Get a file...",    // заголовок диалога
    FileDialog.LOAD);           // тип диалога
    getFile.show();
    FileDialog saveFile=new FileDialog(this,"Save a file...",FileDialog.SAVE);
    saveFile.show();
```

2. 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. Каждый класс менеджера имеет методы и конструкторы, которые позволяют выбирать различные компоновки.

2.1 Типы менеджеров размещения

К простейшим менеджерам относятся FlowLayout и GridLayout, к более сложным - BorderLayout, CardLayout и GridBagLayout. *По умолчанию* в окне апплета и всех создаваемых панелях используется менеджер FlowLayout, а во фреймах и диалоговых окнах - менеджер BorderLayout. *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() При помощи этой ссылки на менеджер можно использовать методы менеджера для выполнения различных действий.

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 очень прост и 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. Содержимое каждого ряда выравнивается. 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

Менеджер FlowLayout не обеспечивает гибкого управления размещением компонентов на экране. В этом режиме компоненты просто последовательно размещаются слева направо, переходя на более нижний ряд, если в этом ряду больше нет места. Причем при изменении размера окна контейнера (когда происходит переразмещение компонентов) состав компонентов в ряду может изменится из-за того, что некоторые компоненты могут перейти в другой ряд (выше или ниже) в силу изменения ширины окна контейнера. 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 BorederLayout окно контейнера разделяется на рамку и центральную часть. При размещении компонентов указывается направление от центра окна, в котором следует размещать компоненты (компоненты размещаются вдоль границ окна контейнера). 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, при изменении размеров окна контейнера.

Менеджер размещения CardLayout

Этот менеджер размещения позволяет изменять набор компонентов, выводимых на экран, прямо во время работы апплета. Режим CardLayout предназначен для создания набора диалоговых панелей («блокнота»), которые можно показывать по очереди в одном окне. Этот менеджер (в отличии от ранее рассмотренных) 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 диалоговых панелей в режиме CardLayout используются отдельные органы управления, рас-

положенные в другой панели. Рассмотрим, как пользоваться режимом размещения CardLayout. Обычно в окне апплета создается две панели, одна из которых предназначена для страниц «блокнота» в режиме размещения CardLayout, а другая содержит органы управления перелистыванием страниц, например, кнопки. Такие методы как first(), last(), next() и previous(), позволят отображать соответственно первую, последнюю, следующую и предыдущую страницу «блокнота». 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 такой метод add(), при помощи которого можно вводить компонент, указывая название компонента и ссылку на сам компонент.

Менеджер размещения GridBagLayout

Менеджер GridLayout работает так, что в каждой ячейке может быть только один компонент. Иногда возникает необходимость разместить один компонент в нескольких ячейках. 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.

Выбор менеджера размещения

Рассмотрим краткие формулировки основных рекомендаций по применению каждого из рассмотренных менеджеров.

Что необходимо?	Что использовать?
Быстро разместить все компоненты, не обращая внимания на совершенство композиции	FlowLayout
Быстро и по возможности красиво распо-	BorderLayout

ложить все компоненты	
Разместить компоненты, имеющие одинаковый размер	GridLayout
Разместить компоненты, некоторые из которых имеют одинаковый размер	Рекомендуется использовать GridLayout в отдельной панели, собрав в нее компоненты, имеющие одинаковый размер, а все остальные компоненты вынести за пределы этой панели
Выводить компоненты на экран по мере необходимости	CardLayout
Отображать некоторые компоненты постоянно, а некоторые - по мере необходимости	Рекомендуется использовать CardLayout в панели, предназначенной для вывода компонентов по мере необходимости, а постоянно отображаемые компоненты вынести за пределы этой панели.
Иметь как можно больше контроля над расположением компонентов, а также обеспечить разумную реакцию на изменение размеров контейнера	GridBagLayout

При использовании в контейнере нескольких панелей вполне можно установить для каждой панели свой менеджер размещения, отличающийся от менеджеров соседних панелей и менеджера всего контейнера. Например, если в большой панели используется GridBagLayout, то для вложенной в нее панели меньших размеров вполне можно выбрать CardLayout.

Когда создается новый контейнер, ему присваивается менеджер размещения по умолчанию. Чтобы изменить менеджер размещения, действующий в данном контейнере, используется выражение типа

setLayout(new BorderLayout());

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());

Добавление компонентов к контейнеру, для которого установлен отличающийся от стандартного менеджер размещения, имеет только одно отличие от обычного способа, а именно - может понадобиться передавать методу add() контейнера какие-то параметры расположения, например:

leftPanel.add("North",radioField);

При использовании менеджера GridBagLayout параметры передаются в объекте специального класса GridBagConstraints, с помощью которого можно установить для каждого компонента значения определенных параметров расположения.

Как правило, компоненты добавляются один за другим в пределах одного горизонтального ряда, пока в этом ряду остается свободное место; следующий добавляемый компонент начинает следующий ряд. Этому правилу в каком-то смысле подчиняется даже BorderLayout - когда добавляется несколько компонентов с одним и тем же параметром расположения (например, "North"), они размещаются в своей части контейнера по тому же закону.

3. Поведение контейнера при наличии элементов управления

При использовании в контейнере элементов управления в поведении контейнера возникают некоторые странности.

Во-первых, при попадании указателя мыши на элементы управления возникает событие MOUSE_EXIT, а при переводе указателя назад в окно контейнера следует событие MOUSE_ENTER. Вывод: области, занятые элементами интерфейса, исключаются из окна контейнера, и события от мыши в этих областях не обрабатываются.

Во-вторых, не происходит ни одного вызова обработчиков нажатий и отпусканий клавиш клавиатуры. Это говорит о том, что контейнера не имеет фокуса ввода. Нет фокуса и у элементов интерфейса. Если нажать клавишу <Tab>, то передачи фокуса не происходит. Вывод: после создания контейнера фокуса ввода нет ни у одного элемента интерфейса.

Если необходимо, чтобы нажатия клавиш отслеживались аплетом, нужно сделать так, чтобы аплет получил фокус. Для получения фокуса ввода этого для контейнера нужно вызвать метод requestFocus().

Если, например, в методе init() в апплет ввести несколько элементов управления, и затем для апплета вызвать метод requestFocus(), тогда при нажатии на кнопки клавиатуры обработчики нажатий и отпусканий клавиш начнут вызываться. Если нажать клавишу <Tab>, фокус ввода перейдет к кнопке. При этом также вызываются обработчики lostFocus() и gotFocus(), отмечающие потерю фокуса ввода одним элементом и получение его другим.

4. Приложение PanelsDemo1

Задание. Создать апплет двойного назначения PanelsDemo1, в окне которого создаются две панели, расположенные горизонтально (для окна апплета используется менеджер размещения GridLayout). Первая из панелей используется как «блокнот», на каждой странице которого находится кнопка (всего 3 «страницы», панель использует менеджер размещения CardLayout). Вторая панель содержит две управляющие кнопки «Previous» и «Next», позволяющие перебирать страницы «блокнота» по очереди. При нажатии на кнопки первой панели в строке состояния навигатора должно отображаться название нажатой кнопки.

Методические указания. Апплет должен быть создан на основе шаблонов, содержащихся в Приложении 4 (или при помощи системы Java Applet Wizard)**.**

Объявление элементов класса апплета.

В классе апплета объявим следующие элементы - ссылки на объекты классов:

Panel cardPanel, controlPanel;

Button btn1,btn2,btn3,btnPrev,btnNext;

В поле cardPanel будет храниться ссылка на панель «блокнота», страницы которого содержат кнопки. Эта панель будет располагаться в верхней части окна апплета.

Поле controlPanel будет использоваться для хранения ссылки на панель кнопок, предназначенных для перелистывания страниц «блокнота» cardPanel.

Ссылки на кнопки, расположенные на страницах «блокнота», будут сохраняться в полях btn1, btn2 и btn3.

Ссылки на кнопки, предназначенные для перелистывания страниц, будут записываться в поля btnPrev и btnNext.

Инициализация апплета (метод init()).

В этом методе необходимо создать панели и добавить в них компоненты. Сначала установим желтый (Color.yellow) цвет фона окна апплета методом setBacground(). Затем выберем для окна апплета схему размещения GridLayout (сетка 2x1), используя метод setLayout() апплета.

Создание панелей апплета.

Создадим объекты cardPanel и controlPanel класса Panel. Затем установим для панелей менеджеры размещения (вызывая метод setLayout() для объектов cardPanel и controlPanel): для панели cardPanel - менеджер размещения CardLayout, а для панели controlPanel - менеджер FlowLayout.

Создание кнопок и ввод их на страницы «блокнота».

Создадим кнопки btn1, btn2 и btn3 с надписями "1st button", "2nd button" и "3rd button" соответственно. После создания кнопок добавим их по очереди в панель cardPanel, заполняя страницы «блокнота».

Создание кнопок и ввод их в панель управления.

Создадим кнопки btnPrev и btnNext с надписями "Previous" и " Next " соответственно. После создания кнопок добавим их по очереди в панель controlPanel.

Ввод панелей в апплет. После завершения формирования панелей cardPanel и controlPanel добавим их по очереди в апплет при помощи метода add() аплета.

Отрисовка содержимого окна (метод paint()).

Так как апплет не выводит в окно никаких графических изображений, то в этом методе не делается никаких действий.

Обработка событий (метод action()).

Переопределим в класса апплета метод action(), вызываемый при нажатиях на кнопку (и при некоторых других действиях пользователя). В данном случае он будет обрабатывать события, поступающие от кнопок, расположенных в обеих панелях. Сначала создадим шаблон этого метода:

```
public boolean action(Event evt, Object obj)
{
        // обработка событий от кнопок
        if(evt.target instanceof Button) // объект является кнопкой?
        {       // получаем ссылку на кнопку, вызвавшую событие
                Button btn=(Button)evt.target;
                // получаем метку кнопки
                String lbl=btn.getLabel();
                // проверка, от какой именно кнопки пришло событие
                // и обработка события от этой кнопки
                // .....
                return true; // если событие обработано, иначе return false
        }
        // необработанные события передаем на дальнейшую обработку
        return false;
}
```

События от панели cardPanel.

Проверяя, какой именно кнопке равен объект btn, необходимо выполнить следующие действия: при нажатии кнопок btn1, btn2 и btn3 в панели состояния отобразить при помощи

функции showStatus() название кнопки, а при нажатии на кнопки btnPrev и btnNext отобразить соответствующую страницу «блокнота». Проверка равенства объекта btn другому объекту осуществляется следующим образом:

```
if(btn.equals(btn1)) // от кнопки с номером 1?
{       // обработка события от кнопки 1
        // .....
}
```

<u>События от панели controlPanel.</u>

Если нажата кнопка btnPrev, то сначала с помощью метода getLayout() получаем ссылку на используемый для панели cardPanel менеджер компоновки, а затем вызываем метод previous() для отображения предыдущей страницы панели cardPanel:

```
CardLayout cl=(CardLayout)cardPanel.getLayout();
cl.previous(cardPanel);
```

Аналогичные действия сделаем, если нажата кнопка btnNext, но для отображения следующей страницы вызовем метод next() класса CardLayout.

5. Приложение PanelsDemo2

Задание. Создать апплет двойного назначения PanelsDemo2, в окне которого создаются три панели, расположенные горизонтально (для окна апплета используется менеджер размещения GridLayout) и имеющие рамку по периметру,. В верхней панели рисуется строка текста «Example of Font, Background and Foreground Colors». Средняя панель используется как «блокнот», предназначенный для выбора цвета фона, цвета изображения и шрифта для верхней панели(всего 3 «страницы», панель использует менеджер размещения CardLayout). Нижняя панель содержит управляющие кнопки «Background Color», «Foreground Color» и «Set Text Font», а также «Previous» и «Next», позволяющие перебирать страницы «блокнота».

Методические указания. Апплет должен быть создан на основе шаблонов, содержащихся в Приложении 4 (или при помощи системы Java Applet Wizard)**.**

<u>Объявление элементов класса апплета.</u>

В классе апплета объявим следующие элементы - ссылки на объекты классов:

```
// верхняя, средняя и нижняя панели
Panel drawPanel; // панель для рисования
Panel cardPanel; // панель для "блокнота"
Panel controlPanel; // панель для кнопок управления
// панели страниц "блокнота"
Panel backgrPanel,foregrPanel,fontPanel;
// меню выбора для панелей страниц "блокнота"
```

```
Choice chBackgr,chForegr,chFont;
// кнопки панели управления
Button btnPrev,btnNext,btnBackgr,btnForegr,btnFont;
// строка для хранения названия выбранного шрифта
String fontName;
```

Ссылки на экземпляры соответствующих классов будут присвоены этим переменным в методе init() апплета.

Инициализация апплета (метод init()).

В этом методе необходимо создать панели и добавить в них компоненты. Сначала установим желтый (Color.yellow) цвет фона окна апплета методом setBacground(). Затем выберем для окна апплета схему размещения GridLayout (сетка 3x1), используя метод setLayout() апплета.

Создание панелей апплета.

Создадим объекты drawPanel, cardPanel и controlPanel класса Panel. Затем установим для панелей менеджеры размещения (вызывая метод setLayout() для объектов cardPanel и controlPanel): для панели cardPanel - менеджер размещения CardLayout, а для панели controlPanel - менеджер FlowLayout.

Создание панелей выбора и ввод их на страницы «блокнота».

Создадим объекты панелей выбора backgrPanel, foregrPanel и fontPanel класса Panel. Затем введем в них соответствующие элементы управления:

```
backgrPanel.add(new Label("Choose Background Color:"));
backgrPanel.add(chBackgr=new Choice());
foregrPanel.add(new Label("Choose Foreground Color:"));
foregrPanel.add(chForegr=new Choice());
fontPanel.add(new Label("Choose Font:"));
fontPanel.add(chFont=new Choice());
```

После этого добавим созданные панели backgrPanel, foregrPanel и fontPanel поочередно в «блокнот» cardPanel, используя метод add(), которому требуется передавать название панели и ссылку на нее, например:

```
cardPanel.add("Background",backgrPanel);
```

Используется именно этот метод введения панелей в «блокнот», потому что иначе метод show() класса CardLayout не сможет отобразить произвольно выбранную панель «блокнота».

Создание кнопок и ввод их в панель управления.

Создадим кнопки btnBackgr, btnForegr, btnFont, btnPrev и btnNext с надписями «Background Color», «Foreground Color», «Set Text Font», «Previous» и «Next». После создания кнопок добавим их по очереди в панель controlPanel.

Ввод панелей в апплет.

После завершения формирования панелей drawPanel, cardPanel и controlPanel добавим их по очереди в апплет при помощи метода add() аплета.

Заполнение меню выбора.

Заполним методом addItem() класса Choice все меню выбора из «блокнота»:

- В меню chBackgr добавим строки "Yellow", "Green", "White".
- В меню chForegr добавим строки "Black", "Red", "Blue".
- В меню chFont добавим строки "Helvetica", "Courier", "TimesRoman".

Выберем шрифт по умолчанию и запомним его имя. Для этого создадим объект класса String, передавая конструктору String() строку "Helvetica", ссылку на этот объект присвоим переменной fontName.

И последнее - отобразим окно апплета вызовом метода show().

Отрисовка содержимого окна (метод paint()).

Изменим метод **paint()** так, чтобы в нем происходил вывод строки «Example of Font, Background and Foreground Colors» в панель drawPanel. Для вывода текста используется выбранный (текущий) шрифт. Кроме того, в этом методе рисуются рамки всех панелей.

Сначала получим контексты отображения gDraw, gCard, gControl панелей drawPanel, cardPanel и controlPanel. Например, для получения контекста панели drawPanel используется следующий код:

```
Graphics gDraw=drawPanel.getGraphics(); // контекст для drawPanel
```

Затем нарисуем рамки вокруг всех панелей апплета, используя их контексты отображения. Например, рамка вокруг панели drawPanel рисуется следующим образом:

```
gDraw.drawRect(0, 0, drawPanel.size().width()-1, drawPanel.size().height()-1);
```

Для панели drawPanel установим текущий выбранный шрифт и выведем строку текста:

```
gDraw.setFont(new Font(fontName,Font.PLAIN,12));
gDraw.drawString("Example of Font, Background and Foreground Colors",10,50);
```

Обработка событий (метод action()).

Переопределим в класса апплета метод action(), вызываемый при некоторых действиях пользователя. В данном случае он будет обрабатывать события, поступающие от кнопок, расположенных в панели controlPanel, и события от меню, расположенных в панели cardPanel. Сначала создадим шаблон этого метода:

```
public boolean action(Event evt, Object obj)
{
        // обработка событий от кнопок
        if(evt.target instanceof Button) // объект является кнопкой?
        {       // получаем ссылку на кнопку, вызвавшую событие
                Button btn=(Button)evt.target;
                // получаем метку кнопки
                String lbl=btn.getLabel();
                // проверка, от какой именно кнопки пришло событие
                // и обработка события от этой кнопки
                // .....
                return true; // если событие обработано, иначе return false
        }
        // обработка событий от меню выбора
        if(evt.target instanceof Choice) // объект является меню выбора?
        {       // получаем ссылку на меню выбора, вызвавшее событие
                Choice ch=(Choice)evt.target;
                // получаем номер выбранного пункта
                int index=ch.getSelectedIndex();
                // проверка, от какого именно меню пришло событие
                // и обработка события от этого меню
                // .....
                return true; // если событие обработано, иначе return false
        }
        // необработанные события передаем на дальнейшую обработку
        return false;
}
```

<u>События от панели controlPanel.</u>

Проверяя, какой именно кнопке равен объект btn, необходимо выполнить следующие действия: при нажатии на кнопки btnPrev, btnNext, btnBackgr, btnForegr, btnFont отобразить соответствующую страницу «блокнота». Проверка равенства объекта btn другому объекту осуществляется следующим образом:

```
if(btn.equals(btnPrev)) // от кнопки "Previous"?
{       // обработка события от кнопки "Previous"
        // .....
```

```
}
```

Если нажата кнопка btnPrev, то сначала с помощью метода getLayout() получаем ссылку на используемый для панели cardPanel менеджер компоновки, а затем вызываем метод previous() для отображения предыдущей страницы панели cardPanel:

```
CardLayout cl=(CardLayout)cardPanel.getLayout();
cl.previous(cardPanel);
```

Аналогичные действия сделаем, если нажата кнопка btnNext, но для отображения следующей страницы вызовем метод next() класса CardLayout.

При нажатии на кнопки btnBackgr, btnForegr, btnFont для отображения соответствующей страницы «блокнота» используется метод show() класса CardLayout. Например, если нажата кнопка btnBackgr:

```
CardLayout cl=(CardLayout)cardPanel.getLayout();
cl.show(cardPanel,"Background");
```

<u>События от панели cardPanel.</u>

Проверяя, какому именно меню равен объект ch, необходимо выполнить следующие действия: изменить цвет фона или цвет переднего плана панели drawPanel или изменить шрифт для текста. Проверка равенства объекта ch другому объекту осуществляется следующим образом:

```
if(ch.equals(chBackgr)) // от меню выбора цвета фона?
{       // обработка события от меню выбора цвета фона
        // .....
}
```

Если событие выбора пункта пришло от меню chBackgr, то, в зависимости от номера выбранного пункта, установим новый цвет фона панели drawPanel при помощи метода setBackground(), перерисуем панель и окно апплета:

```
if(index==0) drawPanel.setBackground(Color.yellow);
else    if(index==1) drawPanel.setBackground(Color.green);
        else    if(index==2) drawPanel.setBackground(Color.white);
drawPanel.repaint(); repaint();
```

Если событие выбора пункта пришло от меню chForegr, то, в зависимости от номера выбранного пункта, установим новый цвет переднего плана панели drawPanel при помощи метода setForeground(), перерисуем панель и окно апплета.

И, наконец, если событие выбора пункта пришло от меню chFont, то, в зависимости от номера выбранного пункта, запишем в строку fontName имя выбранного шрифта, перерисуем панель и окно апплета.

6. Приложение WindowsDemo

Задание. Создать апплет WindowsDemo, в окне которого расположены две кнопки с названиями «Show Frame Window» и «Hide Frame Window». Они предназначены для отображения и временного сокрытия окна класса FrameWnd (новый подкласс класса Frame), в котором выводится строка «This window - object of FrameWnd».В окне класса FrameWnd создается главное меню, содержащее меню «File» (пункты «New» и «Exit») и «Help» (пункты «Content» и «About»), а также в окно добавляется кнопка «Ok». При нажатии на кнопку «Ok» фрейма и при закрытии окна фрейма пользователем происходит его временное сокрытие, а при выборе пункта «Exit» меню фрейма завершается работа всего приложения. При выборе пунктов «New», «Content» и «About» меню окна класса FrameWnd появляется окно диалоговой панели класса DialogWnd (новый подкласс класса Dialog), в которой отображается название выбранного пункта и кнопка «Ok», нажатие на которую вызывает закрытие диалогового окна.

Методические указания. Апплет должен быть создан на основе шаблонов, содержащихся в Приложении 1 (или при помощи системы Java Applet Wizard).

Создание подкласса класса Frame.

Создадим новый подкласс FrameWnd класса Frame (в java-файле апплета), переопределив необходимые методы суперкласса:

```
class FrameWnd extends Frame // объявление нового класса фрейма
{  public FrameWnd(String sTitle) // sTitle - заголовок
   {        super(sTitle); // для правильного создания окна фрейма
            resize(320,240); // задание необходимых размеров окна
            // здесь можно определить различные параметры фрейма,
            // например, форму курсора, меню, добавить компоненты и др.
            // .....
   }
   public boolean handleEvent(Event evt)
   {        switch (evt.id)
            {        case Event.WINDOW_DESTROY:
                              hide(); // временное сокрытие окна фрейма
                              return true;
                     case Event.ACTION_EVENT:
                              // обработка событий от кнопки
                              if(evt.target instanceof Button)
                              {        // получаем ссылку на кнопку
```

```
                        Button btn=(Button)evt.target;
                        // проверка, от какой именно кнопки пришло
                        // событие и обработка события
                        // .....
                        return true;
                    }
                    // обработка событий от меню
                    if(evt.target instanceof MenuItem)
                    {       // получаем название/метку элемента меню
                        String label=(String)evt.arg;
                        // проверка, от какого именно пункта пришло
                        // событие и обработка события
                        // .....
                        return true;
                    }
                    break;
                default: return super.handleEvent(evt);
            }
            return true;
    }
    public void paint(Graphics g)
    {       // вывод в окно фрейма
            // .....
            super.paint(g); // вызов метода родительского класса
    }
}
```

Этот класс предназначен для создания автономного перекрывающегося окна, которое существует независимо от окна навигатора. В дальнейшем в этот класс будут добавлены ряд элементов и в нем будут модифицированы некоторые методы.

Создание подкласса класса Dialog.

Создадим новый подкласс DialogWnd класса Dialog (в java-файле апплета), переопределив необходимые методы суперкласса:

```
class DialogWnd extends Dialog // объявление нового класса диалога
{  // sMsg - сообщение, sTitle - заголовок окна,
   // parent - родительский фрейм, modal - модальность
```

```
public DialogWnd(String sMsg,Frame parent,String sTitle,boolean modal)
{       super(parent,sTitle,modal); // для правильного создания диалога
        resize(100,100); // задание необходимых размеров окна диалога
        // здесь можно создать все необходимые компоненты
        // для размещения внутри диалоговой панели
        // .....
}
public boolean handleEvent(Event evt)
{       switch (evt.id)
        {       case Event.WINDOW_DESTROY:
                        dispose(); // удаление окна диалога
                        return true;
                case Event.ACTION_EVENT:
                        // обработка событий от кнопки
                        if(evt.target instanceof Button)
                        {       // получаем ссылку на кнопку
                                Button btn=(Button)evt.target;
                                // проверка, от какой именно кнопки пришло
                                // событие и обработка события
                                // .....
                                return true;
                        }
                        break;
                default: return super.handleEvent(evt);
        }
        return true;
}
}
```

Этот класс служит для создания диалоговых окон, в которых будет выводиться сообщение. В дальнейшем в этот класс будут добавлены ряд элементов и в нем будут модифицированы некоторые методы.

Класс WindowsDemo. Объявление элементов класса.

В классе апплета объявим следующие элементы - ссылки на объекты классов:

```
FrameWnd frmWnd; // окно фрейма
Button btnShow; // кнопка для отображения фрейма
```

Button btnHide; // кнопка для сокрытия фрейма

Ссылки на экземпляры соответствующих классов будут присвоены этим переменным в методе init() апплета.

Класс WindowsDemo. Инициализация (метод init()).

В этом методе создаются кнопки и добавляются в апплет, а также создается фрейм, который будет отображаться и скрываться при нажатии на эти кнопки.

Сначала создадим кнопки btnShow и btnHide с надписями «Show Frame Window» и «Hide Frame Window» соответственно, а затем добавим их в апплет методом add().

Далее создадим объект нового класса FrameWnd - фрейм с заголовком «Frame from Applet» и ссылку на него присвоим переменной frmWnd.

Класс WindowsDemo. Завершение работы (метод destroy()).

При завершении работы апплета удаляем созданное окно фрейма и освобождаем связанные с ним ресурсы:

frmWnd.dispose();

Класс WindowsDemo. Отрисовка содержимого окна (метод paint()).

Так как апплет не выводит в окно никаких графических изображений, то в этом методе не делается никаких действий.

Класс WindowsDemo. Обработка событий (метод action()).

Переопределим в класса апплета метод action(), вызываемый при некоторых действиях пользователя. В данном случае он будет обрабатывать события, поступающие от кнопок. Сначала создадим шаблон этого метода:

```
public boolean action(Event evt, Object obj)
{
        // обработка событий от кнопок
        if(evt.target instanceof Button) // объект является кнопкой?
        {       // получаем ссылку на кнопку, вызвавшую событие
                Button btn=(Button)evt.target;
                // проверка, от какой именно кнопки пришло событие
                // и обработка события от этой кнопки
                // .....
                return true; // если событие обработано, иначе return false
        }
        // необработанные события передаем на дальнейшую обработку
        return false;
}
```

Проверяя, какой именно кнопке равен объект btn, необходимо выполнить следующие действия: при нажатии на кнопку btnShow отобразить фрейм frmWnd методом show(), а при нажатии на кнопку btnHide скрыть его методом hide(). Проверка равенства объекта btn другому объекту осуществляется следующим образом:

```
if(btn.equals(btnShow)) // от кнопки btnShow?
{       // обработка события от кнопки btnShow
        // .....................................
}
```

Класс FrameWnd. Объявление элементов класса.

В классе фрейма объявим следующие элементы - ссылки на объекты классов:

```
Button btnOk; // кнопка сокрытия фрейма
MenuBar mainMenu; // полоса меню
Menu fileMenu; // меню "File"
Menu helpMenu; // меню "Help"
```

Ссылки на экземпляры соответствующих классов будут присвоены этим переменным в конструкторе фрейма.

Класс FrameWnd. Создание объекта (конструктор FrameWnd()).

Добавим в конструктор класса следующие действия. Сначала установим методом setBackground() цвет Color.yellow в качестве фона окна. Цвет переднего плана установим методом setForeground(), передавая ему константу Color.black.

Затем установим для окна фрейма режим размещения компонент FlowLayout при помощи метода setLayout(). Создадим кнопку btnOk с надписью «Ok» и введем ее во фрейм методом add().

Для того, чтобы сформировать меню фрейма, сначала создадим панель меню mainMenu - объект класса MenuBar, затем создадим выпадающие меню fileMenu (строка "File") и helpMenu (строка "Help") - объекты класса Menu, которые потом добавим в mainMenu его методом add(). Подготовленную панель меню mainMenu установим в качестве главного меню фрейма методом фрейма setMenuBar().

В выпадающие меню fileMenu и helpMenu добавим элементы "New", "Exit" и "Content", About" - объекты класса MenuItem.

Класс FrameWnd. Отрисовка содержимого окна (метод paint()).

В этом методе просто выведем в окно фрейма строку «This window - object of FrameWnd» методом drawString() контекста отображения.

Класс FrameWnd. Обработка событий (метод handleEvent()).

Добавим в метод *handleEvent()* обработку событий, поступающих от кнопки btnOk и от меню фрейма.

Если объект, вызвавший событие, является кнопкой (результат операции evt.target instanceof Button равен true) и эта кнопка является кнопкой btnOk (метод btn.equals(btnOk) возвращает true), то при помощи метода hide() временно скроем окно фрейма.

Если объект, вызвавший событие, является элементом меню (результат операции evt.target instanceof MenuItem равен true), то:

- если выбран элемент меню "Exit" (объект label равен строке "Exit"), то методом System.exit(0) прекратим работу всего апплета Проверка равенства объекта label другому объекту осуществляется следующим образом:

```
if(label.equals("Exit")) // от элемента "Exit"?
{           // обработка события
            // .....
}
```

- если же выбраны элементы "New", "Content" или About", то в каждом случае создадим модальную диалоговую панель класса DialogWnd, родителем которой является текущий экземпляр фрейма, в которой выводится название выбранного элемента меню, и отобразим ее, например:

```
DialogWnd dlgWnd=new DialogWnd("Item New selected",
                                this,"Dialog from Frame",true);
dlgWnd.show();
```

Класс DialogWnd. Объявление элементов класса.

В классе диалога объявим следующие элементы - ссылки на объекты классов:

```
Label lbMsg; // поле для отображения текста сообщения
Button btnOk; // кнопка удаления панели
```

Ссылки на экземпляры соответствующих классов будут присвоены этим переменным в конструкторе фрейма.

Класс DialogWnd. Создание объекта (конструктор DialogWnd()).

Добавим в конструктор класса следующие действия. Сначала установим для окна диалога режим размещения компонент GridLayout (сетка 2x1) при помощи метода setLayout(). Затем создадим метку lbMsg, передавая конструктору класса Label строку sMsg и режим выравнивания по центру. Созданную метку введем в панель диалога методом add(). Создадим кнопку btnOk с надписью «Ok» и введем ее в диалог методом add().

Класс DialogWnd. Обработка событий (метод handleEvent()).

Добавим в метод *handleEvent()* обработку событий, поступающих от кнопки btnOk. Если объект, вызвавший событие, является кнопкой (результат операции evt.target instanceof Button равен true) и эта кнопка является кнопкой btnOk (метод btn.equals(btnOk) возвращает true), то при помощи метода dispose() удалим окно диалога.

Задания к лабораторной работе

Задание 1. Создать апплеты *PanelsDemo1*, *PanelsDemo2* и *WindowsDemo* и объяснить их работу. Первые два апплета должны иметь возможность работать как независимые приложения.

Задание 2. Дать ответы на контрольные вопросы

Контрольные вопросы

41. Что такое контейнеры? Какие два основных вида контейнеров существует?
42. Какие существуют типы контейнеров?
43. Для чего чаще всего используются панели?
44. Какими двумя способами можно рисовать в окне панели?
45. Как происходит обработка события компонентами контейнера?
46. Какова иерархия обработки события различными компонентами контейнера?
47. В чем основное отличие окон и панелей?
48. Что является обязательным параметром конструктора при создании экземпляра класса окон?
49. Каковы отличительные особенности имеют фреймы?
50. Какие методы должны быть переопределены в новом подклассе фреймов?
51. Какой класс контейнеров автоматически поддерживает работу с меню? Почему он это делает автоматически?
52. Как добавить меню в контейнер?
53. Как создать новое меню и добавить в него элементы?
54. Какие существуют классы элементов меню?
55. Какими методами обрабатываются события меню?
56. Для чего в основном используются окна диалогов?
57. Каковы важные отличия окон диалогов от фреймов?
58. Объект какого класса должен обязательно быть родителем диалогового окна?
59. Как создать диалог своего класса?
60. Для чего предназначены менеджеры компоновки? Какие существуют режимы размещения?

ЛАБОРАТОРНАЯ РАБОТА № 6.
МНОГОПОТОКОВЫЕ ПРИЛОЖЕНИЯ (2 часа)

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ

При создании приложений для операционной системы Microsoft Windows с использованием программного интерфейса Win API операционной системы можно решать многие такие задачи, как анимация или работа в сети, и без использования многозадачности (без создания подзадач главной задачи). Например, для анимации можно обрабатывать сообщения соответствующим образом настроенного таймера.

Приложениям Java такая методика недоступна, так как в этой среде не предусмотрено способов периодического вызова каких-либо процедур. Поэтому для решения многих задач не обойтись без мультизадачности.

Прежде чем начать рассматривать процесс реализации мультизадачности, уточним некоторые термины.

1. Процессы, задачи и приоритеты

Обычно в мультизадачной операционной системе выделяют такие объекты, как процессы и задачи. Между этими понятиями существует большая разница, которую следует четко представлять.

Процесс (задача)

Процесс (process) - это объект, который создается операционной системой, когда пользователь запускает приложение. Процессу выделяется отдельное адресное пространство, причем это пространство физически недоступно для других процессов. Процесс может работать с файлами или с каналами связи локальной или глобальной сети.

Поток (нить)

Для каждого процесса операционная система создает одну 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 операционной системы.

Все потоки, созданные процессом, выполняются в адресном пространстве этого процесса и имеют доступ к ресурсам процесса. Однако поток одного процесса не имеет никакого доступа к ресурсам потока другого процесса, так как они работают в разных адресных пространствах. При необходимости организации взаимодействия между процессами или потоками, принадлежащим разным процессам, следует пользоваться системными средствами, специально предназначенными для этого.

Приоритеты потоков

Если процесс создал несколько потоков, то все они выполняются параллельно, причем время центрального процессора (или нескольких центральных процессоров в мульпроцессорных 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.

Следует особо отметить, что 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.

2. Реализация многозадачности в Java

Для создания мультизадачных 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. При этом необходимо переопределить метод run(). Новая реализация этого метода будет работать в рамках отдельного потока.

Во-вторых, создаваемый класс, не являясь подклассом класса Thread, может реализовать интерфейс Runnable. При этом в рамка этого класса необходимо определить метод run(), который будет работать как отдельный поток.

Рассмотрим подробнее обе возможности реализации многозадачности.

2.1 Создание подкласса Thread

Первый способ реализации мультизадачности основан на наследовании от класса Thread. При использовании этого способа для потоков определяется отдельный класс, например:

```
class myThread extends Thread
{       // этот метод получает управление при запуске
        // потока методом start() класса Thread
        public void run()
        {       // здесь можно добавить код, который будет
                // выполняться в рамках отдельного потока
        }
        // здесь можно добавить специализированный для класса код
}
```

Следует обратить внимание на метод run(). Этот метод 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() не вызывается напрямую никакими другими методами. Он 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 выполняется методом stop() класса Thread. Обычно остановка всех работающих потоков, созданных апплетом, выполняется в методе stop() апплета.

Рассмотрим процедуру создания и запуска потоков на примере использования их в апплетах (в обычных приложениях потоки используются аналогично)- *пример 1*. Определим класс апплетов MultiTask (можно использовать шаблоны, содержащиеся в Приложении 1):

```
/*---------- Пример 1. Файл MultiTask.java ----------*/
import java.applet.*;
import java.awt.*;
public class MultiTask extends Applet
{  public MultiTask()
   {        // здесь можно добавить код конструктора
   }
   public String getAppletInfo()
   {        return "Name: Applet\r\n"; // информация об апплете
   }
   public void init()
   {        resize(320, 240); // установка размера апплета
            // здесь можно добавить код инициализации апплета
   }
   public void destroy()
   {        // здесь можно добавить код завершения работы апплета
   }
   public void paint(Graphics g)
   {        // здесь можно добавить код вывода в окно апплета
   }
   public void start()
   {        // здесь можно добавить код, работающий при запуске апплета
   }
   public void stop()
```

```
    {       // здесь можно добавить код, работающий при остановке
    }
    // здесь можно добавить специализированный для класса код
}
```

Перед использованием других потоков необходимо определить их классы, например:

```
    // поток, рисующий прямоугольники в окне апплета
    class DrawRects extends Thread
    {       // конструктор, получающий ссылку на создателя объекта - апплет
            public DrawRects(MultiTask parentObj)
            {       parent=parentObj;
            }
            public void run()
            {       Graphics gr=parent.getGraphics(); // контекст апплета
                    while(true) // в цикле выводятся прямоугольники
                    {       int w=parent.size().width-1, h=parent.size().height-1;
                            gr.setColor(new Color((float)Math.random(),
                                    (float)Math.random(),(float)Math.random()));
                            gr.fillRect((int)(Math.random()*w),(int)(Math.random()*h),
                                    (int)(Math.random()*w),(int)(Math.random()*h));
                    }
            }
            MultiTask parent; // ссылка на создателя объекта
    }
    // поток, рисующий эллипсы в окне апплета
    class DrawOvals extends Thread
    {       // конструктор, получающий ссылку на создателя объекта - апплет
            public DrawOvals(MultiTask parentObj)
            {       parent=parentObj;
            }
            public void run()
            {       Graphics gr=parent.getGraphics(); // контекст апплета
                    while(true) // в цикле выводятся эллипсы
                    {       int w=parent.size().width-1, h=parent.size().height-1;
                            gr.setColor(new Color((float)Math.random(),
                                    (float)Math.random(),(float)Math.random()));
```

```
                gr.fillOval((int)(Math.random()*w),(int)(Math.random()*h),
                                (int)(Math.random()*w),(int)(Math.random()*h));
                }
        }
        MultiTask parent; // ссылка на создателя объекта
}
```

Теперь рассмотрим процесс использования созданных классов. Вначале апплет должен объявить в качестве своих элементов ссылки на объекты классов потоков:

```
DrawRects Rects =null;
DrawOvals Ovals =null;
```

Далее в методе start() апплета создаются объекты потоков и потоки запускаются на выполнение:

```
public void start()
{       if(Rects==null) { Rects=new DrawRects(this); Rects.start(); }
        if(Ovals==null) { Ovals=new DrawOvals(this); Ovals.start(); }
}
```

Когда пользователь покидает страницу апплета, то вызывается метод stop() апплета. Именно в этом методе целесообразно останавливать работающие потоки:

```
public void stop()
{       if(Rects!=null) Rects.stop();   if(Ovals!=null) Ovals.stop();
}
```

2.2 Реализация интерфейса Runnable

Если нет необходимости расширять класс Thread подобно примерам, показанным выше, то можно применить второй способ реализации многозадачности. Допустим, уже существует класс MyClass, функциональные возможности которого удовлетворяют разработчика:

```
class myClass
{       // код класса - объявление его элементов и методов
        .....
        }
```

и теперь необходимо, чтобы он выполнялся как отдельный поток. Для этого необходимо для этого класса реализовать интерфейс Runnable, создавая разделяемый метод run() интерфейса Runnable:

```
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
        }
}
```

Рассмотрим пример и продолжим создание апплета MultiTask. Пусть существует класс, предназначенный для вывода случайной линии в окне апплета MultiTask:

```
class DrawLines
{       // конструктор, получающий ссылку на создателя объекта
        public DrawLines(MultiTask parentObj)
        {       parent=parentObj;
        }
        public void draw()
        {       int w=parent.size().width-1, h=parent.size().height-1;
                Graphics gr=parent.getGraphics();
                gr.setColor(new Color((float)Math.random(),
                        (float)Math.random(),(float)Math.random()));
                gr.drawLine((int)(Math.random()*w),(int)(Math.random()*h),
                        (int)(Math.random()*w),(int)(Math.random()*h));
        }
        MultiTask parent; // ссылка на создателя объекта
}
```

Реализуем для этого класса интерфейс Runnable и создадим метод run(), выполняющийся в рамках отдельного потока:

```
class DrawLines implements Runnable
{       // конструктор, получающий ссылку на создателя объекта
        public DrawLines(MultiTask parentObj)
        {       parent=parentObj;
        }
        public void draw()
        {       int w=parent.size().width-1, h=parent.size().height-1;
                Graphics gr=parent.getGraphics();
                gr.setColor(new Color((float)Math.random(),
```

```
                    (float)Math.random(),(float)Math.random()));
            gr.drawLine((int)(Math.random()*w),(int)(Math.random()*h),
                    (int)(Math.random()*w),(int)(Math.random()*h));
        }
        MultiTask parent; // ссылка на создателя объекта
        // этот метод получает управление при запуске потока
        public void run()
        {       while(true) { draw(); }
        }
}
```

В классе апплета MultiTask объявляется элемент - объект класса Thread:

```
Thread Lines=null;
```

Затем в методе start() апплета создается поток Lines, при этом конструктору Thread() передается экземпляр класса DrawLines, реализующего интерфейс Runnable (создание потока с исполняемым адресатом). Далее поток Lines запускается на выполнение:

```
public void start()
{       if(Rects==null) { Rects=new DrawRects(this); Rects.start(); }
        if(Ovals==null) { Ovals=new DrawOvals(this); Ovals.start(); }
        if(Lines==null) { Lines=new Thread(new DrawLines(this)); Lines.start(); }
}
```

Все пооки апплета останавливаются в его методе stop():

```
public void stop()
{       if(Rects!=null) Rects.stop();   if(Ovals!=null) Ovals.stop();
        if(Lines!=null) Lines.stop();
}
```

2.3 Применение мультизадачности для анимации

Одним из наиболее распространенных применений апплетов - это 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.

Работа апплетов основана на обработке событий. Основной (первичный) класс апплета обрабатывает события, переопределяя те или иные методы суперкласса Applet.

Проблема с периодическим обновлением окна апплета возникает из-за того, что в языке Java не предусмотрено никакого механизма для создания генератора событий, способного вызывать какой-то метод класса с заданным интервалом времени.

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.

Возникает вопрос, а можно ли воспользоваться методом paint() для периодической перерисовки окна апплета, организовав в нем, например, бесконечный цикл с задержкой. К сожалению, так поступать нельзя. Метод paint() после перерисовки апплета должен сразу возвратить управление, иначе работа апплета будет заблокирована.

Единственный выход из создавшейся ситуации - 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 (в апплете MultiTask использовался именно этот способ).

Если необходимо создать только одну задачу, работающую одновременно с кодом апплетом, то проще выбрать способ реализации многозадачности с использованием интерфейса Runnable для подкласса класса Applet.

Идея заключается в том, что основной класс апплета DoubleTask (*пример 2*), который является дочерним по отношению к классу Applet, дополнительно реализует интерфейс Runnable, переопределяя метод run() этого интерфейса:

```
/*---------- Пример 2. Файл DoubleTask.java ----------*/
import java.applet.*;
import java.awt.*;
public class DoubleTask extends Applet implements Runnable
{  public DoubleTask()
   {          // здесь можно добавить код конструктора
   }
   public String getAppletInfo()
   {          return "Name: Applet\r\n"; // информация об апплете
   }
   public void init()
   {          resize(320, 240); // установка размера апплета
              // здесь можно добавить код инициализации апплета
   }
```

```
    public void destroy()
    {       // здесь можно добавить код завершения работы апплета
    }
    public void paint(Graphics g)
    {       // здесь можно добавить код вывода в окно апплета
    }
    public void start()
    {       // создание и запуск потока
            if(task==null) { task=new Thread(this); task.start(); }
    }
    public void stop()
    {       // остановка потока
            if(task!=null) task.stop();
    }
    public void run()
    {       // здесь можно добавить код, который будет
            // выполняться в рамках отдельного потока
    }
    Thread task=null; // ссылка на поток
    // здесь можно добавить специализированный для класса код
}
```

Внутри класса необходимо определить метод run(), который будет выполняться в рамках отдельного потока. При этом можно считать, что код апплета и код метода run() работают одновременно как разные задачи.

Для создание потока используется оператор new. Поток создается как объект класса Thread, причем конструктору передается ссылка на класс апплета:

```
task=new Thread(this);
```

Запуск потока осуществляется методом start для объекта task. При запуске потока начинает работать метод run(), определенный в классе апплета. Причем одновременно с этим код методов апплета также выполняется, так что апплет может обрабатывать различные события в методах обработки событий.

Определим, например, метод paint() апплета следующим образом:

```
public void paint(Graphics g)
{       g.drawString("Random value "+Math.random(),10,20);
}
```

А в методе run() реализуем бесконечный цикл, в котором периодически с задержкой 50 миллисекунд вызывается метод repaint(), вызывающий перерисовку апплета:

```
public void run()
{       while(true)
        {
                repaint();
                try{ Thread.sleep(50); } catch(InterruptedException e) { stop(); }
        }
}
```

Для выполнения задержки метод run() вызывает sleep() класса Thread. Этот метод может вызвать исключение InterruptedException, которое можно обработать с помощью операторов try и catch. В данном случае при возникновении исключения апплет прекращает свою работу.

Такая перерисовка может вызвать мигание окна апплета, поэтому удачной ее назвать нельзя, но этот апплет можно использовать просто как пример реализации «двухзадачности» для апплета.

2.4 Апплет двойного назначения, реализующий интерфейс Runnable

Рассмотрим шаблон апплета двойного назначения, реализующего интерфейс Runnable, создаваемый системой автоматизированной разработки шаблонов апплета Java Applet Wizard, встроенной в Microsoft Visual J++. Данный пример во многом похож на пример 2, но новом апплете не только демонстрируется периодическое обновление окна с использованием многозадачности (одновременно с работой апплета выполняется еще одна задача), но и поддерживается возможность работы программы как независимого приложения.

Для создания шаблона апплета Multi (*пример 3)* среде разработки Microsoft Visual J++ выбрать пункт «New» меню «File». В появившейся диалоговой панели выбрать закладку «Projects» и отметить тип «Java Applet Wizard». В поле Name следует ввести имя апплета (Multi). После нажатия на кнопку «Create» на экране появятся по очереди несколько диалоговых панелей, в которой нужно описать создаваемый шаблон апплета.

В первой диалоговой панели в поле «How would you like to be able to run your program» следует включить переключатель «As an applet and as an application». При этом создаваемая программа сможет работать как под управлением WWW-навигатора, так и как самостоятельное приложение.

Система Java Applet Wizard может создать образец документа HTML, в который будет включен разрабатываемый апплет. Для этого во второй диалоговой панели необходимо включить переключатель «Yes, please» в поле «Would you like a sample HTML file».

В третьей диалоговой панели можно указать, будет ли апплет создавать потоки (мультизадачный режим). Так апплет Multi будет создаваться как многозадачный, то в поле «Would you like your applet to be multi-threaded» необходимо включить переключатель «Yes, please».

На вопрос «Would you like support for animation» можно утвердительно ответить только в том случае, если апплет мультизадачный. Но для создаваемого приложения Multi анимация не предполагается поддерживаться, поэтому следует установить переключатель «No, thank you».

Три переключателя, расположенные в поле «Which mouse event handlers would you like added», позволяют автоматически добавить обработчики сообщений от мыши. Для данного приложения следует оставить переключатели невключенными..

С помощью четвертой панели можно указать, какие параметры должны передаваться апплету через документ HTML при запуске. Для приложения Multi никаких параметров передавать не нужно.

В пятой диалоговой панели дается возможность отредактировать информацию о создаваемом апплете. Эта информация будет выдаваться методом getAppInfo(), определенном в классе апплета.

После завершения заполнения последней панели выдается информационное окно, в котором перечисляются заданные ранее параметры приложения. Если нажать кнопку «Ok», то требуемый проект будет создан.

В результате работы системы Java Applet Wizard будет создано три файла с исходными текстами: текст апплета (файл Multi.java и файл MultiFrame.java) и HTML-документ (файл Multi.html).

Рассмотрим тексты создаваемых файлов (комментарии, создаваемые Java Applet Wizard, переведены в данном примере на русский язык). Тексты подобных шаблонных файлов содержатся в Приложении 6 и их можно в дальнейшем использовать в качестве шаблонов апплета двойного назначения, реализующего интерфейс Runnable.

Листинг Java-файла класса апплета:

```
//************************************************************
// Multi.java:          Applet
//
//************************************************************
import java.applet.*;
import java.awt.*;
import MultiFrame;
```

```
public class Multi extends Applet implements Runnable
{
   // Поток, который будет работать одновременно с апплетом
   // ------------------------------------------------------------
   private Thread m_Multi = null;
   // Признак режима работы программы:
   // true/false - приложение/апплет
   //----------------------------------------------------------------------
   private boolean m_fStandAlone = false;
   //----------------------------------------------------------------------
   public static void main(String args[])
   {          // Создать рамку (фрейм) для апплета
              MultiFrame frame = new MultiFrame("Title");
              // До изменения размеров фрейма отобразить его.
              // Это необходимо для того, чтобы метод insert()
              // выдавал правильные значения
              frame.show(); frame.hide();
              frame.resize(frame.insets().left + frame.insets().right + 320,
                        frame.insets().top + frame.insets().bottom + 240);
              // Создание объекта апплета, связывание апплета и фрейма
              Multi applet_Combi = new Multi();
              frame.add("Center", applet_Combi);
              // Установление признака режима работы - приложение
              applet_Combi.m_fStandAlone = true;
              // Вызов методов ааплета для его запуска
              applet_Combi.init();
              applet_Combi.start();
              // Отображение окна фрейма
              frame.show();
   }
   //----------------------------------------------------------------------
   public Multi()
   {          // Сделать: Добавьте сюда код конструктора
   }
   //----------------------------------------------------------------------
```

```
public String getAppletInfo()
{       return "Name: CombiApplet\r\n" +
                "";
}
//---------------------------------------------------------------------------
public void init()
{       resize(320, 240);
        // Сделать: Добавьте сюда код инициализации
}
//---------------------------------------------------------------------------
public void destroy()
{       // Сделать: Добавьте сюда код завершения работы апплета
}
//---------------------------------------------------------------------------
public void paint(Graphics g)
{       // Сделать: Добавьте сюда код перерисовки окна апплета
        g.drawString("Running: " + Math.random(), 10, 20);
}
//---------------------------------------------------------------------------
public void start()
{       // если поток еще не создан, апплет создает
        // новый поток как объект класса Thread
        if (m_Multi == null)
        {       m_Multi = new Thread(this); // создание потока
                m_Multi.start(); // запуск потока
        }
        // Сделать: Добавьте сюда код, который должен
        // работать при запуске апплета
}
//---------------------------------------------------------------------------
public void stop()
{       // когда пользователь покидает страницу,
        // метод stop() класса Thread останавливает поток
        if (m_MultiTask != null) // если поток был создан
        {       m_Multi.stop(); //остановка потока
```

```
                m_Multi = null; // сброс ссылки на поток
            }
            // Сделать: Добавьте сюда код, который должен
            // работать при остановке апплета
    }
    // Метод, который работает в рамках отдельного потока.
    // Он вызывает периодическое обновление окна апплета
    //---------------------------------------------------------------------
    public void run()
    {       // выполняем обновление окна в бесконечном цикле
            while (true)
            {       try
                    {       // вызов функции обновления
                            repaint();
                            // Сделать: Добавьте сюда код, который должен
                            // здесь работать в рамках отдельного потока
                            //выполнение небольшой задержки
                            Thread.sleep(50);
                    }
                    catch (InterruptedException e)
                    {       // Сделать: Добавьте сюда код, который должен
                            // здесь работать при генерации исключения
                            // если при выполнении задержки произошло
                            // исключение, останавливаем работу апплета
                            stop();
                    }
            }
    }
    // Сделать: Добавьте сюда код, необходимый для работы
    // создаваемого специализированного апплета
```

Листинг Java-файла класса фрейма для апплета:

```
import java.awt.*;
//==========================================
// Этот класс действует как окно, в котором отображается апплет,
```

```
// когда он запускается как обычное приложение
//=============================================
class MultiFrame extends Frame
{
   // Конструктор класса
   //-------------------------------------------------------------------------
   public MultiFrame(String str)
   {        super (str);
            // Сделать: Добавьте сюда код конструктора
   }
   //-------------------------------------------------------------------------
   public boolean handleEvent(Event evt)
   {        switch (evt.id)
            {        // при закрытии окна завершается работа приложения
                     //-----------------------------------------------------------------
                     case Event.WINDOW_DESTROY:
                              // Сделать: Добавьте сюда код, который должен
                              // работать при остановке приложения
                              dispose(); // удаление окна
                              System.exit(0); // завершение приложение
                              return true;
                     default:
                              // передача сообщения на обработку
                              // методу базового класса
                              return super.handleEvent(evt);
            }
   }
}
```

Листинг HTML-файла:

```
<html>
   <head>
            <title> Multi </title>
   </head>
   <body>
```

```
        <hr>
        <applet
                code= Multi.class
                name= Multi
                width=320
                height=240 >
        </applet>
        <hr>
    </body>
</html>
```

3. Потоки (нити)

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
- isAlive - Определяет, является ли поток активным (запущенным и не остановленным)
- suspend - Приостанавливает выполнение потока
- resume - Возобновляет выполнение потока. 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

Для того, чтобы эффективно использовать потоки, необходимо уяснить их различные аспекты, а также особенности работы исполняющей системы языка Java. Рассмотрим атрибуты потоков.

3.1 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 его или остановить:

```
myThread.start();
.....
myThread.stop();
```

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 приоритета

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.

- *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». Если в методе run() содержится бесконечный цикл, то завершить работу потока можно при помощи вызова его метода stop().

Если поток используется в апплете, то при завершении работы апплета поток автоматически получает метод stop(). Однако практическое правило таково: метод stop() следует посылать всем потокам всякий раз в методе stop() апплета самостоятельно, не позволяя потокам выполняться в фоновом режиме, когда сам апплет фактически уже не работает.

3.2 Исключительные ситуации для потоков

Исполняющая система языка Java будет возбуждать исключительную ситуацию IllegalThreadStateException всякий раз, когда будет вызываться метод, которым поток не может оперировать в своем текущем состоянии.

Например, ожидающий поток не может работать с методом resume(). В этом случае поток занят ожиданием, он просто не знает, как реагировать на этот метод.

То же самое справедливо и в том случае, когда происходит попытка вызвать метод suspend() для потока, который не находится в состоянии «Выполняемый поток». Если он уже был приостановлен, просто ожидает, ожидает условия или заблокирован операцией ввода-вывода, поток не понимает как работать с этим методом.

Всякий раз при вызове метода потока, который потенциально может привести к возникновению исключительной ситуации, необходимо обеспечить и способ обработки исключительных ситуаций для того, чтобы перехватывать любые возбуждаемые ситуации, например:

```
try
{       // здесь вызываются методы для потоков
        .....
}
catch(InterruptedException e)
{       // потоку был послан метод, которым он
        // не может оперировать в данном состоянии,
        // можно остановить поток его методом stop()
}
```

3.3 Приоритеты потоков

В языке Java каждый поток обладает приоритетом, который оказывает влияние на порядок его выполнения. Потоки с высоким приоритетом выполняются до потоков с низким приоритетом. Это существенно, поскольку возникают моменты, когда их необходимо разделить подобным образом

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.

Порядок выполнения потоков и количество времени, которое они получат от процессора, - главные вопросы для разработчиков. Каждый поток должен разделять процессорное время с другими, не монополизируя систему.

Система, которая имеет дело со множеством выполняющихся потоков, может быть или приоритетная, или неприоритетная. Приоритетные системы гарантируют, что в любое время будет выполняться поток с самым высоким приоритетом. Виртуальная машина Java является приоритетной, то есть выполняться всегда будет поток с самым высоким приоритетом.

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 выполнятся «Выполняемый поток» с наивысшим приоритетом.

Как выполняются потоки одного и того же приоритета, в спецификации Java не описано. Казалось бы, потоки должны использовать процессор совместно, но это не всегда так. Порядок их выполнения определен основной операционной системой и аппаратными средствами. Операционная система обслуживает потоки при помощи планировщика, который и определяет порядок выполнения.

3.4 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);
Thread t3=new Thread(genericGroup,this);
```

Совсем не обязательно создавать группу, в состав которой входят потоки программы. Можно воспользоваться группой, созданной исполняющей системой Java, или группой, созданной приложением, в котором выполняется апплет.

Если при создании нового потока не указать, к какой конкретной группе он принадлежит, этот поток войдет в состав группы потоков main (главная группа) языка Java. Иногда ее еще называют текущей группой потоков. В случае апплета main может и не быть главной группой. Право присвоения имени принадлежит WWW-броузеру. 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.

Группы потоков поддерживают понятие привилегия доступа. Если не указать тип привилегии доступа для группы, то потоки этой группы смогут осуществлять запросы и поиск информации о потоках в других группах.

По умолчанию создаваемым потокам не присваивается определенный уровень защиты. В результате любой поток из любой группы может свободно контролировать и изменять потоки в других группах. Однако можно использовать абстрактный класс SecurityManager для указания ограничений доступа к определенным группам потоков. Для этого необходимо создать подкласс класса SecurityManager и заменить те методы, которые используются для защиты потоков. В основном эта процедура применима для потоков приложений, так как некоторые WWW-броузеры не позволяют заменить уровни защиты.

4. Приложение VertScroller

Задание. Создать апплет двойного назначения VertScroller, в окне которого отображаются строки текста, медленно всплывающие вверх. Для выполнения плавного сдвига создать поток, который периодически рисует новые строки в нижней части окна, сдвигая перед этим старые строки вверх. Основной класс апплета должен реализовывать интерфейс Runnable, поэтому для реализации многозадачности не нужно создавать подкласс класса Thread.

Методические указания. Апплет должен быть создан на основе шаблонов, содержащихся в Приложении 6 (или при помощи системы Java Applet Wizard)**.**

Объявление элементов класса апплета.

В классе апплета объявим следующие элементы:

```
// выводимые строки
String strs[]={"First string", "Second string", "Third string"};
// текущий размер окна апплета
Dimension appSize;
```

Перерисовка окна апплета (метод paint()).

Для перерисовки изображения в окне используется метод paint(), которому передается объект g типа Graphics (g - контекст отображения для окна). Методами этого класса пользуются для вывода графической информации в окно апплета.

В данном апплете этот метод используется для определения текущего размера appSize окна и для закраски фона окна и рисовании рамки окна. Сначала раскрасим фон окна и нарисуем рамку окна. Для этого сделаем следующие действия: при помощи метода size() класса апплета получим текущий размер окна и присвоим ссылку на него объекту appSize; методом

g.setColor() выберем цвет фона Color.yellow; зарисуем всю внутреннюю область окна с помощью метода g.fillRect() и объекта appSize; методом g.setColor() выберем цвет рамки Color.black; нарисуем рамку вокруг окна методом g.drawRect(), используя объект appSize.

Метод, работающий в рамках отдельного потока (метод run()).

Поток, выполняющийся в рамках метода run() одновременно с кодом апплета, будет по очереди извлекать строки из массива strs и отображать их в нижней части окна: Сначала определим переменные метода и получим контекст окна апплета и высоту символов текущего шрифта:

```
public void run()
{       int counter=0; // счетчик сдвигов
        int current=0; // номер текущей рисуемой строки
        int hy=1; // размер сдвига по вертикали
        Graphics g=getGraphics(); // контекст окна апплета
        int yChar=g.getFontMetrics().getHeight(); // высота символа
        while (true)
        {       try
                {       // вывод в окно апплета
                        Thread.sleep(50);
                }
                catch (InterruptedException e)
                {       stop();
                }
        }
}
```

В блоке try метода run() выполним свертку окна апплета: 1) с помощью метода g.copyArea() скопируем верхнюю часть экрана, ограниченную прямоугольником с началом в точке (0, hy+1), шириной appSize.width-1 и высотой appSize.height-1, в область с началом в точке (0, -hy); 2) методом g.setColor() выберем цвет фона Color.yellow; 3) нарисуем методом g.fillRect() прямоугольник с началом в точке (1, appSize.height-hy-1), шириной appSize.width-2 и высотой hy.

Далее увеличим счетчик сдвигов counter на 1, а затем проверим его. Если значение counter равно yChar+5 (то есть сдвинута полная строка), то рисуем строку с номером current в нижней части окна. Для этого: 1) сбрасываем счетчик сдвигов counter в 0; 2) если строка первая (current равно 0) то методом g.setColor() устанавливаем красный цвет, иначе - черный; 3) методом g.drawString() выводим строку с номером current в точке с координатами (10,

appSize.height-10); 4) увеличиваем номер текущей рисуемой строки current на 1; 5) если нарисованы все строки (current больше или равно strs.length), то устанавливаем номер текущей рисуемой строки current в 0.

Далее выполняется задержка на 50 миллисекунд, после чего работа бесконечного цикла возобновляется с самого начала.

Задания к лабораторной работе

Задание 1. Проверить и объяснить работу апплетов *MultiTask, DoubleTask* и *Multi,* рассматриваемых в данной главе в качестве примеров и отмеченных курсивом.

Задание 2. Создать апплет двойного назначения *VertScroller* и объяснить его работу.

Задание 3. Дать ответы на контрольные вопросы.

Контрольные вопросы

61. Что такое процесс (задача) и поток (нить)?
62. Чем определяется порядок передачи управления потоками?
63. Какие есть способы реализации многозадачности в Java?
64. Что необходимо сделать для создания подкласса потоков (подкласса Thread)?
65. Когда запускается на выполнение метод run() подкласса Thread?
66. 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 приоритетная система и какой системой является виртуальная машина Java?
13. Что такое группы потоков и чем они полезны?
14. Что такое родовая группа потоков и главная группа потоков?
15. Чем определяется уровень защиты группы потоков?

ЛАБОРАТОРНАЯ РАБОТА № 7

АВТОНОМНЫЕ ПРИЛОЖЕНИЯ. ПОТОКИ ДАННЫХ. РАБОТА С ЛОКАЛЬНЫМИ ФАЙЛАМИ (2 ЧАСА)

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ

Язык Java очень широко используется для создания апплетов, встраиваемых в html-страницы, однако необходимость реализации апплетов на компьютерах пользователей сети WWW накладывает на них некоторые ограничения. Так, например, апплеты не имеют доступа к файлам, расположенным на удаленном компьютере, на котором эти апплеты были запущены, хотя и могут при этом обращаться к файлам, которые находятся в каталоге сервера WWW, откуда эти апплеты были загружены.

Программировать на языке Java можно и за рамками модели апплета, создавая отдельные, независимые приложения. Эти приложения похожи на обычные программы, написанные на других языках программирования, и имеют доступ к локальной файловой системе компьютера, на котором они запущены. Автономные приложения Java могут работать также и с удаленными файлами (через сеть Internet или Intranet).

1. Самостоятельные графические приложения

Рассмотрим, как в самостоятельных приложениях создать графический интерфейс пользователя, работающий без оболочки времени выполнения.

Все, что нужно для создания отдельного приложения Java, - это иметь метод main(), определенный в классе. Когда запускается программа java.exe на обработку этого класса, она вызывает метод main().

Для того, чтобы сделать приложение таким же дружественным и иллюстративным, как апплет, необходимо создать фрейм либо непосредственно в методе main(), либо в одном из методов, им вызываемом. Можно создавать не только объект класса Frame, но и его подклассов, например, как в приложении StandAlone (пример 1). Текст подобного шаблонного файла содержится в Приложении 7 и его можно в дальнейшем использовать в качестве шаблона самостоятельного графического Java-приложения.

```
/*---------- Пример 1. Файл StandAlone.java ----------*/
import java.awt.*;
// Основной класс для приложения StandAlone
public class StandAlone
{  public static void main(String args[])
   {// Создать рамку (фрейм)
            MainWndFrame frame = new MainWndFrame("Title");
            // Отображение окна фрейма
            frame.show();
   }
   // Сделать: Добавьте сюда код, необходимый для работы
```

```
    // создаваемого специализированного приложения
}
// Класс главного фрейма для приложения
class MainWndFrame extends Frame
{   public MainWndFrame(String str) // конструктор
    {       super (str);
            // Сделать: Добавьте сюда код конструктора
            resize(320,240); // задание необходимых размеров окна
            // здесь можно определить различные параметры фрейма,
            // например, форму курсора, меню, добавить компоненты и др.
    }
    public boolean handleEvent(Event evt) // обработчик событий
    {       switch (evt.id)
            {       case Event.WINDOW_DESTROY:
                    // при закрытии окна завершается работа приложения
                    //-----------------------------------------------
                            // Сделать: Добавьте сюда код, который должен
                            // работать при остановке приложения
                            dispose(); // удаление окна
                            System.exit(0); // завершение приложение
                            return true;
                    default:
                            // передача сообщения на обработку
                            // методу базового класса
                            return super.handleEvent(evt);
            }
    }
}
```

В предыдущих главах подробно рассматривался Abstract Windowing Toolkit (AWT). Все элементы управления и контейнеры можно использовать и при создании отдельного графического приложения. Можно также создать подкласс Applet и ввести его во фрейм, тогда отдельное графическое приложения будет использовать все возможности апплетов (включая получение графических и звуковых данных из Internet).

Графические данные из Internet (без создания объекта подкласса Applet) можно получить при помощи класса Toolkit. Класс Toolkit - это абстрактный класс, который служит

связкой между AWT и локальной системой управления окнами. Этот класс определяет метод getImage(), который работает аналогично методу с таким же именем класса Applet. Чтобы из самостоятельного приложения получить изображение из Internet, следует внутри любого подкласса Frame сделать следующие действия:

```
Toolkit T=getToolkit();
Im=T.getImage(U); // U - URL-адрес, объект класса URL
```

Теперь, за исключением проигрывания звуковых фрагментов, можно сделать с отдельными приложениями все, что можно сделать с апплетами.

Для приложения можно воспользоваться другой версией метода getImage() класса Toolkit для загрузки графических данных прямо из файловой системы (без использования только URL-адреса), как это сделано в следующем классе фрейма:

```
class MainWndFrame extends Frame
{       Image Im;
        public MainWndFrame(String str) // конструктор
        {       super (str);
                // Сделать: Добавьте сюда код конструктора
                Toolkit T=getToolkit();
                Im=T.getImage("image.gif"); // файл из текущего каталога
        }
        public void paint(Graphics g)
        {       g.drawImage(Im,10,10,this);
        }
        // остальной код класса
        .....
}
```

В данном случае метод getImage() ищет файл image.gif в текущем каталоге файловой системы.

2. Потоки ввода-вывода в Java

Как уже упоминалось, апплеты не имеют доступа к файлам, расположенным на удаленном компьютере, на котором эти апплеты были запущены, хотя и могут при этом обращаться к файлам, которые находятся в каталоге сервера WWW, откуда эти апплеты были загружены. Обычные же приложения имеют доступ к локальной файловой системе компьютера, на котором они запущены, также они могут работать и с удаленными файлами (через сеть Internet или Intranet).

В любом случае, для создания автономных приложений или же апплетов, взаимодействующих с сервером WWW через сеть, необходимо познакомиться с классами, предназначенными для ввода и вывода. Приложение Java может работать с потоками нескольких типов:

- со стандартными потоками ввода-вывода;
- с потоками, связанными с локальными файлами;
- с потоками, связанными с файлами в оперативной памяти;
- с потоками, связанными с удаленными файлами.

2.1. Обзор классов Java для работы с потоками

Рассмотрим кратко классы, связанные с потоками.

Стандартные потоки

Для работы со стандартными потоками в классе System имеется 3 статических объекта: System.in, System.out и System.err. Поток System.in связан с клавиатурой, потоки System.out и System.err - с консольным выводом приложения Java.

Базовые классы для работы с потоками и файлами

Рассмотрим предварительно иерархию классов, предназначенных для ввода-вывода (пакет java.io). Все основные классы, рассматриваемые далее, произошли от класса Object:

Object → InputStream

Object → OutputStream

Object → RandomAccessFile

Object → File

Object → FileDescriptor

Object → StreamTokenizer

Класс InputStream является базовым для большого количества классов, на базе которых создаются потоки ввода. В основном, именно эти производные классы применяются программистами, так как в них имеется намного более мощные методы, чем в классе InputStream.

Аналогично класс OutputStream служит в качестве базового для различных классов, имеющих отношение к потокам вывода.

С помощью класса RandomAccessFile можно организовать работу с файлами в режиме прямого доступа, когда программа может указать смещение относительно начала файла и размер блока данных, над которым выполняется операция ввода или вывода. (Отметим, что классы InputStream и OutputStream также можно использовать для обращения к файлам в режиме прямого доступа.)

Класс File предназначен для работы с каталогами и файлами. С помощью этого класса можно получить список файлов и каталогов, расположенном в заданном каталоге, создать и удалить каталог, переименовать файл или каталог, а также выполнить некоторые другие операции.

С помощью класса FileDescriptor можно проверить идентификатор открытого файла.

Очень полезен класс StreamTokenizer, он позволяет организовывать выделение из входного потока данных элементов, отделенных друг от друга заданными разделителями, такими, например, как запятая, пробел и др.

Классы, производные от InputStream

От класса InputStream производится много других классов, методы которых позволяют работать с потоком ввода не на уровне отдельных байт, а на уровне объектов различных классов, например, класса String. Рассмотрим иерархию производных классов:

Object → InputStream → FileInputStream

Object → InputStream → PipedInputStream

Object → InputStream → SequenceInputStream

Object → InputStream → StringBufferInputStream

Object → InputStream → ByteArrayInputStream

Object → InputStream → FilterInputStream

Object → InputStream → FilterInputStream → BufferedInputStream

Object → InputStream → FilterInputStream → DataInputStream

Object → InputStream → FilterInputStream → LineNumberInputStream

Object → InputStream → FilterInputStream → PushBackInputStream

Класс FileInputStream позволяет создать поток ввода на базе класса File Или FileDescriptor.

С помощью классов PipedInputStream и PipedOutputStream можно организовать двустороннюю передачу данных между двумя одновременно работающими задачами многопотокового приложения.

Класс SequenceInputStream позволяет объединить несколько входных потоков в один поток. Если в процессе чтения будет достигнут конец первого потока такого объединения, в дальнейшем чтение будет выполняться из второго потока и т.д.

Класс StringBufferInputStream позволяет создавать потоки ввода на базе строк класса String, используя при этом только младшие байты хранящихся в такой строке символов. (Этот класс может служить дополнением для класса ByteArrayInputStream, который также предназначен для создания потоков на базе данных из оперативной памяти.)

При необходимости можно создать входной поток данных не на базе локального или удаленного файла, а на базе массива, расположенного в оперативной памяти. Класс ByteArrayInputStream предназначен именно для этого - конструктору класса передается ссылка на массив и в итоге создается входной поток данных, связанный с этим массивом. (Потоки в оперативной памяти могут быть использованы для временного хранения данных. Так как апплеты не могут обращаться к локальным файлам, для создания временных файлов можно использовать потоки в оперативной памяти на базе ByteArrayInputStream. Другую возможность создания временных файлов предоставляет пользователю класса StringBufferInputStream.)

Класс FilterInputStream является абстрактным классом, на базе которого создаются все следующие рассматриваемые классы потоков ввода.

Буферизация операций ввода и вывода в большинстве случаев значительно ускоряет работу приложений, так как при ее использовании сокращается количество обращений к системе для обмена данными с внешними устройствами. Класс BufferedInputStream может быть использован для организации буферизированных потоков ввода. Конструктору этого класса в качестве параметра передается ссылка на объект класса InputStream. Таким образом, нельзя просто создать объект класса BufferedInputStream, не создав при этом объект класса InputStream.

Очень часто возникает необходимость записывать в потоки данных и читать оттуда данные не только на уровне байт или текстовых строк, но объекты других типов, например, целые числа или числа типа double, массивы и т.д. Класс DataInputStream содержит методы, позволяющие извлекать из входного потока данные в перечисленных выше форматах (выполнять форматированный ввод данных). Этот класс также реализует интерфейс DataInput, служащий для этой цели.

С помощью класса LineNumberInputStream можно работать с текстовыми потоками, состоящими из отдельных строк, разделенных символами возврата каретки \r и перехода на следующую строку \n. Методы этого класса позволяют следить за нумерацией строк в таких потоках.

Класс PushBackInputStream позволяет возвратить в поток ввода только что прочитанный оттуда символ, с тем, чтобы после этого данный символ можно было прочитать снова.

Классы, производные от OutputStream

Класс OutputStream предназначен для создания потоков вывода. Приложения, как правило, непосредственно не используют этот класс для операций вывода, так же как и класс InputStream для операций ввода. Вместо этого применяются классы, имеющие следующую иерархию:

Object → OutputStream → FileOutputStream

Object → OutputStream → PipedOutputStream

Object → OutputStream → ByteArrayOutputStream

Object → OutputStream → FilterOutputStream

Object → OutputStream → FilterOutputStream → BufferedOutputStream

Object → OutputStream → FilterOutputStream →DataOutputStream

Object → OutputStream → FilterOutputStream → PrintStream

Класс FileOutputStream позволяет создать поток вывода на базе класса File или FileDescriptor.

Классы PipedOutputStream и PipedInputStream предназначены для организации двусторонней передачи данных между двумя одновременно работающими задачами многопоточного приложения.

С помощью класса ByteArrayOutputStream можно создать поток вывода в оперативную память.

Абстрактный класс FilterOutputStream служит базовым классом для всех рассматриваемых ниже потоков вывода.

Класс BufferedOutputStream предназначен для создания буферизированных потоков вывода. Как уже отмечалось, буферизация ускоряет работу приложения.

С помощью класса DataOutputStream приложения Java могут выполнять форматированный вывод данных. Класс DataOutputStream реализует интерфейс DataOutput.

Потоки, созданные с использованием класса PrintStream, предназначены для форматного вывода данных различных типов с целью их визуального представления в виде текстовой строки.

2.2 Стандартные потоки ввода-вывода

Для работы со стандартными потоками в классе System имеется 3 статических объекта: System.in, System.out и System.err. По своему назначению эти потоки больше всего напоминают стандартные потоки ввода, вывода и вывода сообщений об ошибках операционной системы MS-DOS.

Поток System.in (стандартный поток ввода) связан с клавиатурой, потоки System.out (стандартный поток вывода, вывод при помощи этого потока может быть переназначен в файл) и System.err (стандартный поток вывода ошибок) - с консольным выводом приложения Java.

Отметим, что стандартные потоки, как правило, не используются апплетами, так навигаторы WWW общаются с пользователем через окно апплета и извещения от мыши и клавиатуры, а не через консоль.

Стандартный поток ввода

Стандартный поток ввода System.in определен как статический объект класса InputStream, который содержит только простейшие методы для ввода данных. Самые необходимые методы:

public int read();

public int read(byte b[]);

Первый метод читает следующий вводимый с клавиатуры символ, при достижении символа конца ввода метод возвращает -1.

Второй метод читает данные из потока в массив, ссылка на который передается через единственный параметр. Количество считанных данных определяется размером массива, т.е. значением b.length. Метод возвращает количество считанных байт данных или-1, если достигнут конец потока.

При возникновении ошибок при работе со стандартным потоком ввода создается исключение IOException, обработку которого необходимо предусмотреть.

Замечание. На машинах, работающих с Unix и Dos/Windows, символы конца ввода формируются разными способами. В Unix должны быть использованы символы Ctrl+D (^D), а в среде Dos/Windows используются Ctrl+Z (^Z).

Стандартный поток вывода

Стандартный поток вывода System.out создан на базе класса PrintStream, предназначенного для форматированного вывода данных различного типа с целью их визуального отображения в виде текстовой строки.

Для работы со стандартным потоком вывода чаще всего используются методы print() и println(), хотя метод write() также доступен.

В классе PrintStream определено несколько реализаций метода print() с параметрами различных типов:

public void print(boolean b);

public void print(char c);

public void print(char s[]);

public void print(double d);

public void print(float f);

public void print(int i);

public void print(long l);

public void print(Object obj);

public void print(String s);

При помощи этих методов можно записать в стандартный поток вывода текстовое представление данных различного типа.

Для каждого метода print() существует подобный ему метод println(), который отличается от соответствующего метода print() тем, что добавляет к записываемой в поток строке символ перехода на следующую строку. Кроме того, существует метод println() без параметров, он записывает в поток только символ перехода на следующую строку.

Стандартный поток вывода сообщений об ошибках

Стандартный поток вывода сообщений об ошибках System.err так же как и поток out, создан на базе класса PrintStream. Поэтому для вывода сообщений об ошибках также можно использовать вышеописанные методы print() и println().

Пример использования стандартных потоков ввода и вывода

Рассмотрим приложение TestInOut *(пример 2)*, в котором пользователю предлагается ввести строку, состоящую из чисел, разделенных пробелами. После ввода строка сначала печатается по отдельным словам, а затем по числам, входящим в нее.

```
/*------------- Пример 2. Файл TestInOut.java -------------*/
class TestInOut
{  public static void main(String ARGV[]) throws java.io.IOException
   {        StringBuffer buffer=new StringBuffer();
            int c;
            System.out.print("Для прекращения ввода используется символ:");
            System.out.println("в Windows - Ctrl+Z, в Unix - Ctrl+D");
            System.out.println(
                     "Введите несколько слов (чисел), разделенных пробелом:");
            // -------------посимвольный ввод до символа конца ввода
            while((c=System.in.read())!=-1)
            {        buffer.append((char)c); // добавить символ в буфер
            }
            // -------------вывод по словам (токенам)
            System.out.println("\n\nВведенные слова:");
            String string=new String(buffer); // создать по буферу строку
            java.util.StringTokenizer tok=new java.util.StringTokenizer(string);
            while(tok.hasMoreTokens()) // пока есть токены
            {        // получить токен и напечатать
                     String word=tok.nextToken(); System.out.print(word+" ");
            }
```

```
            // ----------вывод по числам
            System.out.println("Введенные числа:");
            tok=new java.util.StringTokenizer(string);
            while(tok.hasMoreTokens()) // пока есть токены
            {       // преобразовать токен в целочисленный упаковщик
                    Integer I=Integer.valueOf(tok.nextToken());
                    // получить из упаковщика целое число и напечатать
                    System.out.print(" "+ I.intValue());
            }
    }
}
```

2.3 Потоки, связанные с локальными файлами

Рассмотрим процесс создания и использования потоков, связанных с локальными файлами, а также вопрос закрытия потоков после использования и принудительный сброс буферов.

2.3.1 Создание потоков, связанных с локальными файлами

Для создания входного или выходного потока, связанного с локальным файлом следует воспользоваться классами из библиотеки Java, созданными на базе классов InputStream и OutputStream. Однако необходимо всегда помнить о некоторых особенностях методики использования рассматриваемых ранее потоков.

Особенность этой методики заключается в том, для создания потоков, связанных с файлами, необходимо воспользоваться сразу несколькими классами, а не одним, на первый взгляд казалось бы, наиболее подходящим.

Рассмотрим пример. Пусть необходимо создать выходной поток для записи форматированных данных (например, текстовых строк). За форматированный вывод отвечает поток класса DataOutputStream. Однако, сразу создать объект класса DataOutputStream, связанный с файлом нельзя, так как в классе DataOutputStream предусмотрен только один конструктор, которому в качестве параметра необходимо передать ссылку на объект класса OutputStream:

public DataOutputStream(OutputStream out);

Что же касается конструктора класса OutputStream, то он выглядит следующим образом:

public OutputStream();

Так как ни в одном, ни в другом конструкторе не предусмотрено никаких ссылок на файлы, то пока совершенно непонятно, как с использованием одних только классов

DataOutputStream и OutputStream можно создать выходной поток, связанный с локальным файлом.

Создание потока форматированных данных, связанного с локальным файлом.

Создание потоков, связанных с файлами и предназначенных для форматированного ввода и вывода, необходимо выполнять в несколько приемов. При этом вначале необходимо создать потоки на базе класса FileOutputStream или FileInputStream, а затем передать ссылку-на созданный поток конструктору класса DataOutputStream или DataInputStream.

В классах FileOutputStream и FileInputStream предксмотрены конструкторы, которым в качестве параметроа передается либо ссылка на объект класса File, либо ссылка на объект класса FileDescriptor, либо текстовая строка пути к файлу, например:

public FileOutputStream(File fileObj);

public FileOutputStream(FileDescriptor fdObj);

public FileOutputStream(String name);

Таким образом, если нужен выходной поток для записи в файл форматированных данных, то это делается в два этапа:

- сначала необходимо создать поток как объект класса FileOutputStream;
- затем ссылку на этот поток следует передать конструктору класса DataOutputStream.

Полученный таким способом объект dataOut класса DataOutputStream

DataOutputStream dataOut;

dataOut=new DataOutputStream(new FileOutputStream("output.txt"));

можно использовать как выходной поток, связанный с файлом output.txt, записывая в него форматированные данные:

Для создания входного потока форматированных данных из файла следует пользоваться классами FileInputStream и DataInputStream, действуя по аналогичной методике.

Добавление буферизации.

В случае, если необходим не простой выходной или входной поток, а буферизированный, необходимо еще воспользоваться классом BufferedOutputStream или BufferedIntputStream. Вот два конструктора, предусмотренные, например, в классе BufferedOutputStream:

public BufferedOutputStream(OutputStream out);

public BufferedOutputStream(OutputStream out, int size);

Первый из них создает буферизированный выходной поток на базе потока класса OutputStream, а второй делает то же самое, но дополнительно позволяет указать размер буфера в байтах

Если необходимо создать выходной буферизированный поток для записи форматированных данных, связанный с файлом, то создание потока выполняется в три приема:

- создается поток, связанный с файлом, как объект класса FileOutputStream;
- ссылка на этот поток передается конструктору класса BufferedOutputStream, в результате чего образуется буферизированный поток, связанный с файлом;
- ссылка на буферизированный поток, созданный на предыдущем шаге, передается конструктору класса DataOutputStream, который и создает нужный поток.

Вот фрагмент исходного текста программы, который создает выходной буферизированный поток для записи форматированных данных в файл output.txt:

```
DataOutputStream dataOut;
dataOut=new DataOutputStream(
        new BufferedOutputStream(
                new FileOutputStream("output.txt")));
```

Для создания входного буферизированного потока форматированных данных из файла следует пользоваться классами FileInputStream, BufferedIntputStream и DataInputStream, действуя по аналогичной методике.

Исключения при создании потоков, связанных с файлами.

При создании потоков на базе класса FileOutputStream или FileInputStream могут возникать исключения FileNotFoundException (попытка открыть входной поток данных для несуществующего файла), SecurityException (попытка открыть файл, для которого запрещен доступ, например, попытка открытия файла с атрибутом read-only для записи) и IOException (если файл не может быть открыт для записи по каким-либо другим причинам).

2.3.2 Запись данных в поток и чтение их из потока

Для обмена данными с потоками можно использовать как методы write() и read(), так и методы, допускающие ввод или вывод форматированных данных. В зависимости от того, на базе какого класса создан поток, зависит набор доступных методов, предназначенных для чтения или записи данных.

Простейшие методы.

Для выходного потока на базе класса FileOutputStream можно использовать для записи в него 3 разновидности метода write():

- public void write(byte b[]) - записывает в поток содержимое массива (длиной b.length байт), ссылка на который передается через параметр, начиная с текущей позиции в потоке (после выполнения текущая позиция продвигается вперед).
- public void write(byte b[], int off, int len) - позволяет дополнительно указать начальное смещение off записываемого блока данных в массиве и колтчество записываем байт len

массива.

- public void write(int b) - просто записывает в поток 1 байт данных.

Если в процессе записи происходит ошибка, то возникает исключение IOException.

Для входного потока на базе класса FileInputStream можно использовать для чтения из него 3 разновидности метода read():

- public int read() - просто читает из потока 1 байт данных. Если достигнут конец файла, товозвращается значение -1.
- public int read(byte b[]) - читает данные в массив, причем количество прочитанных данных определяется размером массива (b.lenght байтов). Метод возвращает количество прочитанных байтов данных или значение -1, если в процессе чтения был достигнут конец файла.
- public int read(byte b[], int off, int len) - читает данные в область массива, заданную своим смещением и длиной.

Если в процессе чтения происходит ошибка, то возникает исключение IOException.

Методы для чтения и записи форматированных данных.

Классы DataOutputStream и DataInputStream предлагают более удобные методы для записи и чтения из потока, допускающие форматированный вывод и ввод данных.

Вот, например, какой набор методов можно использовать для записи форматированных данных в поток класса DataOutputStream:

public final void writeBoolean(boolean v);

public final void writeByte(int v);

public final void writeBytes(String s);

public final void writeChar(int v);

public final void writeChars(String s);

public final void writeDouble(double v);

public final void writeFloat(float v);

public final void writeInt(int v);

public final void writeLong(long v);

public final void writeShort(int v);

public final void writeUTF(String str);

Хотя имена этих методов говорят сами за себя, сделаем несколько замечаний. Метод writeByte() записывает в поток 1 байт (младший байт параметра v). В отличие от метода writeByte() метод writeChar() записывает в поток двухбайтовое символьное значение в кодировке Unicode.

Если необходимо записать в выходной поток текстовую строку, то это можно сделать с помощью методов writeBytes(), writeChars() или writeUTF(). Первый метод записывает в выходной поток только младшие байты символов, второй - двухбайтовые символы в кодировке Unicode, третий предназначен для записи строки в машинно-независимой кодировке UTF-8.

Все перечисленные выше методы в случае возникновения ошибки создают исключение IOException, которое необходимо обработать.

Для чтения форматированных данных из потока класса DataInputStream используются следующие методы:

public final boolean readBoolean();

public final byte readByte();

public final char readChar();

public final double readDouble();

public final float readFloat();

public final void readFully(byte b[]);

public final void readFully(byte b[], int off, int len);

public final int readInt();

public final String readLine();

public final long readLong();

public final short readShort();

public final int readUnsignedByte();

public final int readUnsignedShort();

public final String readUTF();

public final static String readUTF(DataInput in);

public final int skipBytes(int n);

Следует обратить внимание на то, что среди этих методов нет тех, что специально предназначены для чтения данных, записанных из строк методами writeBytes() и writeChars() класса DataOutputStream.

Тем не менее, если входной поток состоит из отдельных строк, разделенных символами возврата каретки и перевода чтроки, то такие строки можно получить методом readLine(), например:

```
String s="",ss;
while((ss=dataIn.readLine())!=null) s=s+ss+"\r\n";
```

Также можно воспользоваться методом readFully, который заполняет массив байтов. Этот массив потом нетрудно преобразовать в строку типа String, так как в классе String предусмотрен соответствующий конструктор.

Отметим также метод skipBytes(), который позволяет пропустить из входного потока заданное количество байт.

Методы класса DataInputStream, предназначенные для чтения данных, могут создавать исключения IOException и EOFException. Первое из них возникает в процессе ошибки, второе - при достижении конца входного пока в процессе чтения.

2.3.3 Закрытие потоков

Так как в системе интерпретации приложений Java есть процесс «сборки мусора», то возникает вопрос - выполняет ли он автоматическое закрытие потоков, с которыми приложение завершило работу?

Нужно сказать, процесс «сборки мусора» не делает ничего подобного. «Сборка мусора» выполняется только для объектов, размещенных в оперативной памяти. Для потоков же программисты должны предусмотреть явное закрытие, для чего используется метод close().

2.3.4 Принудительный сброс буферов

Еще один важный момент работы связан с буферизированными потоками. Уже отмечалось, что буферизация ускоряет работу приложений с потоками, так как при ее использовании сокращается количество обращений к системе ввода-вывода. Можно постепенно в течении долгого времени добавлять в буферизированный поток данные, и только потом эти данные физически записать в файл на диске.

Во многих случаях, однако, приложение должно, не отказываясь совсем от буферизации, выполнять принудительную запись буферов в файл. Это можно сделать с помощью метода flush().

2.3.5 Приложение StreamDemo

Создадим приложение, которое демонстрирует наиболее распространенный способ работы с файлами через буферизированные потоки.

Задание. Создать автономное приложение StreamDemo, в окне которого содержится область редактирования, в которой пользователь может вводить текст. Приложение должно иметь выпадающее меню «File» (пункты «Open File» и «Save File»). При выборе пункта «Open File» в область редактирования записывается содержимое файла out.txt, а при выборе пункта Save File» содержимое области редактирования записывается в файл out.txt. Для записи и чтения необходимо использовать буферизированные потоки форматированных данных.

Замечание. Исходные тексты этого приложения сохранить для их последующей модификации в процессе создания приложения FileDialogDemo (см. далее).

Методические указания. Приложение должно быть создано на основе шаблона, содержащегося в Приложении 7.

Объявление элементов класса фрейма.

В классе фрейма объявим следующие элементы - ссылки на объекты классов:

TextArea text; // ссылка на область редактирования

Конструктор класса фрейма (метод MainWndFrame()).

В конструкторе класса фрейма создадим панель меню mainMenu класса MenuBar и при помощи метода setMenuBar() установим эту панель для фрейма.

После этого создадим меню «File» - объект file класса Menu и добавим его в панель меню mainMenu методом add().

Затем в меню file введем методом add() элементы «Open File» и «Save File» - объекты класса MenuItem.

В качестве поля редактирования создадим текстовую область text класса TextArea, а затем добавим ее в центральную часть фрейма методом add().

Обработка событий (метод handleEvent()).

В этот метод добавим обработку события Event.ACTION_EVENT (сообщения от меню). Следует обратить внимание, что фрагменты кода, где будет происходить работа с файлами, должны быть заключены в блоки try-catch, что является необходимым для обработки возможных исключительных ситуаций:

```
case Event.ACTION_EVENT:
        if(evt.target instanceof MenuItem) // события меню
        {       String label=(String) evt.arg;
                if(label.equals("Open File"))
                {       // обработка исключения при работе с файлами
                        try
                        {       // добавить необходимые действия
                        }
                        catch(IOException e)
                        {       // вывод названия возникшего исключения
                                System.out.println(e.toString());
                        }
                }
                else if(label.equals("Save File"))
```

```
            {       // обработка исключения при работе с файлами
                    try
                    {       // добавить необходимые действия
                    }
                    catch(IOException e)
                    {       // вывод названия возникшего исключения
                            System.out.println(e.toString());
                    }
            }
            else return false;
    }
    return true;
```

При выборе пользователем пункта меню «Open File» объявим переменные s и ss класса String (переменную s проинициализируем пустой строкой).

Затем создадим входной буферизированный поток форматированных данных, связанный с файлом out.txt. Для этого выполним следующие действия:

- создадим поток класса FileInputStream, связанный с файлом out.txt, и ссылку на этот поток присвоим переменной file класса FileInputStream;
- поток file, связанный с файлом, передадим конструктору класса BufferedInputStream, в результате чего образуется буферизированный поток, ссылку на который присвоим переменной buf класса BufferedInputStream;
- буферизированный поток buf передадим конструктору класса DataInputStream, который и создаст нужный поток, ссылку на который присвоим переменной dataIn класса DataInputStream.

Далее прочитаем в строку s все строки из потока dataIn при помощи метода readLine(). После считывания файла поток dataIn закроем методом close().

Содержимое строки s запишем в текстовую область, применяя метод setText() для объекта text и передавая этому методу строку s.

При выборе пользователем пункта меню «Save File» объявим переменную s класса String, в которую при помощи метод setText() для объекта text прочитаем содержимое текстовой области.

Затем создадим выходной буферизированный поток форматированных данных, связанный с файлом out.txt. Для этого выполним следующие действия:

- создадим поток класса FileOutputStream, связанный с файлом out.txt, и ссылку на этот поток присвоим переменной file класса FileOutputStream;

- поток file, связанный с файлом, передадим конструктору класса BufferedOutputStream, в результате чего образуется буферизированный поток, ссылку на который присвоим переменной buf класса BufferedOutputStream;
- буферизированный поток buf передадим конструктору класса DataOutputStream, который и создаст нужный поток, ссылку на который присвоим переменной dataOut класса DataOutputStream.

Далее запишем строку s в поток dataOut при помощи метода writeBytes(). После записи выполним принудительный сброс буфера потока dataOut методом flush() и закроем поток dataOut методом close().

2.4 Потоки в оперативной памяти

С оперативной памятью можно работать также как и с файлом (а точнее говоря, с потоком). Это очень удобно во многих случаях, в частности, файлы, отображаемые на память можно использовать для передачи данных между одновременно работающими потоками выполнения задачи. Кроме того, так как апплетам запрещено обращаться к файлам, расположенным на удаленном компьютере, на котором эти апплеты были запущены, то при необходимости создания временных потоков ввода и вывода (временных файлов) последние могут быть размещены в оперативной памяти.

В библиотеке классов Java есть три класса, специально предназначенных для создания потоков в оперативной памяти. Рассмотрим эти классы подробнее.

Класс ByteArrayOutputStream

Класс ByteArrayOutputStream создан не базе класса OutputStream. В нем имеется два конструктора:

public ByteArrayOutputStream();

public ByteArrayOutputStream(int size);

Первый конструктор создает выходной поток в оперативной памяти с начальным размером буфера, равным 32 байта. Второй позволяет указать необходимый размер буфера.

В классе ByteArrayOutputStream определено несколько достаточно полезных методов:

- public void reset() - сбрасывает счетчик байтов, записанных в выходной поток. Если данные, записанные в поток, больше не нужны, то можно вызвать этот метод и использовать затем память, выделенную для потока, для записи других данных.
- public int size() - с помощью этого метода можно определить количество байт данных, записанных в поток.
- public byte[] toByteArray() - позволяет скопировать данные, записанные в поток, в массив байт. Этот метод возвращает ссылку на созданный для этой цели массив.
- public void wtiteTo(OutputStream out) - с помощью этого метода можно скопиро-

вать содержимое данного потока в другой выходной поток, ссылка на который передается методу через параметр.

Для выполнения форматированного вывода в поток в оперативной памяти необходимо создать поток на базе класса DataOutputStream, передав соответствующему конструктору ссылку на поток класса ByteArrayOutputStream

Класс ByteArrayInputStream

С помощью класса ByteArrayInputStream можно создать входной поток на базе массива байтов, расположенного в оперативной памяти. В этом классе определено два конструктора:

public ByteArrayInputStream(byte buf[]);

public ByteArrayInputStream(byte buf[], int offset, int length);

Первый конструктор получает через единственный параметр ссылку на массив, который будет использован для создания входного потока. Второй позволяет дополнительно указать смещение offset и размер области памяти length, которая будет использована для создания потока.

Вот несколько методов, определенных в классе ByteArrayInputStream:

public int available();

public int read();

public int read(byte b[], int off, int len);

public void reset();

public long skip(long n);

Наиболее интересным является метод available(), с помощью которого можно определить, сколько байт имеется во входном потоке.

Обычно класс ByteArrayInputStream используется совместно с классом DataInputStream, что позволяет организовать форматный вывод данных.

Класс StringBufferInputStream

Класс StringBufferInputStream предназначен для создания входного потока на базе текстовой строки класса String. Ссылка на эту строку передается конструктору класса StringBufferInputStream через параметр:

public StringBufferInputStream(String s);

В классе StringBufferInputStream определены те же методы, что и в только что рассмотренном классе ByteArrayInputStream.

Для более удобной работы можно создать поток форматированных данных, для чего на базе потока класса StringBufferInputStream конструируется поток класса DataInputStream.

3 Работа с локальной файловой система

В предыдущих разделах рассматривались классы, предназначенные для чтения и записи потоков. Однако часто возникает необходимость выполнения и таких операций, как определение атрибутов файла, создание или удаление каталогов, удаление файлов, получение списка всех файлов в каталоге и т.д.

Для выполнения всех этих операций в приложениях Java используется класс с именем File.

3.1 Работа с файлами и каталогами

Создать объект класса File можно при помощи вызова одного из конструкторов:

public File(File dir, String name)

public File(String path);

public File(String path, String name);

Первый конструктор имеет можно указать отдельно каталог и имя файла, для которого создается объект; второй конструктор имеет один параметр - строку пути к каталогу или файлу; третий конструктор позволяет указать полный путь к каталогу и имя файла.

После создания объекта класса File нетрудно определить атрибуты этого объекта, воспользовавшись следующими методами этого класса:

- exist() - проверяет существование файла или каталога.
- canRead() - проверяет возможность чтения из файла.
- canWrite() - проверяет возможность записи в файл.
- isDirectory() - проверка того, соответствует ли созданный объект каталогу.
- isFile() - проверка того, соответствует ли созданный объект файлу.
- getName() - возвращает имя файла или каталога (выделяет его из полного пути).
- getAbsolutePath() - возвращает абсолютный путь к файлу или каталогу, который может быть машинно-зависимым.
- getPath() - позволяет определить машинно-независимый путь к файлу или каталогу.
- getParent() - определяет родительский каталог для объекта.
- length() - определяет длину файла в байтах.
- lastModified() - определяет время последней модификации файла или каталога (выдает время в относительных единицах с момента запуска системы).

С помощью методов mkdir() и mkdirs() можно создавать новые каталоги. Первый из них создает один каталог, второй - все подкаталоги, ведущие к создаваемому каталогу.

Для переименования файла или каталога необходимо создавать два объекта класса File, один из которых соответствует старому имени, а второй - новому. Затем для первого из этих объектов нужно вызвать метод renameTo(), передавая ему в качестве параметра ссылку на второй объект.

Для сравнения объектов класса File можно воспользоваться методом equals(). Нужно только отметить, что этот метод сравнивает пути к файлам и каталогам, но не сами файлы и каталоги.

Для получения списка содержимого каталога, соответствующего созданному объекту класса File, следует воспользоваться методом list(). Один вариант этого метода возвращает массив строк с именами содержимого каталога. Второй позволяет получить список не всех объектов, хранящихся в каталоге, а только тех, что удовлетворяют условиям, определенным в передаваемом этому методу фильтре FilenameFilter.

Перед тем, как передать методу list() фильтр, этот фильтр необходимо создать. Сначала необходимо определить новый класс, реализующий интерфейс FilenameFilter, переопределив в нем метод accept() класса FilenameFilter. В этом методе определяется, какие файлы будут выбираться из общего списка. Если имя файла подходит критерию отбора, то метод accept() должен возвратить значение true, если не подходит - значение false. Например, создадим класс, определяющий фильтр на основе расширения файла:

```
public class ExtFilter implements FilenameFilter
{       private String extension; // для хранения строки вида ".ext"
        public ExtFilter(String ext); // e - расширение файла
        {       extension="."+ext; // формирование строки
        }
        public boolean accept(File dir,String fileName)
        {       // return true - для файлов, удовлетворяющих условию,
                // иначе - return false
                return(fileName.endsWith(extension); // только для файлов *.ext,
        }
}
```

Передавая затем объект класса фильтров методу list() класса File, можно получить отфильтрованные элементы каталога, например:

```
ExtFilter txtFiles=new ExtFilter("txt"); // объект-фильтр
File dir=new File("\\tmp\\example"); // объект-каталог
  // получение списка файлов с расширением .txt
  String d[]=dir.list(txtFiles);
```

for(int i=0; i<d.length; i++) System.out.println("\t"+d[i]);

3.2 Приложение DirList

Задание. Создать автономное приложение DirList, в окне которого содержится текстовая область, в которой выводится список всех файлов и каталогов, расположенных в заданном каталоге. Имена файлов выводятся строчными буквами, а имена каталогов - прописными.

Методические указания. Приложение должно быть создано на основе шаблона, содержащегося в Приложении 7.

Конструктор класса фрейма (метод MainWndFrame()).

В конструкторе класса фрейма объявим следующие переменные:

```
TextArea text; // текстовая область
File dir; // объект, ассоциированный с каталогом
String[] dirList; // список файлов и каталогов
File file; // объект, ассоциированный с файлом
```

В качестве поля редактирования создадим текстовую область text при помощи оператора new, установим для этой области запрет на редактирование методом setEditable(), а затем добавим эту область в центральную часть фрейма методом add().

Создадим объект класса File для каталога "c:" и ссылку на этот объект присвоим переменной dir. При помощи метода list() получим список dirList всех файлов и каталогов каталога dir.

Далее в цикле по i (i от 0 до dirList.length): создадим объект file, передавая конструктору строку dir.getPath()+"\\"+dirList[i] (полное имя каждого пункта списка); если объект file является файлом (проверяется при помощи метода isFile()), то добавить в текстовую область text имя файла строчными буквами, иначе - вывести имя каталога прописными буквами, например:

text.appendText(dirList[i].toLowerCase()+"\r\n");

3.3 Произвольный доступ к файлам

В ряде случаев, например при создании системы управления базой данных, требуется обеспечить произвольный доступ к файлу. Рассматриваемые в предыдущем примере потоки ввода-вывода пригодны лишь для последовательного доступа, так как в соответствующих классах нет средств позиционирования внутри файла.

Библиотека классов Java содержит класс RandomAccessFile, который предназначен специально для организации прямого доступа к файлам как для чтения, так и для записи. Этот класс позволяет считывать данные из файла прямо в массивы, строковые переменные и

переменные примитивных типов Java, причем данные в файл можно записывать в обратном порядке.

Класс RandomAccessFile можно создать с помощью либо объектов File, либо переменной типа String, в которой описывается путь к файлу:

public RandomAccessFile(File file, String mode);

public RandomAccessFile(String name, String mode);

При создании объекта RandomAccessFile указывается режим работы с файлом: читать файл или читать и записывать. Файл открывается на чтение и/или запись после реализации объекта RandomAccessFile. Открыв файл, можно читать или писать данные одним из простейших способов - либо байт за байтом, либо строку за строкой. Так же как и все методы класса File, каждый метод класса RandomAccessFile вызывает исключение IOException.

Позиционирование внутри файла обеспечивается методом seek(), в качестве параметра которому передается абсолютное смещение внутри файла. После вызова этого метода текущая позиция в файле устанавливается в соответствии с переданным параметром.

В любой момент можно определить текущую позицию внутри файла, вызвав функцию getFilePointer().

Еще один метод, имеющий отношение к позиционированию, называется skipBytes() - он продвигает текущую позицию в файле на заданное количество байт вперед.

Дескриптор файла можно определить с помощью метода getFD(), а длину файла в байтах - методом length().

Ряд методов предназначен для выполнения как обычного, так и форматированного ввода из файла. Этот набор аналогичен методам, определенным для потоков:

public int read();

public int read(byte b[]);

public int read(byte b[], int off, int len);

public final boolean readBoolean();

public final byte readByte();

public final char readChar();

public final double readDouble();

public final float readFloat();

public final void readFully(byte b[]);

public final void readFully(byte b[], int off, int len);

public final int readInt();

public final String readLine();

public final long readLong();

public final short readShort();

public final int readUnsignedByte();

public final int readUnsignedShort();

public final String readUTF();

Существуют также методы, позволяющие выполнять обычную или форматированную запись в файлы с прямым доступом:

public void write(byte b[]);

public void write(byte b[], int off, int len);

public void write(int b);

public final void writeBoolean(boolean v);

public final void writeByte(int v);

public final void writeBytes(String s);

public final void writeChar(int v);

public final void writeChars(String s);

public final void writeDouble(double v);

public final void writeFloat(float v);

public final void writeInt(int v);

public final void writeLong(long v);

public final void writeShort(int v);

public final void writeUTF(String str);

3.4 Просмотр локальной файловой системы

Для того, чтобы позволить пользователю просматривать локальную файловую систему, предназначен класс FileDialog пакета java.awt, который создает экземпляр диалогового окна. Этот класс закрыт для апплетов; им совершенно незачем иметь доступ к файловой системе локального компьютера, на который они загружены. Самостоятельные же приложения могут использовать класс FileDialog.

Методы класса FileDialog

- FileDialog(Frame parent, String title) - Конструктор, создает экземпляр класса, режим по умолчанию (FileDialog.LOAD)
- FileDialog(Frame parent, String title,int mode) - Конструктор, создает экземпляр класса, режим задается (FileDialog.LOAD или FileDialog.SAVE)
- getMode - Поучение режима диалога
- setDirectory - Задает каталог диалога
- setFile - При вызове до начала изображения диалога задает файл по умолчанию для диалога

- getFile - Получает имя определенного файла
- getDirectory - Получает имя каталога диалога
- setFilenameFilter - Устанавливает фильтр имени файла
- getFilenameFilter - Получает фильтр имени файла

Пользоваться классом FileDialog достаточно легко, его единственная задача - снабдить пользователя стандартным модальным диалоговым окном для просмотра файлов. Когда эта задача выполнена, применяется метод getFile(), чтобы получить имя файла, и метод getDirectory(), чтобы получить путь к файлу. Фактически файловый диалог не касается непосредственно системы файлов; он только делает доступным то, что выбрал пользователь.

Рассмотрим примеры использования класса FileDialog. Пусть в классе фрейма необходимо вызвать файловый диалог, тогда можно использовать следующие фрагменты кода:

```
// создание объекта и его использование
FileDialog dlg=new FileDialog(this,"Open File",FileDialog.LOAD);
dlg.setFile("*.txt"); // установка имени файла по умолчанию
if(!isVisible()) show(); // родительский фрейм должен быть видимым
dlg.show(); // отображение диалога
String path=dlg.getDirectory()+dlg.getFile(); // получить имя файла
// теперь можно создать входной поток, связанный с файлом,
// или создать объект класса RandomAccessFile для чтения
```

или

```
// создание объекта и его использование
FileDialog dlg=new FileDialog(this,"Save File",FileDialog.SAVE);
if(!isVisible()) show(); // родительский фрейм должен быть видимым
dlg.setFile("*.txt"); // установка имени файла по умолчанию
dlg.show(); // отображение диалога
String path=dlg.getDirectory()+dlg.getFile(); // получить имя файла
// теперь можно создать выходной поток, связанный с файлом,
// или создать объект класса RandomAccessFile для записи
```

Следует обратить внимание на проверку того, виден ли фрейм-родитель до отображения диалога: как и при работе со всеми диалоговыми окнами, фрейм-родитель должен быть активным до того, как производится попытка показать диалоговое окно.

Кроме установки режима можно изменить и поведение диалогового окна, задав FilenameFilter перед тем как диалог появится на экране. После создания нового класса и реализации в нем интерфейса FilenameFilter, объект этого класса можно передать классу

FileDialog методом setFilenameFilter(), Этим можно ограничить выбор тех файлов, которые FileDialog предоставляет пользователю.

3.5 Приложение FileDialogDemo

Задание. На основе приложения StreamDemo создать автономное приложение FileDialogDemo, в котором имя файла для чтения и записи содержимого области редактирования не задается статически в приложении, а выбирается пользователем при помощи соответствующих файловых диалогов. Для операций записи и чтения в файл использовать класс RandomAccessFile.

Методические указания. Приложение должно быть создано на основе приложения StreamDemo.

Обработка событий (метод handleEvent()).

При обработке событий от меню добавить выбор имени файла для открытия и выбор имени файла для сохранения (для этого использовать объекты класса FileDialog). Полученные от диалогов полные имена выбранных файлов сохранять в переменной path класса String.

Для чтения и записи в файл использовать не потоки dataIn и dataOut классов DataInputStream и DataOutputFile соответственно, а потоки dataIn и dataOut класса RandomAccessFile. Конструктору этого класса следует передать имя файла path и режим работы с файлом (чтение/запись) в соответствии с обрабатываем событием.

При записи в файл необходимо удалить вызов метода flush() для объекта dataOut, так как метод с таким именем отсутствует в классе RandomAccessFile.

Задания к лабораторной работе

Задание 1. Проверить и объяснить работу приложений StandAlone, TestInOut, рассматриваемых в данной главе в качестве примеров и отмеченных курсивом.

Задание 2. Создать приложения StreamDemo, DirList, FileDialogDemo и объяснить их работу.

Задание 3. Дать ответы на контрольные вопросы.

Контрольные вопросы

1. Какие ограничения накладываются на апплеты при работе с файлами?
2. Что необходимо сделать для создания автономного Java-приложения?
3. Как организовать графический интерфейс пользователя в самостоятельных приложениях?
4. С какими потоками может работать Java-приложение?
5. Какие существуют базовые классы Java для работы с потоками и файлами?
6. Какие классы Java являются производными от класса InputStream?
7. Какие классы Java являются производными от класса OutputStream?

8. Какие стандартные потоки ввода-вывода существуют в Java, каково их назначение? На базе каких классов создаются стандартные потоки?
9. Чем является поток System.in? Какими методами чаще всего пользуются при работе с этим потоком?
10. Чем является поток System.out? Какими методами чаще всего пользуются при работе с этим потоком?
11. Чем является поток System.err? Какими методами чаще всего пользуются при работе с этим потоком?
12. В чем заключается особенность создания потока, связанного с локальным файлом?
13. Как создать поток для форматированного обмена данными, связанного с локальным файлом?
14. Как добавить буферизацию для потока форматированного обмена данными, связанного с локальным файлом?
15. Выполняется ли процессом «сборки мусора» автоматическое закрытие потоков, с которыми приложение завершило работу?
16. За счет чего буферизация ускоряет работу приложений с потоками?
17. Когда применяется принудительный сброс буферов?
18. В каких случаях чаще всего используются потоки в оперативной памяти?
19. Для выполнения каких операций применяется класс File?
20. Для чего применяются фильтры файлов и как создать и использовать фильтр?
21. Для чего предназначен класс RandomAccessFile? Чем он отличается от потоков ввода и вывода?

ЛАБОРАТОРНАЯ РАБОТА №8

СЕТЕВЫЕ ПРИЛОЖЕНИЯ: ПЕРЕДАЧА ДАННЫХ С ИСПОЛЬЗОВАНИЕМ СОКЕТОВ (2 ЧАСА)

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ

Сетевые классы, содержащиеся в API (пакет java.net), можно разделить на две основные группы: классы, занимающиеся сокетами, и классы, занимающиеся URL. В данной главе будет обсуждаться первая группа классов.

1. Сокеты

В библиотеке классов Java есть очень удобное средство, с помощью которого можно организовать взаимодействие между приложениями Java, работающими как на одном и том же, так и на разных узлах сети. Это средство - так называемые сокеты.

Сокеты можно представить в виде двух розеток, в которые включен провод, предназначенный для передачи данных через сеть. В компьютерной терминологии - сокеты - это программный интерфейс, предназначенный для передачи данных между приложениями. Сокеты не реализуют метод передачи данных - они создают экземпляры более общих классов ввода/вывода, называемых потоками (thread), которые выполняют эту работу. Сокет переносит данные по сети; потоки занимаются тем, что закладывают данные в сокет и выдают их наружу.

Прежде чем приложение сможет выполнять передачу и прием данных, оно должно создать сокет, указав при этом адрес узла, номер порта, через который будут передаваться данные, и тип сокета - существует два типа сокетов: потоковые и датаграммные.

С помощью потоковых сокетов можно создавать каналы передачи данных между двумя приложениями Java в виде потоков ввода-вывода. Потоки могут быть входными или выходными, обычными или форматированными, с использованием или без использования буферизации.

Потоковые сокеты позволяют передавать данные только между двумя приложениями, так как они предполагают создание канала между эти двумя приложениями. Однако иногда нужно обеспечить взаимодействие нескольких клиентских приложений с одним сервером. В этом случае можно создавать в серверном приложении отдельные задачи и отдельные каналы для каждого клиентского приложения, либо воспользоваться датаграммными сокетами - они позволяют передавать данные всем узлам сети.

Для передачи данных через датаграммные сокеты не нужно создавать канал - данные непосредственно посылаются тому приложению, для которого они предназначены, с использованием адреса этого приложения в виде сокета и номера порта. При этом одно клиентское приложение может обмениваться данными с несколькими серверными приложениями или, наоборот, одно серверное приложение - с несколькими клиентскими.

Нужно отметить, что, к сожалению, датаграммные сокеты не гарантируют доставку передаваемых пакетов данных. Даже если пакеты данных, передаваемые через такие сокеты, дошли до адресата, не гарантируется, что они будут получены в той же самой последовательности, в которой были переданы. Потоковые сокеты, напротив, гарантируют доставку пакетов данных, причем в правильной последовательности.

Причина отсутствия такой гарантии доставки данных при использовании датаграммных сокетов заключается в использовании такими сокетами протокола с негарантированной доставкой UDP. Потоковые сокеты работают через протокол гарантированной доставки TCP.

2. Протокол TCP/IP, адрес IP и класс InetAddress

Идея сокетов неразрывно связана с TCP/IP (Transmission Contol Protocol/ Internet Protocol - Протокол контроля передачи/Межсетевой протокол Интернет) - протоколом, использующимся в Internet. По существу сокет является непрерывной связью данных между двумя хостами сети (все компьютеры, подключенные к сети, называются узлами или хостами (host)). Сокет определяется сетевым адресом конечного компьютера (endpoint), а также портом на каждом хосте. Компьютеры в сети направляют приходящие из сети потоки данных в специальные принимающие программы, присваивая каждой программе отдельный номер. - порт программы. Аналогично, когда генерируются выходные данные, программе, инициализирующей передачу данных, присваивается номер порта для транзакции. В TCP/IP резервируются определенные номера портов для специальных протоколов - например, 25 для SMTP и 80 для HTTP. Все номера портов, меньшие 1024, зарезервированы на каждом хосте для системного администратора.

Каждый хост в сети, осуществляющей связь по протоколу TCP/IP (в том числе и Internet) присвоен уникальный численный идентификатор, называемый IP-адресом. IP-адрес состоит из четырех десятичный чисел в диапазоне от 0 до 255 и в общем случае представляется числом с точками - например, 194.84.124.60.

Фактически адрес IP является 32-разрядным числом (4 байта), а упомянутые числа представляют собой отдельные байты адреса IP.

Так как работать с цифрами удобно только компьютеру, была придумана система доменных адресов (система DNS - Domain Name System). При использовании этой системы адресам IP ставится в соответствие так называемый доменный адрес, такой, как www.vvsu.ru. Человек задает имя хоста, и его компьютер запрашивает местный DNS-сервер, который определяет по данному имени IP-адрес.

Для работы с IP-адресами в библиотеке классов Java имеется класс InetAddress. Заметим, что при создании сокета можно задавать либо имя хоста в форме строки, либо IP-адрес в форме InetAddress. InetAddress для созданного уже сокета можно определить при помощи метода getInetAddress() класса сокетов Socket. Знать этот адрес бывает полезно, например, при необходимости открытия нового соединения с той же машиной. Тогда немного быстрее будет вместо имени хоста воспользоваться InetAddress, чтобы избежать дополнительного преобразования DNS.

Рассмотрим наиболее интересные методы класса InetAddress. Прежде всего необходимо создать объект класса InetAddress. Эта процедура выполняется не с помощью оператора new, а с применением статических методов getLocalHost(), getByName() и getAllByName.

Создание объекта класса InetAddress для локального узла

Метод getLocalHost() создает объект класса InetAddress для локального узла, то есть для той рабочей станции, на которой выполняется приложение Java. Так как этот метод статический, то его можно вызывать, используя имя класса InetAddress:

```
InetAddress iaLocal;
iaLocal=InetAddress.getLocalHost();
```

Создание объекта класса InetAddress для удаленного узла

В том случае, когда интерес представляет удаленный узел сети Internet или корпоративной сети Intranet, объект класса InetAddress для него можно создать с помощью методов getByName() или getAllByName(). Первый из них возвращает по имени узла узла, а второй - массив всех адресов IP, связанных с данным узлом. Если узла с таким не существует, при выполнении обоих методов возникает исключение UnknownHostException.

Методам getByName() или getAllByName() можно передавать не только имя узла, такое, как например www.vvsu.ru, но и строку адреса IP в виде четырех десятизначных чисел, разделенных точками.

Другие методы класса InetAddress

После создания объекта класса для локального или удаленного узла можно использовать и другие методы класса InetAddress.

Метод getAddress() возвращает массив из четырех байт IP-адреса объекта. Байт с нулевым индексом массива возвращает старший байт адреса IP.

С помощью метода getHostName можно определить имя узла, для которого был создан объект класса InetAddress.

Метод toString() возвращает текстовую строку, которая содержит имя узла, разделитель / и адрес IP в виде четырех десятичных чисел, разделенных точками.

Метод equals() предназначен для сравнения адресов IP как объектов класса InetAddress.

3. Потоковые сокеты

Как уже отмечалось, интерфейс сокетов позволяет передавать данные между двумя приложениями, работающих на одном или разных узлах сети. В процессе создания канала передачи данных одно из приложений выполняет роль сервера, а другое - роль клиента. После того, как канал будет создан, приложения становятся равноправными - они могут передавать друг другу данные симметричным образом.

3.1 Создание и использование канала передачи данных

Рассмотрим процесс создания канала в деталях.

Инициализация сервера

Первое, что должно сделать серверное приложение, - это создать объект класса ServerSocket, указав конструктору номер используемого порта:

SeverSocket ss=new ServerSocket(9999);

Объект класса SeverSocket вовсе не является сокетом, он предназначен всего лишь для установки канала связи с клиентским приложением, после чего создается сокет класса Socket, пригодный для передачи данных.

Установка канала связи с клиентским приложением выполняется при помощи метода accept(), определенного в классе ServerSocket:

Socket s=ss.accept();

Метод accept() приостанавливает работу вызвавшей его задачи до тех пор, пока клиентское приложение не установит канала связи с сервером. Если приложение однопоточное (однозадачное), его выполнение будет блокировано до момента установки канала связи. Избежать полной блокировки приложения можно, если выполнять создание канала передачи данных в рамках отдельной задачи.

Как только канал создан, можно использовать сокет сервера для образования входного и выходного потоков класса InputStream и OutputStream соответственно:

InputStream is=s.getInputStream();

OutputStream os=s.getOutputStream();

Эти потоки можно использовать таким же образом, как и потоки, связанные с файлами.

Следует обратить внимание, что при создании серверного сокета не указывается адрес IP и тип сокета. Во-первых, тип сокета указывается только для датаграммных сокетов. Во-вторых, что касается адреса IP, то он, очевидно, равен адресу IP узла, на котором запущено серверное приложение. В классе ServerSocket определен метод getInetAddress(), позволяющий при необходимости узнать этот адрес.

Инициализация клиента

Клиентскому приложению необходимо лишь создать сокет как объект класса Socket, указав адрес IP серверного приложения и номер порта, используемого сервером:

Socket s=new Socket(□ localhost□ , 9999);

Здесь в качестве адреса IP указывается специальный адрес localhost, предназначенный для тестирования серверных приложений (при запуске их на одном и том же узле), а в качестве номера порта - значение 9999, использованное сервером.

Теперь для клиентского приложения можно создавать входной и выходной потоки:

InputStream is=s.getInputStream();

OutputStream os=s.getOutputStream();

Передача данных между клиентом и сервером

После того, как серверное и клиентское приложение создали потоки для приема и передачи данных, оба этих приложения могут читать т писать в канал данных, вызывая методы read() и write(), определенные в классах InputStream и OutputStream. На базе потоков InputStream и OutputStream можно создать буферизированные потоки и потоки для передачи форматированных данных.

Завершение работы сервера и клиента

После завершения передачи данных необходимо закрыть входной и выходной потоки в приложениях, вызвав для объектов потоков метод close():

```
is.close();
os.close();
```

Когда канал перадачи данных больше не нужен, сервер и клиент должны закрыть сокет, вызвав метод close() класса Socket:

```
s.close();
```

Серверное приложение, кроме того, должно закрыть соединение, вызвав метод **close()** для объекта класса ServerSocket:

```
ss.close();
```

3.2 Конструкторы и методы класса Socket

Чаще всего для создания сокетов в клиентских приложениях используются один из двух конструкторов:

```
public Socket(String host, int port);
public Socket(InetAddress address, int port);
```

Первый из этих конструкторов позволяет указывать адрес серверного узла в виде текстовой строки (доменный адрес или IP-адрес), а второй - в виде ссылки на объект класса InetAddress. Вторым параметром задается номер порта, с использованием которого будут передаваться данные.

Заметим, что апплет может получить текстовую строку, содержащую доменный адрес того узла, с которого он был загружен на данный локальный хост, следующим вызовом:

```
String baseAddress=getCodeBase().getHost();
```

В класса Socket есть еще два конструктора, последний параметр которых задает типа сокета. Но работать с ними не рекомендуется. Для потоковых сокетов обычно используются первые два конструктора, а для работы с датаграммами - отдельный класс DatagramSocket.

Методы getInputStream() и getOutputStream() класса Socket предназначены для создания соответственно входного и выходного потоков. Эти потоки связаны с сокетом и должны быть использованы для передачи данных по каналу связи.

Методы getInetAddress() и getPort() позволяют определить адрес IP и номер порта, связанные с данным сокетом (для удаленного узла). Метод getLocalPort() возвращает для данного сокета номер локального порта.

После того, как работа с сокетом завершена, его необходимо закрыть методом close().

3.3 Пример использования потоковых сокетов

Рассмотрим в качестве примера два приложения Server и Client (пример 1), работающих с потоковыми сокетами. Одно из них выполняет роль сервера, а другое служит клиентом. Эти приложения иллюстрируют применение потоковых сокетов для создания канала и передачи данных между клиентским и серверным приложением.

Приложение Server выводит на печать строку "Socket Server Application" и затем переходит в состояние ожидания соединения с клиентским приложением.

Приложение Client выводит сообщение "Socket Client Application" и устанавливает соединение с сервером, используя потоковый сокет с тем же самым номером, какой был задан в серверном приложении. В качестве запуска процесса передачи данными клиентское приложение помещает в свой выходной поток нулевой символ для передачи его серверу.

До нажатия пользователем символа ‘Q’ (или ‘q’) приложение Client в цикле: через свой входной поток получает от сервера количество переданных серверу символов; печатает это число; дает возможность пользователю ввести символ на консоли (любой символ и Ctrl-Z в качестве признака конца ввода); а затем передает этот символ серверу посредством записи его в свой выходной поток.

В свою очередь приложение Server в цикле: считывает из своего входного потока символ, переданный ему клиентом; в случае нажатия на клавишу ‘Q’ (или ‘q’) прекращает свою работу, иначе увеличивает количество переданных ему символов и записывает его в свой выходной поток для передачи его клиенту.

Отметим, что для получения и приема данных используются буферизированные потоки форматированных данных.

```
/*---------- Пример 1. Файл Server.java ----------*/
import java.io.*;
import java.net.*;
class Server
{  public static void main(String args[])
   {        char b; int count=-1; boolean quit=false;
            System.out.println("Socket Server Application");
            System.out.println("---- Start ----");
            try
```

```
{       // установка канала связи
        ServerSocket ss=new ServerSocket(2000);
        // ожидание соединения
        Socket s=ss.accept();
        // входной поток для приема данных от клиента
        DataInputStream dataIn=new DataInputStream(
                        new BufferedInputStream(
                                s.getInputStream()));
        // выходной поток для записи данных для клиента
        DataOutputStream dataOut=new DataOutputStream(
                        new BufferedOutputStream(
                                s.getOutputStream()));
        // цикл обработки команд от клиента
        while(!quit)
        {       // чтение символа, переданного клиентом
                b=dataIn.readChar(); System.out.println(""+b);
                if(b!=-1)
                {       if(b=='Q'||b=='q') quit=true;
                        else
                        {       count++;
                                // передача числа клиенту
                                dataOut.writeInt(count);
                                dataOut.flush();
                        }
                }
        }
        dataIn.close(); // закрытие входного потока
        dataOut.close(); // закрытие выходного потока
        s.close(); // закрытие сокета
        ss.close(); // закрытие соединения
}
catch(IOException ioe)
{       System.out.println(ioe.toString());
}
System.out.println("---- Finish ----");
```

```
    }
}

/*------------- Пример 1. Файл Client.java -------------*/
import java.io.*;
import java.net.*;
class Client
{   public static void main(String args[])
    {           int b=0; boolean quit=false;
                System.out.println("Socket Client Application");
                try
                {           // сокет для связи с сервером
                            // в процессе отладки сетевых приложений, когда
                            // сервер и клиент работают на одном узле,
                            // в качестве адреса сервера используется строка
                            // "localhost" - «сервер работает на этом же узле»
                            Socket s=new Socket("localhost",2000);
                            // в рабочей версии клиента указывается IP-адрес сервера
                            //Socket s=new Socket("194.84.124.60",2000);
                            //Socket s=new Socket("gosha",2000);
                            //Socket s=new Socket("gosha.vvsu.ru",2000);
                            // входной поток для приема данных от сервера
                            DataInputStream dataIn=new DataInputStream(
                                        new BufferedInputStream(
                                                    s.getInputStream()));
                            // выходной поток для передачи данных серверу
                            DataOutputStream dataOut=new DataOutputStream(
                                        new BufferedOutputStream(
                                                    s.getOutputStream()));
                            // инициализирующая передача данных серверу
                            dataOut.writeChar(0); dataOut.flush();
                            // цикл ввода команд серверу
                            while(!quit)
                            {           // чтения числа, переданного сервером
                                        if(b!=-1)
```

```
                    {       b=dataIn.readInt();
                            System.out.println(" "+b);
                    }
                    // ввод пользователем символа
                    b=System.in.read();
                    if(b!=-1)
                    {       if((char)b=='Q'||(char)b=='q') quit=true;
                            // передача символа серверу
                            dataOut.writeChar(b);
                            dataOut.flush();
                    }
            }
            dataIn.close(); // закрытие входного потока
            dataOut.close(); // закрытие выходного потока
            s.close(); // закрытие сокета
        }
        catch(IOException ioe)
        {       System.out.println(ioe.toString());
        }
    }
}
```

4. Датаграммные сокеты (несвязываемые датаграммы)

Протокол TCP/IP поддерживает также доставку несвязываемых датаграмм и их возврат с помощью UDP (User Datagram Packet, Пакет пользовательских датаграмм). Датаграммы UDP, также как и сокеты TCP, позволяют связаться с удаленным хостом по протоколу TCP/IP. В отличие от сокетов TCP, осуществляющих связь с логическим соединением, датаграммы, UDP связываются без логического соединения. Если сокет TCP можно сравнить с телефонным звонком, то датаграмму UDP можно сравнить с телеграммой. Доставка датаграммы UDP не гарантирована, и даже если такая датаграмма доставлена, нет гарантии, что она доставлена правильно и, возможно, придется иметь дело с утерянными или неупорядоченными пакетами данных.

Из-за ненадежности UDP большинство программистов при написании сетевых приложений предпочитают работать с сокетами TCP. Однако они работают быстрее потоковых сокетов и обеспечивают возможность широковещательной рассылки пакетов данных одновременно всем узлам сети.

Для работы с датаграммами приложение должно создать объект на базе класса DatagramSocket, а также подготовить объект класса DatagramPacket, в который будет записан принятый от партнера по сети блок данных.

Канал, а также входные и выходные потоки создавать не нужно. Данные передаются и принимаются методами send() и receive(), определенными в классе DatagramSocket.

4.1 Конструкторы и методы класса DatagramSocket

Рассмотрим конструкторы и методы класса DatagramSocket, предназначенный для создания и использования несвязанных датаграмм.

public DatagramSocket(int port);

public DatagramSocket();

Первый из этих конструкторов позволяет определить порт для сокета, второй предполагает использование любого свободного порта.

Обычно серверные приложения работают с использованием какого-то заранее определенного порта, номер которого известен клиентским приложениям. Поэтому для серверных приложений больше подходит первый из приведенных выше конструкторов.

Клиентские приложения, напротив, часто применяют свободные на локальном узле порты, поэтому для них годится конструктор без параметров.

С помощью метода getLocalPort() приложение всегда может узнать номер порта, закрепленного за данным сокетом.

Прием и передача данных на датаграммном сокете выполняется соответственно с помощью методов receive() и send() (метод receive() ждет(!) получения датаграммы). В качестве параметра этим методам передается ссылка на пакет данных (соответственно принимаемый и передаваемый), определенный как объект класса DatagramPacket (см. далее).

Так как «сборка мусора» в Java выполняется только для объектов, находящихся в оперативной памяти, то такие объекты как потоковые и датаграммные сокеты следует закрывать самостоятельно. В классе DatagramSocket для этого определен метод close().

Конструкторы и методы класса DatagramPacket

Перед тем как принимать или предавать данные с использованием методов receive() и send() класса DatagramSocket, необходимо подготовить объекты класса DatagramPacket. Метод receive() запишет в такой объект принятые данные, а метод send() перешлет данные из объекта класса DatagramPacket узлу, адрес которого указан в пакете.

Подготовка объекта класса DatagramPacket для приема пакетов выполняется с помощью следующего конструктора:

public DatagramPacket(byte buf[], int length);

Этому конструктору передается ссылка на массив buf, в который нужно будет записать данные и размер этого массива length.

Если необходимо подготовить пакет для передачи, то следует воспользоваться конструктором, который дополнительно позволяет задать IP-адрес узла и номер порта адресата:

public DatagramPacket(byte buf[], int length, InetAddress add, int port);

Таким образом, информация о том, в какой узел и на какой порт необходимо доставить пакет данных, хранится не в сокете, а в пакете, то есть в объекте класса DatagramPacket.

Метод getData() класса DatagramPacket возвращает ссылке на массив данных пакета. Размер пакета, данные из которого храняться в этом массиве, легко определить с помощью метода getLength(). Методы getAddress() и getPort() позволяют определить адрес и номер порта узла , откуда пришел пакет, или узла, для которого предназначен пакет.

Проблемы клиент-серверной системы

Если создается клиент-серверная система, в которой сервер имеет заранее известный адрес и номер порта, а клиенты - произвольные адреса и различные номера портов, то после получения пакета от клиента сервер может определить с помощью методов getAddress() и getPort() адрес клиента для установления с ним связи.

Если адрес сервера неизвестен, клиент может получать широковещательные пакеты, указав в объекте класса DatagramPacket адрес сети.

Как узнать адрес сети? Адрес IP состоит из двух частей - адреса сети и адреса узла. Для разделения компонент 32-разрядного адреса IP используется 32-разрядная маска, в которой битам адреса сети соответствуют единицы, а битам узла - нули.

Например, например адрес узла в сети может быть указан как 194.84.124.60. Исходя из значения старшего байта адреса, это есть сеть класса C, для которой по умолчанию используется маска 255.255.255.0. Следовательно, адрес сети будет 194.84.124.0.

4.3 Пример использования датаграммных сокетов

Рассмотрим в качестве примера два приложения ServerDatagram и ClientDatagram (пример 2), работающих с датаграммными сокетами. Одно из них выполняет роль сервера, а другое служит клиентом. Эти приложения иллюстрируют применение датаграммных сокетов для передачи данных от нескольких копий одного и того же клиентского приложения одному серверу с его известными адресом и номером порта.

Клиентские приложения посылают серверу строки, которые пользователь вводит с клавиатуры. Сервер принимает эти строки, отображая их в своем консольные вместе с номером порта клиента и его адресом. Когда с консоли клиента будет введена строка «QUIT» (или «quit»), этот клиент и сервер завершают свою работу. Работа других клиентов может быть завершена подобным образом, причем независимо от того, работает север или нет.

```
/*---------- Пример 2. Файл ServerDatagram.java ----------*/
import java.io.*;
import java.net.*;
import java.util.*;
class ServerDatagram
{   public static void main(String args[])
    {       boolean quit=false;
            String str; StringTokenizer strFull;
            System.out.println("Server Application");
            System.out.println("---- Start ----");
            byte buf[]=new byte[512]; // буфер для данных
            InetAddress srcAdd; // адрес узла, откуда пришел пакет
            int srcPort; // порт, откуда пришел пакет
            try
            {       // создание сокета сервера
                    DatagramSocket s=new DatagramSocket(9999);
                    // создание пакета для приема команд
                    DatagramPacket pIn=new DatagramPacket(buf,512);
                    // цикл получения команд от клиента
                    while(!quit)
                    {       // ---- получение данных от клиентов
                            s.receive(pIn); // получение пакета
                            srcAdd=pIn.getAddress(); // адрес узла-отправителя
                            srcPort=pIn.getPort(); // порт узла-отправителя
                            // ---- при получении "quit" - выход
                            // удаление незначащих байт из массива
                            strFull=new StringTokenizer(new String(buf,0),"\r\n");
                            str=(String)strFull.nextElement();
                            if(str.toLowerCase().equals("quit")) quit=true;
                            // ---- вывод полученной от клиента строки
                            str="Address:"+srcAdd.toString()+
                                    " Port:"+srcPort+" String:"+str;
                            System.out.println(str);
                    }
                    s.close(); // закрытие сокета
```

```
            }
            catch(IOException ioe)
            {       System.out.println(ioe.toString());
            }
            System.out.println("---- Finish ----");
    }
}
/*------------- Пример 2. Файл ClientDatagram.java -------------*/
import java.io.*;
import java.net.*;
import java.util.*;
class ClientDatagram
{   public static void main(String args[])
    {       boolean quit=false; int length;
            String str; StringTokenizer strFull;
            System.out.println("Client Application");
            System.out.println("Enter any string or 'quit' to exit...");
            byte buf[]=new byte[512]; // буфер для передачи данных
            try
            {       // создание сокета клиента с
                    // использованием свободного порта
                    DatagramSocket s=new DatagramSocket();
                    // адрес узла, на котором запущен сервер
                    // в процессе отладки сетевых приложений, когда
                    // сервер и клиент работают на одном узле,
                    // в качестве адреса сервера используется адрес
                    // данного локального узла, выдаваемый методом
                    // InetAddress.getLocalHost();
                    InetAddress serverAdd=InetAddress.getLocalHost();
                    // в рабочей версии клиента указывается IP-адрес сервера
                    //InetAddress serverAdd=
                    //      InetAddress.getByName("194.84.124.60");
                    //InetAddress serverAdd=InetAddress.getByName("gosha");
                    //InetAddress serverAdd=
                    //      InetAddress.getByName("gosha.vvsu.ru");
```

```
                    // создание пакета для передачи команд серверу
                    DatagramPacket pOut=
                            new DatagramPacket(buf,512,serverAdd,9999);
                    // цикл посылки команд серверу
                    while(!quit)
                    {       // ввод строки с клавиатуры
                            length=System.in.read(buf);
                            if(length==1)
                                    // только символ перевода строки
                                    continue;
                            // ---- посылка данных
                            s.send(pOut); // посылка пакета
                            // ---- при вводе "quit" - выход
                            // удаление незначащих байт из массива
                            strFull=new StringTokenizer(new String(buf,0),"\r\n");
                            str=(String)strFull.nextElement();
                            if(str.toLowerCase().equals("quit")) quit=true;
                    }
                    s.close(); // закрытие сокета
            }
            catch(IOException ioe)
            {       System.out.println(ioe.toString());
            }
    }
}
```

Клиенты этой клиент-серверной системы не получают никакого подтверждения в ответ на переданные ему пакеты данных. Можно изменить программу клиента, добавив такую возможность. Однако следует учесть, что так как датаграммные сокеты не гарантируют доставки пакетов, ожидание ответа может продлиться бесконечно долго.

Чтобы избежать этого, приложение должно выполнять работу с сервером в отдельной задаче, причем главная задача должна ждать ответ ограниченное время.

5 Приложения ServerSocketApp и ClientSocketApp

Задание. На основе текстов приложений Server-Client (пример 1) создать приложения ServerSocketApp-ClientSocketApp, иллюстрирующие работу с потоковыми сокетами.

До ввода пользователем строки «QUIT» (или «quit») приложение ClientSocketApp должно в цикле: через свой входной поток получать от сервера количество переданных серверу строк (массивов байт); печатать это число; давать возможность пользователю ввести строку на консоли, помещая ее в массив байт; передавать этот массив байт серверу посредством записи его в свой выходной поток; преобразовывать массив байт в строку и в случае получения строки «QUIT» (или «quit») прекращать свою работу.

В свою очередь приложение ServerSocketApp в цикле должно: считывать из своего входного потока массив байт, переданный ему клиентом; преобразовывать массив байт в строку; печатать строку; в случае получения строки «QUIT» (или «quit») прекращать свою работу, иначе увеличивать количество переданных ему массивов байт и записывать это число в свой выходной поток для передачи его клиенту.

Методические указания. При создании приложений ServerSocketApp-ClientSocketApp воспользоваться методами ввода с консоли массива байт, создания строки на основе этого массива байт и выделения значащих символов из строк, как это сделано в приложениях ServerDatagram-ClientDatagram (пример 2).

Метод main() приложения ClientSocketApp (модификация метода main() приложения Client)

Добавим описание следующих переменных:

byte buf[]=new byte[512]; // буфер для передачи данных

String str; StringTokenizer strFull; // для выделения из буфера строки

В качестве запуска процесса передачи данными поместим в выходной поток клиента dataOut для передачи серверу не нулевой символ, как раньше, а строку "\r\n", вызывая для объекта dataOut последовательно методы writeChars() и flush().

В цикле до тех пор, пока переменная quit не равна true, сделаем следующие действия:

- чтение целого числа из входного потока клиента dataIn методом readInt();
- печать этого числа методом System.out.println();
- чтение байтов, вводимых пользователем с консоли, в буфер buf методом System.in.read();
- если количество прочитанных байт (выдаваемых методом read()) равно 1 (введен только символ перевода строки), то перейти к следующей итерации цикла;
- удалить незначащие байты массива buf, помещая полученную строку в str (см. приложение ClientDatagram);
- если строка str равна строке «QUIT» (или «quit»), то присвоить переменной quit значение true;
- поместить в выходной поток клиента dataOut для передачи серверу массив бай-

тов buf, вызывая для объекта dataOut последовательно методы write() и flush().

Метод main() приложения ServerSocketApp (модификация метода main() приложения Server)

Добавим описание следующих переменных:

byte buf[]=new byte[512]; // буфер для получаемых данных

String str; StringTokenizer strFull; // для выделения из буфера строки

В цикле до тех пор, пока переменная quit не равна true, сделаем следующие действия:

- чтение массива байт, переданного клиентом, из входного потока сервера dataIn методом read() в массив buf;
- удалить незначащие байты массива buf, помещая полученную строку в str (см. приложение ServerDatagram);
- печать полученной строки методом System.out.println();
- если строка str равна строке «QUIT» (или «quit»), то присвоить переменной quit значение true;
- иначе увеличить счетчик count; передать значение count клиенту, вызывая для объекта dataOut последовательно методы writeInt() и flush().

Задания к лабораторной работе

Задание 1. Проверить и объяснить работу приложений Server-Client, ServerDatagram-ClientDatagram, рассматриваемых в данной главе в качестве примеров и отмеченных курсивом.

Задание 2. Создать приложения ServerSocketApp и ClientSocketApp и объяснить их работу.

Задание 3. Дать ответы на контрольные вопросы.

Контрольные вопросы

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 предназначен для работы с IP-адресами?
6. Как создать объект этого класса для локального и для удаленного узлов?
7. Каким образом задается адрес узла при создании объекта, отвечающего за адрес IP?
8. Какова последовательность действий приложений Java, необходимая для создания канала и передачи данных между клиентским и серверным приложением?
9. Почему приложение должно самостоятельно закрывать все потоки ввода-вывода?
10. Каковы недостатки и преимущества датаграммных сокетов?

11. Что должны сделать приложения для работы с датаграммами?

12. Какие методы применяются для посылки и получения датаграмм?

13. Какой класс отвечает за пакет пересылаемых или получаемых данных? Какую информацию содержит объект этого класса?

14. Как серверное приложение может определить адрес клиентского приложения, приславшего датаграмму?

15. Если адрес сервера неизвестен клиентскому приложению, то как оно может получать его широковещательные пакеты (датаграммы)?

16. Как следует организовать работу сетевых приложений, использующих датаграммы и ожидающих подтверждения получения посланных ими датаграмм?

ЛАБОРАТОРНАЯ РАБОТА № 9

СВЯЗЬ ПО СЕТИ С ПОМОЩЬЮ URL (2 ЧАСА)

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ

Адрес IP позволяет идентифицировать узел, однако его недостаточно для идентификации ресурсов, имеющихся на этом узле, таких, как работающие приложения или файлы. Причина очевидна - на узле, имеющем адрес IP, может существовать много различных ресурсов.

1. Универсальный адрес ресурсов URL

Для ссылки на ресурсы сети Internet применяется так называемый универсальный адрес ресурсов URL (Universal Resource Locator). В общем виде этот адрес выглядит следующим образом:

[protocol://]host[:port][path]

Строка адреса начинается с протокола protocol, который должен быть использован для доступа к ресурсу. Документы HTML, например, передаются с сервера Web удаленным пользователям с помощью протокола HTTP. Файловые серверы в сети Internet работают с протоколом FTP. Наиболее распространенные протоколы перечислены в следующей таблице:

Описание	Протокол
HyperText Transfer Protocol (HTTP)	http://
File Transfer Protocol (FTP)	ftp://
Wide Area Index and Search (WAIS)	wais://

Telnet	telnet://
Usenet (News)	new://
Simple Mail Transfer Protocol (SMTP)	mailto:
Протокол для работы с локальными файлами	file://

Параметр host обязательный. Он должен быть указан как доменный адрес или как адрес IP (в виде четырех десятизначных чисел). Например:

http://www.vvsu.ru

http://194.84.124.12

Необязательный параметр port задает номер порта для работы с сервером. По умолчанию для протокола HTTP используется порт с номером 80, однако для специализированных серверов это может быть и не так.

Номер порта идентифицирует программу, работающую в узле сети TCP/IP и взаимодействующую с другими программами, расположенными на том же или другом узле сети. Например, при разработке программы, передающей данные через сеть TCP/IP с использованием интерфейса сокетов, при создании канала с удаленным компьютером необходимо указать не только адрес IP, но и номер порта, который будет использован для передачи данных. Ниже показано, как нужно указывать в адресе URL номер порта:

http://www.vvsu.ru:8082

Рассмотрим теперь параметр path, определяющий путь к объекту. Если в качестве адреса URL указать навигатору только доменное имя сервера, сервер перешлет навигатору свою главную страницу. Имя файла этой страницы зависит от сервера. Большинство серверов на базе операционной системы UNIX посылают по умолчанию файл документа с именем index.html. Сервер Microsoft Information Server может использовать для этой цели имя default.htm или любое другое, определенное при установки сервера.

Для ссылки на конкретный документ HTML или на файл любого другого объекта необходимо указать в адресе URL его путь, включающий имя файла, например:

http://www.vvsu.ru/cts/rus_win/index.html

http://www.vvsu.ru/cts/teachers/arhipova/pictures/clock.gif

Корневой каталог сервера Web обозначается символов /. Если путь вообще не задан по умолчанию используется корневой каталог.

2. Класс java.net.URL в библиотеке классов Java

Для работы с ресурсами, заданными своими адресами URL, в библиотеке классов Java имеется очень удобный и мощный класс с названием URL. Инкапсулируя в себе достаточно сложные процедуры сетевого программирования, класс URL предоставляет небольшой

набор простых в использовании конструкторов и методов. Работать с этим классом могут как автономные приложения, так и апплеты

Конструкторы класса URL

Класс URL содержит четыре конструктора. Первый из них создает объект URL в виде сетевого ресурса, адрес URL которого передается конструктору в виде текстовой строки через параметр spec.

Public URL(String spec);

В процессе создания объекта проверяется заданный адрес URL, а также наличие указанного в нем ресурса. Если адрес указан неверно или заданный в нем ресурс отсутствует, возникает исключение MalformedURLException. Это же исключение возникает при попытке использовать протокол, с которым данная система не может работать.

Второй вариант конструктора допускает раздельное указание протокола, адреса узла, номера порта, а также имени файла:

public URL(String protocol, String host, int port, String file);

Третий вариант предполагает использование номера порта, принятого по умолчанию (для протокола HTTP это порт с номером 80):

public URL(String protocol, String host, String file);

Четвертый вариант конструктора допускает указание контекста адреса URL и строки адреса URL:

public URL(URL context, String spec);

Этот конструктор создает URL каталога или файла по его пути spec относительно заданной URL-ссылки context.

Хотя фирма Sun разработала поддержку URL для очень ограниченного числа протоколов – DOC, FILE и HTTP (протокол FILE URL применяется для локальных файлов, а DOC URL использован в броузере Hotjava), следует отметить, что в классе URL имеется возможность создания поддержки других протоколов.

Некоторые методы класса URL

С помощью метода getHost() можно определить имя узла, соответствующего данному объекту URL. Метод getFile() позволяет получить информацию о файле, связанном с данным объектом URL. Метод getPort() предназначен для определения номера порта, на котором выполняется связь для объекта URL. С помощь метода getProtocol() можно определить протокол, с использованием которого установлено соединение с ресурсом, заданным объектом URL.

С помощью метода sameFile() можно определить, ссылаются ли два объекта класса URL на один и тот же ресурс. Для определения идентичности двух адресов можно также воспользоваться методом equals().

Для доступа к ресурсам и их содержимому используются методы openStream(), getContent(), openConnection().

3. Использование класса java.net.URL

После реализации класса URL часто бывает необходимо получить доступ к ресурсам, на которые он указывает. Класс URL предлагает для этого три основных метода: openStream(); getContent(); openConnection(). Рассмотрим эти методы подробнее.

3.1 Чтение из потока класса InputStream, полученного от объекта класса URL

Метод openStream() позволяет создать входной поток класса InputStream для чтения файла ресурса, связанного с созданным объектом класса URL. Для выполнения операции чтения из созданного таким образом потока можно использовать любую разновидность метода read(), определенных в классе InputStream. После использования потока его следует закрыть методом close() класса InputStream.

Пару методов (openStream() из класса URL и read() класса InputStream) можно применить для решения задачи получения содержимого двоичного или текстового файла (например, HTML-файл), хранящегося в одном из каталогов сервера Web. После этого обычное приложение или апплет может выполнить локальную обработку полученного файла на компьютере удаленного пользователя.

Например, рассмотрим фрагмент апплета, в котором создается URL к файлу, расположенному на WWW-сервере апплета (откуда апплет загружен на удаленный компьютер), затем открывается поток, связанный с этим файлом, из которого потом читается содержимое HTML-файла:

```
URL myUrl;
try
{       myUrl=new URL(getCodeBase(),"index.html");
} catch(MalformedURLException e) { /* обработка исключения */ }
try
{       InputStream InStream=myUrl.openStream();
        // чтение данных из потока InStream
        InStream.close();
} catch(IOException e) { /* обработка исключения */ }
```

Приведем в качестве примера приложение *UrlOpenStream (пример 1)*, в котором при помощи входного потока считывается html-файл из каталога удаленного сервера Web и его содержимое выводится на консоль:

```
/*------------ Пример 1. Файл UrlOpenStream.java ------------*/
import java.net.*;
import java.io.*;
class UrlOpenStream
{  public static void main(String args[])
   {        URL Url;
            try
            {/// создание URL файла и открытие
                  // входного потока, связанного с этим файлом
                  Url=new URL("http://www.microsoft.com");
                  InputStream InStream=Url.openStream();
                  // чтение данных из потока InStream
                  int b;
                  while ((b=InStream.read())!=-1)
                  {        System.out.print(""+(char)b);
                  }
                  InStream.close(); // закрытие потока
            }
            catch(Exception e) { System.out.println(e.toString()); }
   }
}
```

3.2 Получение содержимого файла, связанного с объектом класса URL

Метод getContent() открывает поток к ресурсу в точности также, как это делает метод openStream(), но затем пытается определить MIME потока (тип файла) и конвертировать поток в объект Java. Зная тип MIME потока данных, URL может передать поток данных методу, созданному для работы именно с этим типом данных. Этот метод должен выдать данные, инкапсулированные в соответствующем типе объекта Java. Например, если создан URL, указывающий на изображение в формате GIF, метод getContent() должен понять, что поток относится к типу MIME «image/gif», и вернуть экземпляр класса Image. Объект Image будет содержать копию GIF-картинки. Для того, чтобы метод getContent() вернул объект в соответствии с MIME, необходимо определить для этого MIME собственный класс ContentHandler,

который в конечном счете и обрабатывает данные ресурса, когда вызывается метод getContent() класса URL или URLConnection.

Практически, можно использовать метод getContent() для получения текстовых файлов, расположенных в сетевых каталогах. К сожалению, метод getContent() непригоден для получения документов HTML, так как для данного ресурса не определен обработчик содержимого, предназначенный для создания объекта. Метод getContent не способен создать объект из чего-либо другого, кроме как из текстового файла.

Приведем пример использования метода getContent() в приложении. Сначала создается объект URL, потом вызывается метод getContent(), чтобы восстановить ресурс в объекте Java, а затем применяется операция instanceof для определения того, какой тип объекта возвращен:

```
URL myUrl; Object obj;
try
{       myUrl=new URL
                ("http://www.vvsu.ru/cts/teachers/arhipova/t.txt");
} catch(MalformedURLException e) { /* обработка исключения */ }
try
{       obj=myUrl.getContent();
} catch(IOException e) { /* обработка исключения */ }
if(obj instanceof String) { /* действия, если строка */ }
else { /* тип неизвестен, действия по умолчанию */ }
```

Рассмотрим в качестве примера апплет *UrlGetContent (пример 2)*, в котором получается содержимое файла из каталога сервера Web (с которого загружен апплет). В случае, если файл содержит простой текст, то содержимое файла выводится в текстовую область, введенную в апплет:

```
/*---------- Пример 2. Файл UrlGetContent.java ----------*/
import java.applet.*;
import java.awt.*;
import java.net.*;
public class UrlGetContent extends Applet
{  String S; Object obj;
   public void init()
   {       resize(600, 400);
           // создание текстовой области и введение ее в апплет
           TextArea text=new TextArea(30,80);
```

```
        add(text);
        URL Url;
        try
        {       // создание URL файла с сервера апплета
                // и получение содержимого этого файла
                Url=new URL(getCodeBase(),"UrlGetContent.java");
                obj=Url.getContent();
                // проверка, является ли содержимое текстом
                if(obj instanceof String) { S=(String)obj; }
                else { S="Unknown object"; }
        }
        catch(Exception e) { S=e.toString(); }
        // вывод либо содержимого файла, либо предупреждения
        text.setText(S);
    }
    public void paint(Graphics g){}
}
```

4. Соединение с помощью объекта класса URLConnection

Если создается приложение, которое позволяет читать из каталога сервера WWW текстовые или двоичные файлы, то можно создать поток методом openStream() или получить содержимое файла методом getContent(). Однако есть и другая возможность.

Чтение файла

Сначала можно создать канал как объект класса URLConnection, вызвав метод openConnection(),определенный в класса URL, а затем можно создать для этого канала входной поток, воспользовавшись методом getInputStream, определенным в классе URLConnection, или получить содержимое файла методом getContent() этого же класса, как это сделано в следующих фрагментах апплетов:

```
    URL myUrl; URLConnection myUrlConnection;
    try
    {       myUrl=new URL(getCodeBase(),"index.html");
            myUrlConnection=myUrl.openConnection();
    } catch(MalformedURLException e) { /* обработка исключения */ }
    try
    {       InputStream InStream=myUrlConnection.getInputStream();
            // чтение данных из потока InStream
```

```
            InStream.close();
    } catch(IOException e) { /* обработка исключения */ }
или
    URL myUrl; URLConnection myUrlConnection; Object obj;
    try
    {       myUrl=new URL(getCodeBase(),"index.html");
            myUrlConnection=myUrl.openConnection();
    } catch(MalformedURLException e) { /* обработка исключения */ }
    try
    {       obj=myUrlConnection.getContent();
    } catch(IOException e) { /* обработка исключения */ }
    if(obj instanceof String) { /* действия, если строка */ }
    else { /* тип неизвестен, действия по умолчанию */ }
```

Такая методика позволяет определить или установить перед созданием потока некоторые характеристики канала, а также получить дополнительную информацию о ресурсе.

Рассмотрим информационные методы класса URLConnection. Метод getContentLength() возвращает размер ресурса в байтах. Тип MIME ресурса (например, «image/gif») можно определить при помощи метода getContentType(). Дату посылки ресурса возвращает метод getDate(). При помощи ряда методов getHeaderField...() можно прочитать различную информацию из заголовка файла.

Приведем в качестве примера приложение *UrlOpenConnection (пример 3)*, в котором открывается соединение с файлом каталога удаленного сервера и информация об этом файле выводится на консоль:

```
/*---------- Пример 3. Файл UrlOpenConnection.java ----------*/
import java.net.*;
public class UrlOpenConnection
{  public static void main(String args[])
   {       URL Url; URLConnection UrlConnection;
           try
           {       // создание URL файла и открытие соединения с ним
                   Url=new URL
                   ("http://www.vvsu.ru/cts/teachers/arhipova/pictures/clock.gif");
                   UrlConnection=Url.openConnection();
                   // информация о файле
                   System.out.println("Size:"+UrlConnection.getContentLength());
```

```
                System.out.println("Type:"+UrlConnection.getContentType());
                System.out.println("Date:"+UrlConnection.getDate());
            }
            catch(Exception e) { System.out.println(e.toString()); }
        }
    }
```

Запись в файл

После создания канала как объекта класса URLConnection вызовом метода openConnection(),определенного в класса URL, можно не только создавать для этого канала входной поток, воспользовавшись методом getInputStream(), определенным в классе URLConnection, или получить содержимое файла методом getContent() этого же класса, но и создавать выходной поток для записи в этот файл. Для создания выходного потока используется метод getOutputStream() класса URLConnection:

```
    try
    {       InputStream InStream=myUrlConnection.getInputStream();
            // чтение данных из потока InStream
            InStream.close(); // закрытие потока
            OutputStream OutStream=myUrlConnection.getOutputStream();
            // запись данных в поток
            OutStream.flush();
            OutStream.close();
    }
    catch(Exception e) { /* обработка исключения */ }
```

Если попытаться вызвать методы getInputStream() и getOutputStream() для такого типа URL, который не поддерживает эти методы , то возникает исключение UnknownServiceException.

Настройка класса URLConnection

Для конфигурации класса URLConnection существуют различные варианты настройки (элементам настройки присваиваются значения по умолчанию). Некоторые из этих значений по умолчанию можно изменить.

Так, например, методом setDoInput() задается флаг doIinput, который показывает, поддерживает ли данный URLConnection входные данные (по умолчанию значение этого флага - true). Метод getDoInput() возвращает установленное значение флага doInput.

Метод setDoOutput() устанавливается флаг doOutput, который показывает, поддерживает ли данный URLConnection выходные данные (по умолчанию значение этого флага - false). Метод getDoOutput() возвращает установленное значение флага doOutput.

5. Приложение Diagram

Применим на практике технологию передачи файлов из каталога WWW-сервера в аплет.

Задание. Создать апплет *Diagram,* который читает с сервера WWW файл data.txt с исходными данными для построения круговой диаграммы. Содержимым файла является текстовая строка вида «10,20.3,13.1,6.9,35,14.7», где каждое число является долей чего-либо в процентах, причем сумма этих чисел не должна превышать 100.

Содержимое файл на сервере WWW может периодически изменяться (независимо от работы апплета), поэтому в апплете требуется предусмотреть процесс обновления информации и ее перерисовки при нажатии на левую клавишу мыши в области окна апплета.

Методические указания. Апплет должен быть создан на основе шаблонов, содержащихся в Приложении 3 (или при помощи системы Java Applet Wizard).

Объявление элементов класса апплета.

В классе апплета объявим следующие элементы - ссылки на объекты классов:

```
URL url; // URL сервера, откуда загружен апплет
String content; // содержимое ресурса сервера - файла с данными
String strErr; int err=0; // строка с сообщением и признак ошибки
int x,y,w; // область диаграммы
double beginPos, angle; // начальный угол сектора и его величина
int rColor,gColor,bColor; // компоненты цвета сектора
double unit=3.60; // величина сектора для 1% равна 360/100 градусов
```

Инициализация апплета (метод init()).

В методе init() апплета создадим объект url класса URL, который отвечает за URL-адрес сервера WWW, откуда был загружен апплет. Затем вызовем метод getFileContent(), который осуществляет считывание данных из файла.

Все вышеописанные действия следует заключить в блок try-catch для обработки возможных исключительных ситуаций:

```
try
{           // получение URL сервера апплета и вызов метода getFileContent
}
catch(MalformedURLException e)
```

```
{           // обработка исключения
}
```

При обработке исключения установим значение err равным 1, переменной strErr присвоим строку "Type of error: MalformedURLException", а затем перерисуем окно апплета вызовом метода repaint().

Считывание содержимого файла с WWW-сервера (метод getFileContent()).

Создадим в классе апплета метод getFileContent(), который имеет следующее описание:

```
void getFileContent();
```

В этом методе в блоке try-catch получим содержимое файла, обработаем исключительную ситуацию, а затем перерисуем окно апплета вызовом метода repaint()

```
try
{           //получение содержимого файла и проверка его типа
}
catch(Exception e)
{           // обработка исключения
}
repaint();
```

Для получения содержимого файла вызовем для объекта url метод getContent(), присваивая возвращаемое этим методом значение объекту obj класса Object. Затем проверим, является ли obj экземпляром класса String. Если да, то: установим значение err равным 0; переменной strErr присвоим строку ""; переменной content присвоим значение obj. Если нет, то: установим значение err равным 2; переменной strErr присвоим строку "Type of error: Content of file isn't string".

При обработке исключения установим значение err равным 3, а переменной strErr присвоим строку"Type of error: getContent exception".

Отрисовка содержимого окна апплета (метод paint()).

В методе paint() проверим значение err. Если оно не равно 0, то выведем в окно апплета строку strErr и прекратим работу метода оператором return.

Если ошибок нет, то нарисуем по считанным данным круговую диаграмму.

Сначала определим размеры окна апплета методом size(), присваивая возвращаемое значение переменной dim класса Dimension. Затем зададим координаты (x, y) левого верхнего угла области диаграммы, а также размер w этой области по горизонтали и вертикали, учитывая размер окна апплета dim.

Установим методом setColor() цвет Color.black для контекста g и нарисуем рамку окна апплета методом drawRect(), используя объект dim. Далее методом drawOval() выведем круг, вписанный в прямоугольник с началом в точке (x, y) высотой и шириной w.

Теперь приступим в выводу секторов диаграммы. Для этого создадим разборщик строки исходных данных:

```
StringTokenizer ct=new StringTokenizer(content,",\r\n");
```

Далее в цикле по всем данным (числам) из строки определяем угол angle, соответствующий считанной величине в процентах и рисуем соответствующий сектор:

```
// цикл по всем значениям из строки ct
beginPos=0; // начальный угол для первого значения из строки
while(ct.hasMoreElements())
{       String word=(String)ct.nextElement(); // получить слово
        Double value=new Double(word); // преобразовать в число
        double percent=value.doubleValue(); // преобразовать в тип double
        angle=percent*unit; // angle - величина в градусах
        // отрисовка очередногосектора величиной angle
        // градусов, начиная от начального угла beginAngle,
        // изменение beginAngle и его проверка выхода за 360 градусов
}
```

В этом цикле для вывода каждого сектора получим случайные значения для компонент rColor, gColor, bColor цвета сектора, например:

```
rColor=(int)(255.*Math.random());
```

Затем установим методом setColor() цвет сектора (new Color(rColor,gColor,bColor)) и с помощью метода fillArc() нарисуем сектор круга, вписанного в прямоугольник с началом в точке (x, y) высотой и шириной w, начиная от угла beginPos величиной angle.

Потом прибавим к величине beginPos значение angle, теперь beginPos является начальным углом для следующего сектора. Проверим, чтобы новое значение beginPos не превышало 360. Если оно превышает это значение, то значит сумма считанных чисел больше 100%, поэтому установим значение err равным 4, переменной strErr присвоим строку "Type of error: summa greater than 100%", а затем перерисуем окно апплета вызовом метода repaint().

Обработка нажатия левой клавишей мыши в области окна апплета (метод mouseDown()).

В этом методе для обновления информации о данных перечитаем файл с сервера WWW и нарисуем новую диаграмму вызовом метода getFileContent().

Задания к лабораторной работе

Задание 1. Проверить и объяснить работу приложений UrlOpenStream, UrlGetContent и UrlOpenConnection**,** рассматриваемых в данной главе в качестве примеров и отмеченных курсивом.

Задание 2. Создать приложение Diagram и объяснить его работу.

Задание 3. Дать ответы на контрольные вопросы.

Контрольные вопросы

67. Что такое URL, каков его формат?
68. Какой класс применяется для работы с ресурсами узла сети?
69. Как организовать чтение из потока, полученного от объекта класса URL?
70. Как получить содержимое файла, связанного с объектом класса URL?
71. Как организовать соединение узлов сети с помощью объекта URL?
72. Что можно сделать с помощью объекта класса URLConnection?
73. Как и какую информацию можно получить о ресурсе удаленного узла?
74. Как организовать входной и выходной потоки для канала, являющегося объектом класса URLConnection?

ЛАБОРАТОРНАЯ РАБОТА № 10

СОЗДАНИЕ И ИСПОЛЬЗОВАНИЕ СЕРВЛЕТОВ (2 ЧАСА)

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ

Получить JSDK 2.0 (Windows)

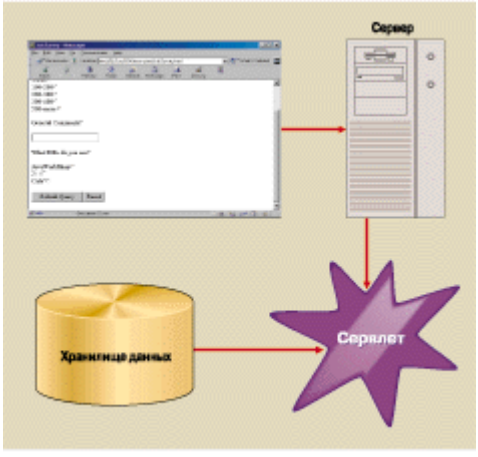
Замечание. Установить в autoexec.bat:

set CLASSPATH=.;c:;C:\Windows\java\classes;C:\Windows\java\trustlib;**c:\jsdk2.0\lib\jsdk.jar**;

Получить JSDK 2.0 (Solariss)

Если вам необходимо расширить функциональные возможности сервера Web, можно написать CGI-сценарий. А можно разработать и установить расширение на основе ISAPI. Ознакомьтесь с несколькими классами из *набора Java Servlet Development Kit (JSDK)*.

С помощью JSDK можно разрабатывать так называемые сервлеты - специальные программы, выполняющиеся в рамках серверов, способные обрабатывать сложные клиентские запросы и динамически генерировать ответы на них. Когда на объект, содержащий сервлет, делается запрос, сервлет выполняется, и его вывод посылается клиенту. Сервлеты могут также быть настроены на взаимодействие с другими программами сервера. Тогда сервлет может выбирать: либо послать сгенерированную информацию клиенту, либо, например, передать сохранить ее в файле на сервере, либо же сделать и то и другое.



Примером использования сервлетов может служить расширение, читающее запрос на языке SQL, анализирующее его и делающее выборку данных из

хранилища, а также пересылающее клиенту HTML-страницу, сгенерированную автоматически на основе полученных данных (см. рис. справа).

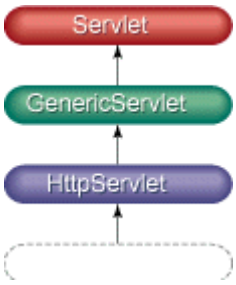
Преимуществом сервлетов можно считать то, что они пишутся на объектно-ориентированном языке высокого уровня, к которому имеется масса дополнений и программных интерфейсов, а это значительно увеличивает область применения расширений на основе сервлетов. Если у вас есть готовая серверная логика, написанная на Java, то превратить ее в сервлет легче легкого.

Вся библиотека классов языка Java у вас в руках! Подключать и настраивать сервлеты также несложно.

Наибольшее распространение получили сервлеты, обрабатывающие запросы по протоколу HTTP - стандартному протоколу обмена данными WWW.

Рассмотрим наиболее часто используемые программистами классы и методы JSDK. Однако, если ваши задачи сложны и требуют тонкого подхода, внимательно изучите справочник по API, имеющийся в составе Java Servlet Development Kit. Последний можно бесплатно загрузить с сервера *http://www.javasoft.com*.

1. Как устроен сервлет



Главным родителем всех сервлетов является интерфейс Servlet. Следующий уровень иерархии - абстрактный класс GenericServlet, частично реализующий методы Servlet и служащий базой для создания экзотических сервлетов. Скорее всего, вам никогда не доведется обращаться к нему. Разумнее будет воспользоваться в качестве базового абстрактным классом HttpServlet. Таким образом**,** главное, что нужно сделать для получения работоспособного сервлета, - это создать класс, наследующий HttpServlet. А уже потом следует заняться написанием логики работы. На рис. слева представлена иерархическая диаграмма, показывающая взаимосвязь классов в Java Servlet Development Kit. Пунктирной рамкой отмечено место, отведенное сервлету пользователя.

Базовая часть классов JSDK помещена в пакет javax.servlet. Однако класс HttpServlet и все, что с ним связано, располагаются на один уровень ниже в пакете javax.servlet.http. Это следует учитывать при разработке. Также не забывайте, что основные сервлетообразующие классы находятся в архивном файле jsdk.jar, поэтому этот архив должен упоминаться в переменной среды CLASSPATH или же должен быть задан вручную в параметре командной строки -classpath при запуске компилятора javac.

Последовательность работы сервлета проста: инициализация, обслуживание запросов и завершение существования. В каждой из этих фаз сервер, на котором выполняется сервлет, вызывает соответствующий метод интерфейса Servlet.

Первым вызывается метод init. Он дает сервлету возможность инициализировать важные данные и подготовиться для обработки запросов. Чаще всего в этом методе программисты помещают исходный текст, кэширующий данные фазы инициализации и промежуточные результаты различных вычислений.

Заметим, что сервлет не нуждается в отдельной загрузке в память каждый раз, когда происходит запрос на него. Будучи помещенным в память при обработке некоторого запроса ,и, обработав этот запрос , сервлет еще некоторое время (оно зависит от настроек Web-сервера) не выгружается из памяти, что помогает ему быстрее обрабатывать другие запросы на него. Только если в течении этого времени на сервлет не было сделано ни одного запроса, он выгружается из оперативной памяти.

После этого сервер находится в ожидании запросов от клиентов. Появившийся запрос немедленно преобразуется в вызов метода service сервлета, а все параметры запроса упаковываются в объект класса ServletRequest, который передается в качестве первого параметра метода service. Второй параметр метода - объект класса ServletResponse. Туда упаковываются выходные данные в процессе формирования ответа клиенту. Каждый новый запрос приводит к новому вызову метода service. В соответствии со спецификацией JSDK, метод service должен уметь обрабатывать сразу несколько запросов, т. е. быть синхронизирован для выполнения в многопоточных средах. Это особенно критично, когда в нем происходит обращение к разделяемым данным. Если же нужно избежать множественных запросов, сервлет должен реализовать интерфейс SingleThreadModel. Последний не содержит ни одного метода и служит лишь меткой, говорящей серверу об однопоточной природе сервлета. При обращении к такому сервлету каждый новый запрос будет стопориться в очереди и ожидать, пока не завершится обработка предыдущего запроса.

Завершив выполнение сервлета, сервер вызывает его метод destroy, предполагая, что в нем сервлет "почистит" занятые ранее ресурсы. Программируя этот метод, разработчик должен помнить, что в многопоточных средах сервер может вызвать метод destroy в любой момент, даже когда еще не завершился метод service. Поэтому на передний план выходит обязательная синхронизация доступа к разделяемым ресурсам.

Интерфейсом Servlet предусмотрена реализация еще двух методов: getServletConfig и getServletInfo. Первый возвращает объект типа ServletConfig, содержащий параметры конфигурации сервлета, а второй - строку, кратко описывающую назначение сервлета, и прочую полезную информацию.

Сервлет HttpServlet, отвечающий за обработку запросов HTTP, устроен несколько сложнее. Он уже имеет реализованный метод service, служащий диспетчером для других методов, каждый из которых обрабатывает методы доступа к ресурсам. В спецификации HTML

определены следующие методы: GET, HEAD, POST, PUT, DELETE, OPTIONS и TRACE. Наиболее часто употребляются GET - универсальный запрос ресурса по его универсальному адресу (URL) - и POST, с помощью которого на сервер передаются данные, введенные пользователем в поля интерактивных Web-страниц. Полное описание протокола HTTP 1.1 можно найти в Internet по адресу *http://info.internet.isi.edu/in-notes/rfc/files/ rfc2068.txt*.

Возвратимся к методу service HttpServlet. В его задачу входит анализ полученного через запрос метода доступа к ресурсам и вызов соответствующего метода, имя которого сходно с названием метода доступа к ресурсам, но в начале имени добавляется префикс do: doGet, doHead, doPost, doPut, doDelete, doOptions и doTrace. От разработчика же требуется переопределить нужный метод (чаще всего это doGet), разместив в нем функциональную логику.

2. Вспомогательные классы

Чтобы успешно использовать сервлеты, необходимо ознакомиться с рядом вспомогательных классов. Для начала узнаем, какие методы предлагает интерфейс ServletRequest, передающий сервлету запрос от клиента. Конечно же, запрос поступает не в том виде, в котором он приходит на сервер. Поток данных от клиента сортируется и упаковывается. Далее, вызывая методы интерфейса ServerRequest, можно получать определенный тип данных, посланных клиентом. Так, метод getCharacterEncoding определяет символьную кодировку запроса, а методы getContentType и getProtocol - MIME-тип пришедшего запроса, а также название и версию протокола соответственно. Информацию об имени сервера, принявшего запрос, и порте, на котором запрос был "услышан" сервером, выдают методы getServerName и getServerPort. Интересные данные можно узнать и о клиенте, от имени которого пришел запрос. Его IP-адрес возвращается методом getRemoteAddr, а его имя - методом getRemoteHost.

Если вас интересует прямой доступ к содержимому полученного запроса, самый надежный способ получить его - вызвать метод getInputStream или getReader. Первый возвращает ссылку на объект класса ServletInputStream, а второй - на BufferedReader. После этого можно читать любой байт из полученного запроса, используя технику работы с потоками Java.

Если, обращаясь к серверу, клиент помимо универсального адреса задал параметры, сервлету может понадобиться узнать их значение. Примером может служить электронная анкета, выполненная в виде Web-страницы с формой ввода, значения полей и кнопок которой автоматически преобразуются в параметры URL. Три специальных метода в интерфейсе ServletRequest занимаются разбором параметров и "выдачей на-гора" их значений. Первый из них, getParameter, возвращает значение параметра по его имени или null, если параметра с

таким именем нет. Похожий метод, getParameterValues, возвращает массив строк, если задан сложный параметр, скажем, значения полей формы. И еще один метод, getParameterNames, возвращает энумератор, позволяющий узнать имена всех присланных параметров.

В классе HttpServletRequest имеются различные дополнительные методы, обеспечивающие программисту доступ к деталям протокола HTTP. Так, вы можете запросить массив cookies, полученный с запросом, используя метод getCookies. Узнать о методе доступа к ресурсам, на основе которого построен запрос, можно с помощью вызова getMethod. Строку запроса HTTP можно получить методом getQueryString. Даже имя пользователя, выполнившего запрос, не укроется от сервлета, если применить метод getRemoteUser.

Это, разумеется, лишь малая толика тех возможностей, которыми обладают вышеупомянутые классы ServletRequest и HttpServletRequest. Поэтому следует внимательно прочитать документацию по Servlet API.

Генерируемые сервлетами данные пересылаются серверу-контейнеру с помощью объектов, наследующих интерфейс ServletResponse, а сервер, в свою очередь, пересылает ответ клиенту, инициировавшему запрос. У интерфейса ServletResponse всего несколько методов, причем полезными оказываются не все. Чаще всего приходится задавать MIME-тип генерируемых данных методом setContentType и находить ссылки на потоки вывода двумя другими методами: getOutputStream возвращает ссылку на поток ServletOutputStream, а метод getWriter вернет ссылку на поток типа PrintWriter. Вы увидите, как ими пользоваться, когда мы будем рассматривать практический пример.

В классе HttpServletResponse, реализующем интерфейс ServletResponse, обнаруживаются еще несколько полезных методов. Например, вы можете переслать cookie на клиентскую станцию, вызвав метод addCookie. О возникших ошибках сообщается вызовом sendError, которому в качестве параметра передается код ошибки и при необходимости текстовое сообщение. Кроме того, по мере надобности в заголовок ответа можно добавлять параметры, для чего служит метод setDateHeader.

Ранее уже упоминался метод getServletConfig, но не было сказано об объекте ServletConfig, у которого имеются очень полезные методы. Инсталлируя сервлет на сервере, вы можете задавать параметры инициализации, о которых будет сказано в следующей части. Имена этих параметров можно получить через энумератор, возвращаемый методом getInitParameterNames. Значение же конкретного параметра получают вызовом getInitParameter. Также важен метод получения контекста сервлета getServletContext, через обращение к которому можно узнать много полезного о среде, в которой запущен и выполняется сервлет.

Контекст выполнения сервлета интересен тем, что дает примитивные средства для общения с сервером. Скажем, для выполнения задачи требуется узнать MIME-тип того или иного файла. Всегда пожалуйста, вызовите метод getMimeType контекста. Или нужно узнать истинный маршрут файла относительно каталога, в котором сервер хранит документы. Тоже нет проблем, метод getRealPath к вашим услугам. Информация же о самом сервере предоставляется по вызову getServerInfo().

Отдельно стоят метод, загружающий сервлет по имени getServlet, и метод, возвращающий энумератор с именами всех установленных сервлетов getServletNames. Однако не рекомендуется пользоваться ими, так как в будущей версии Java Servlet Development Kit их может уже и не быть.

И напоследок самый важный метод log. С его помощью нужные текстовые данные пишутся в протокол работы сервлетов.

3. Запуск и настройка сервлетов

Настало время остановиться на деталях настройки и запуска сервлетов. Для этого следует поговорить о файле свойств. Имя этого файла — servlet.properties. В нем в виде пар «ключ—значение» хранятся свойства, используемые для конфигурации, создания и инициализации сервлетов. Изначально для любого из сервлетов предопределено два свойства. Первое, servlet.<имя сервлета>.code, определяет имя сервлета и ставит его в соответствие двоичному class-файлу сервлета. Например, если вы скомпилировали класс сервлета с именем MyServletClassName, то получите в результате компиляции файл с именем MyServletClassName. class и можете присвоить ему краткое имя, скажем, myservlet, следующим образом:

servlet.myservlet.code=MyServletClassName

Теперь, когда вы обратитесь к сервлету с именем myservlet, сервер найдет эту строку в файле свойств и, опираясь на найденное значение, загрузит класс MyServletClassName, инициализирует его и передаст ему ваш запрос. Следует помнить о том, что имя class-файла должно задаваться полностью, включая имя пакета, в котором определен класс. Второе свойство, servlet.<имя сервлета>.initargs, определяет передаваемые сервлету параметры инициализации. Значения такого рода параметров могут быть получены сервлетом методом getInitParameter. Если параметров несколько, они отделяются друг от друга запятыми. Пример задания параметров:

*servlet.myservlet.initargs= *

someParameterName1=someValue, someParameterName2=otherValue

Файл servlet.properties помещают в определенный каталог на сервере, где хранятся class-файлы. Заметим, однако, что разные серверы допускают альтернативное местоположение файла свойств (по настройке администратора).

Теперь поговорим об утилите servletrunner, которая по сути является простейшим Web-сервером, специально предназначенным для работы с сервлетами. После запуска он «слушает» порт 8080, и если произошел запрос, то, обратившись к сервлету, servletrunner получает от него ответ и пересылает последний программе-клиенту. Командная строка servletrunner имеет различные опции, но полезными могут оказаться лишь следующие:

p port – «прослушиваемый» в ожидании запроса порт;

t timeout – время тайм-аута в мс;

d dir – каталог, где лежат сервлеты;

r root – корневой каталог, в котором хранятся документы;

s filename – альтернативное имя файла свойств servlet property file name

v – отображать выводимые в стандартные потоки данные.

Никаких дополнительных настроек вам не потребуется. Просто запустите утилиту servletrunner и обращайтесь к ней с помощью браузера или другой клиентской программы. Главное, чтобы servletrunner могла найти ваши сервлеты.

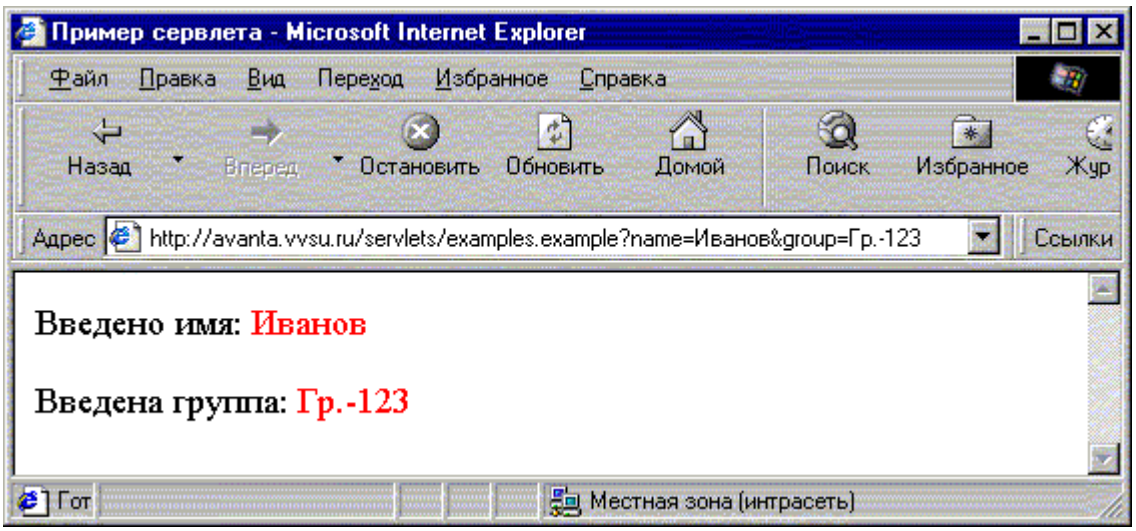
Теперь вкратце о том, как обратиться к сервлету из Web-браузера. По умолчанию адрес URL для отправки запроса состоит из нескольких частей: имени компьютера, номера порта, каталога servlet, имени сервлета и списка параметров. К примеру, обратиться к сервлету myservlet, находящемуся на локальном компьютере, можно так:

http://localhost:8080/servlet/myservlet?param1=somevalue&param2=othervalue

Напомним, что, тоже по умолчанию, утилита servletrunner (и некоторые серверы Web) «слушает» запросы к порту с номером 8080. Приведенный выше запрос обращается к сервлету myservlet, передавая ему параметр param со значением somevalue. Это весьма полезно, когда нужно задавать параметры запроса «на лету» (в отличие от параметров инициализации).

4. Сервлет example, принимающий параметры

Чтобы все рассказанное о сервлетах не осталось для вас пустыми словами, создадим собственный сервлет *example* (пример 1), который будет выводить в окне браузера передаваемые сервлету имена и группы студентов или сообщение о том, что данный параметр не введен (итоги работы сервлета см. рис. ниже):



Приведем сначала листинг исходного текста сервлета, а затем рассмотрим его более подробно (мы не станем детально описывать теги HTML, которые сервлет вставляет в генерируемую страницу, – обратитесь самостоятельно к *документации по этому языку*).

/*---------- Пример 1. Файл example.java ----------*/

package examples; // пакет для сервлета-примера

// способы вызова клиентом класса example.example через строку адреса окна браузера

// http://avanta.vvsu.ru/servlets/examples.example?name=Имя&group=Группа

// http://avanta.vvsu.ru/servlets/examples.example?name=Имя

// http://avanta.vvsu.ru/servlets/examples.example?group=Группа

// http://avanta.vvsu.ru/servlets/examples.example

// предполагается, что сервлет находится в директории сервлетов на WWW-сервере avanta.vvsu.ru

import java.io.;*

import javax.servlet.;*

import javax.servlet.http.;*

// класс example пакета examples

public class example extends HttpServlet // класс example пакета examples

{

public void doGet (HttpServletRequest request,HttpServletResponse response) throws ServletException, IOException

{

// получение значения параметра имени, передаваемое через параметры

String NAME=request.getParameter("name");

```
if( NAME==null) NAME="Имя не задано"; // параметр отсутствует в списке
        else
{
// если в параметре возможны символы кириллицы, то необходимо преобразовать значение параметра
NAME=new String(NAME.getBytes(),"windows-1251");
}
// получение значения параметра группы, передаваемое через параметры
String GROUP=request.getParameter("group");
if(GROUP==null) GROUP="Группа не задана"; // параметр отсутствует в списке
        else
{
// если в параметре возможны символы кириллицы, то необходимо преобразовать значение параметра
GROUP=new String(GROUP.getBytes(),"windows-1251");
}
// передача типа генерируемого сервлетом содержимого
response.setContentType("text/html; charset=windows-1251");
// получение выходного потока для вывода генерируемого содержимого
Writer out=new BufferedWriter(new
            OutputStreamWriter(response.getOutputStream(),"Cp1251"));
// начало формирования результата в формате html-файла
out.write("<HTML>\n");  // html-тег начала файла
out.write("<HEAD><TITLE>Пример  сервлета</TITLE></HEAD>\n");  // html-тег заголовка файла
out.write("<body>\n"); // html-тег начала содержимого
// формирование содержимого
out.write("<p>\n"); // html-тег начала абзаца
out.write("<font size=\"+1\" color=\"Black\">Введено имя: </font>\n"); // вывод простого текста некоторым шрифтом
out.write("<font size=\"+1\" color=\"Red\">"+NAME+"</font>\n");  // вывод простого текста некоторым шрифтом
out.write("</p>\n"); // html-тег завершения абзаца
```

```
out.write("<p>\n"); // html-тег начала абзаца
out.write("<font size=\"+1\" color=\"Black\">Введена группа: </font>\n"); // вывод простого текста некоторым шрифтом
out.write("<font size=\"+1\" color=\"Red\">"+GROUP+"</font>\n"); // вывод простого текста некоторым шрифтом
out.write("</p>\n"); // html-тег завершения абзаца
// завершение формирования результата
out.write("</body>\n"); // html-тег завершения содержимого
out.write("</html>\n"); // html-тег завершения файла
// закрытие поток вывода
out.flush(); out.close();
return;
}
}
/*--------------------------------------------------------------------------------------*/
```

Начнем с того, что опишем класс сервлета, наследующий HttpServlet, и импортируем необходимые классы. После чего займемся методом doGet – главной частью сервлета:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class example extends HttpServlet
{
public void doGet (HttpServletRequest request,HttpServletResponse response) throws ServletException, IOException
{
```

Следующий блок занимается разбором параметров, переданных сервлету. Отметим, что если в параметре возможны символы кириллицы, то необходимо преобразовать значение параметра, создавая новую строку на базе старой и используя необходимую кодировку:

```
String NAME=request.getParameter("name");
if( NAME==null) NAME="Имя не задано";
        else                    {
NAME=new String(NAME.getBytes(),"windows-1251");
        }
```

```
String GROUP=request.getParameter("group");
if(GROUP==null) GROUP="Группа не задана";
        else
{
        GROUP=new String(GROUP.getBytes(),"windows-1251");
}
```

После чего методом setContentType серверу сообщается, что возвращаемые данные – страница HTML (MIME-тип «text/html»). Если этого не сделать, то сервер решит, что вы посылаете обычный текст:

```
response.setContentType("text/html; charset=windows-1251");
```

Методом getWriter сервлет получает доступ к потоку вывода, через который серверу посылается результат:

```
Writer out=new BufferedWriter(new
OutputStreamWriter(response.getOutputStream(),"Cp1251"));
```

Далее генерируем детали стандартного заголовка Web-страницы. Генерируемое содержание Web-страницы отправляется клиенту при помощи выходного потока и его метода write:

```
out.write("<HTML>\n");
out.write("<HEAD><TITLE>
```

Пример сервлета`</TITLE></HEAD>\n");`

Затем сервлет выводит основную часть содержимого Web-страницы. В нашем случае просто происходит генерация двух абзацев текста, в которых выводятся значения полученных сервлетом параметров:

```
out.write("<p>\n");
out.write("<font size=\"+1\" color=\"Black\">Введено имя: </font>\n");
out.write("<font size=\"+1\" color=\"Red\">"+NAME+"</font>\n");
out.write("</p>\n");
out.write("<p>\n");
out.write("<font size=\"+1\" color=\"Black\">Введена группа: </font>\n");
out.write("<font size=\"+1\" color=\"Red\">"+GROUP+"</font>\n");
out.write("</p>\n");
```

Завершается вывод генерацией стандартного окончания HTML-страницы:

```
out.write("</body>\n");
```

out.write("</html>\n");

Перед завершением метода следует отправить содержимое потока вывода клиенту и закрыть этот поток:

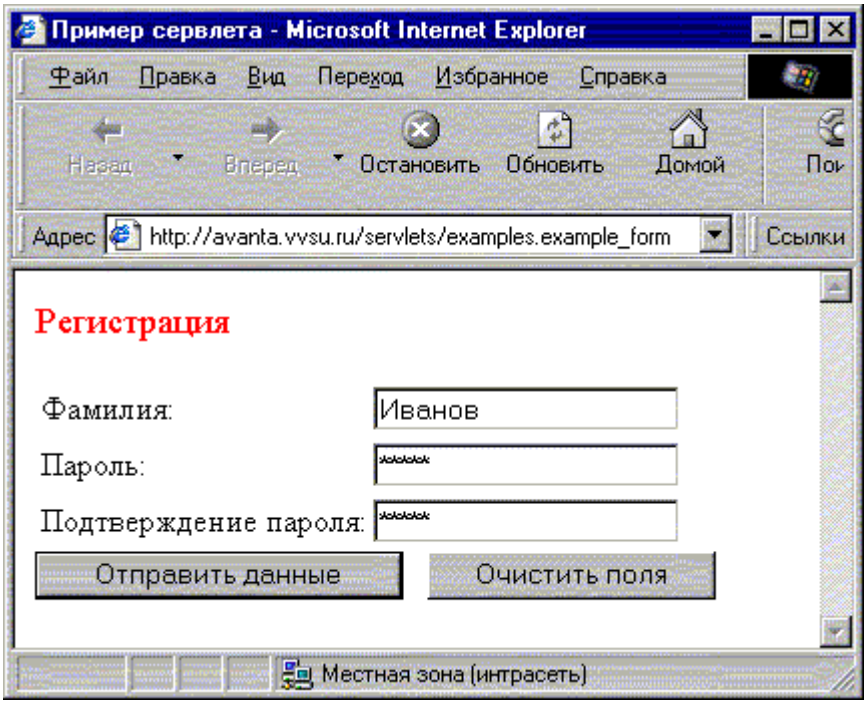
out.flush(); out.close();

return;

}

}

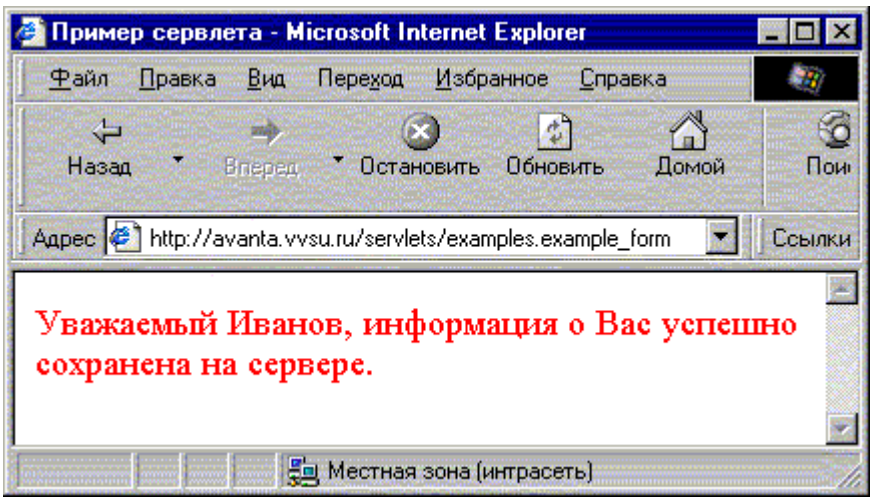
Отлаживать подобного рода сервлеты не просто, а очень просто. Необходимо скопировать class-файл это сервлета в подкаталог example каталога сервлетов Web-сервера, затем строке адреса окна браузера на клиентском месте набрать URL сервлета (например, http://avanta.vvsu.ru/servlets/examples.example?name=Иванов&group=Гр.-123)и нажать Enter — и все ваши ошибки сразу же становятся видны.



5. Сервлет, обрабатывающий запросы на основе методов CET и POST

Приведем пример сервлета *example_form* (пример 2), запрос на который можно делать не только методом GET при помощи универсального запроса ресурса по его универсальному адресу URL, но и методом POST, с помощью которого на сервер передаются данные, введенные пользователем в поля интерактивных Web-страниц. В этом сервлете при запросе его методом GET (через ссылку на его URL) - генерируется некая форма, данные из которой затем методом POST отправляются на Web-сервер для обработки этим же сервлетом:

Приведем только листинг исходного текста сервлета, он достаточно ясен для понимания и без подробного объяснения:



/---------- Пример 2. Файл example_form.java ----------*/*

package examples; // пакет для сервлета-примера

// способ вызова клиентом класса example.example_form через строку адреса окна браузера (методом GET)

// http://avanta.vvsu.ru/servlets/examples.example_form

// способ вызова Web-сервером класса example.example_form для обработки данных интерактивной формы, передаваемых методом POST

//

http://avanta.vvsu.ru/servlets/examples.example_form?name=Имя&pwd=Пароль&pwd1=ПодтверждениеПароля

// предполагается, что сервлет находится в директории сервлетов на WWW-сервере avanta.vvsu.ru

import java.io.;*

import javax.servlet.;*

import javax.servlet.http.;*

public class example_form extends HttpServlet

{

// ----------

public void doGet (HttpServletRequest request,HttpServletResponse response) throws ServletException, IOException

{

// передача типа генерируемого сервлетом содержимого

response.setContentType("text/html; charset=windows-1251");

// получение выходного потока для вывода генерируемого содержимого

Writer out=new BufferedWriter(new OutputStreamWriter(response.getOutputStream(),"Cp1251")); // начало формирования результата в формате html-файла

```
out.write("<HTML><HEAD><TITLE>Пример
сервлета</TITLE></HEAD><body>\n");
// формирование содержимого - формы ввода
out.write("<p><font size=\"+1\" color=\"Red\">Регистрация</font></p>\n");
out.write("<form action=\"http://avanta.vvsu.ru/servlets/examples.example_form\" meth-
od=\"POST\">\n");
out.write("<table>\n");
out.write("<tr><td>Фамилия:</td><td><input type=\"Text\"
name=\"name\"></td></tr>\n");
out.write("<tr><td>Пароль:</td><td><input type=\"Password\"
name=\"pwd\"></td></tr>\n");
out.write("<tr><td>Подтверждение пароля:</td><td><input type=\"Password\"
name=\"pwd1\"></td></tr>\n");
out.write("</table>\n");
out.write("<input type=\"Submit\" value=\"Отправить данные\">  \n");
out.write("<input type=\"Reset\" value=\"Очистить поля\"><p>\n");
out.write("</form>\n");
// завершение формирования результата и закрытие потока
out.write("</body></html>\n");
out.flush(); out.close();
return;
}
// ----------------------------------------------------------------------------------------------
public void doPost (HttpServletRequest request,HttpServletResponse response) throws
ServletException, IOException
{
// получение значения параметра имени, передаваемое через параметры
String NAME=request.getParameter("name");
if( NAME!=null) NAME=new String((NAME.trim()).getBytes(),"windows-1251");
// получение значения параметра пароля, передаваемое через параметры
String PWD=request.getParameter("pwd");
if(PWD!=null) PWD=new String((PWD.trim()).getBytes(),"windows-1251");
// получение значения параметра подтвержденияпароля, передаваемое через пара-
метры
String PWD1=request.getParameter("pwd1");
```

```
if(PWD1!=null) PWD1=new String((PWD1.trim()).getBytes(),"windows-1251");
// передача типа генерируемого сервлетом содержимого
response.setContentType("text/html; charset=windows-1251");
// получение выходного потока для вывода генерируемого содержимого
Writer out=new BufferedWriter(new OutputStreamWrit-
er(response.getOutputStream(),"Cp1251"));                    // начало
формирования результата в формате html-файла
out.write("<HTML><HEAD><TITLE>Пример
сервлета</TITLE></HEAD><body>\n");
//  формирование содержимого
if(NAME==null||NAME.equals(""))
out.write("<p>Ошибка! Не введено имя... </p><p><a
href=\"http://avanta.vvsu.ru/servlets/examples.example_form\">Назад, в заполнение фор-
мы</a></p>\n");
        else if(PWD==null||PWD.equals(""))
out.write("<p>Ошибка! Не введен пароль... </p><p><a
href=\"http://avanta.vvsu.ru/servlets/examples.example_form\">Назад, в заполнение фор-
мы</a></p>\n");
        else if(PWD1==null||PWD1.equals(""))
out.write("<p>Ошибка! Не введено подтверждение пароля... </p><p><a
href=\"http://avanta.vvsu.ru/servlets/examples.example_form\">Назад, в заполнение фор-
мы</a></p>\n");
else if(!PWD1.equals(PWD))
out.write("<p>Ошибка! Подтверждение пароля не совпадает с паролем...
</p><p><a href=\"http://avanta.vvsu.ru/servlets/examples.example_form\">Назад, в заполнение
формы</a></p>\n");
{
// здесь можно сделать какие-либо действия по регистрации пользователя,
// например, сохранение информации о нем в базе данных
// . . . . . .
// выдача результатов регистрации
out.write("<p><font size=\"+1\" color=\"Red\">Уважаемый "+NAME+", информа-
ция о Вас успешно сохранена на сервере.</font></p>\n");                    }
// завершение формирования результата и закрытие потока
out.write("</body></html>\n");
out.flush(); out.close();
return;
```

```
    }
}
/*-------------------------------------------------------------*/
```

6. Сервлет MyselfInfo

Теперь Вам предлагается выполнить самостоятельную практическую работу - создать и проверить сервлет, который выдает клиенту некоторую информацию о Вас. Постарайтесь, чтобы внешний вид генерируемой страницы был достаточно привлекателен для посетителей этой Вашей "домашней странички".

Задания к лабораторной работе

Задание 1. Проверить и объяснить работу сервлетов example и example_form, рассматриваемых в данной главе в качестве примеров и отмеченных курсивом.

Задание 2. Создать сервлет MyselfInfo и объяснить его работу.

Задание 3. Дать ответы на контрольные вопросы.

Контрольные вопросы

1. Что такое сервлеты?
2. Что самое необходимо сделать в первую очередь для создания приложение-сервлета?
3. Какова последовательность работы сервлета? Какие методы какого интерфейса при этом вызываются сервером?
4. Почему метод service сервлета должен быть синхронизирован для работы в многопоточных средах?
5. Какой класс JSDK отвечает за обработку HTTP-запросов?
6. Какие функции выполняет метод service класса HttpServlet? Что должен сделать разработчик для того, чтобы сервлет специальным образом реагировал на HTTP-запрос?
7. При помощи объектов какого класса сервлету передается запрос от клиента? Какую информацию можно узнать с помощью этого класса?
8. С помощью объектов какого класса сервлет пересылает генерируемые им данные клиенту?

ЛАБОРАТОРНАЯ РАБОТА № 11

РАБОТА С БАЗАМИ ДАННЫХ, ИСПОЛЬЗОВАНИЕ ИНТЕРФЕЙСА JDBC(2 ЧАСА)

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ

Интерфейс связности баз данных Java JDBC (Java Database Connectivity) представляет собой прикладной программный интерфейс, реализованный в виде ряда классов и интерфейсов языка Java. Его назначение состоит в том, чтобы предоставить апплетам и прило-

жениям (в частности, сервлетам) возможность взаимодействия с машинами реляционных баз данных независимо от платформы.

Для тех программистов, которые занимались разработкой приложений открытого интерфейса связности баз данных ODBC (Open Database Connectivity), API-интерфейс JDBC может показаться знакомым. Интерфейс JDBC во многих отношениях напоминает ряд объектно-ориентированных оболочек ODBC. Это подобие является не случайным, а неизбежным, поскольку оба интерфейса, ODBC и JDBC, основаны на стандартах X/Open SQL Call Level Interface и ANSI SQL-92 (SQL - Structured Query Language - язык структурированных запросов).

Хотя в основе интерфейсов ODBC и JDBC лежат указанные выше общепринятые стандарты, у них имеются существенные отличия. Так, API-интерфейс ODBC реализован на языке C, тогда как интерфейс JDBC реализован на языке Java. API-интерфейс на основе языка Java дает интерфейсу JDBC существенное преимущество объектно-ориентированного характера. Кроме того, API-интерфейс на основе языка Java обладает преимуществом большей переносимости, поскольку код Java интерпретируется виртуальной машиной во время выполнения, а код C должен быть скомпилирован для конкретной платформы. С другой стороны, большинство собственных интерфейсов систем управления реляционными базами данных RDBMS (Relation Database Management System) создано на основе языка Си в значительной степени опирается на использование указателей, которые отсутствуют в языке Java. В результате многие драйверы JDBC опираются на библиотеки для конкретных платформ, которые служат в качестве моста между реализациями на языке Java и собственной библиотекой конкретной RDBMS.

Для интерфейса JDBC отнюдь не обязательно наличие JDBC-совместимых драйверов. Сопряжение интерфейсов ODBC и JDBC является составной частью пакета разработки программ JDBC SDK компании JavaSoft и позволяет приложениям получать доступ к драйверам ODBC через API-интерфейс JDBC. Это сопряжение фактически служит в качестве драйвера JDBC, преобразуя вызовы интерфейса JDBC в соответствующие вызовы интерфейса ODBC и передавая их для обработки диспетчеру драйверов ODBC. Что же касается интерфейса ODBC, то для него указанное сопряжение является лишь очередным клиентским приложением.

1. Написание апплетов, сервлетов и приложений JDBC

Этот раздел посвящен рассмотрению пакета java.sql, отдельные части которого опираются на классы из пакета JDK. Представленные здесь сведения и примеры основаны на версии 1.1.x пакета JDK и версии 1.2x API-интерфейса Java.

Последующие разделы предназначены для демонстрации использования конкретных классов и интерфейсов JDBC. Обработка исключений в примерах, представленных в этих разделах, опущена, поскольку обработка исключений в интерфейсе JDBC описана отдельно. А пока предполагается, что каждый фрагмент кода включен в блок try.

1.1 Соединение с базой данных

Прежде чем появится возможность установить соединение с RDBMS, необходимо загрузить соответствующий драйвер JDBC. Этот драйвер можно загрузить явным образом с помощью метода forName() класса java.lang.Class. Чтобы загрузить драйвер, передайте этому методу полностью определенное составное имя класса. Например, если используется драйвер Oracle, то загрузить его можно с помощью следующего оператора (при этом пакет классов oracle должен быть установлен на компьютере в директории, доступной виртуальной машине Java для поиска классов):

```
Class.forName("oracle.jdbc.driver.OracleDriver"); // загрузка драйвера
```

Есть и другой способ загрузки драйвера. В нем вместо метода Class.forName() используется метод registerDriver() класса DriverManager:

```
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver()); // загрузка драйвера
```

Загруженный тем или иным способом драйвер драйвер становится доступным для диспетчера драйверов, который используется для установления соединения. Соединение его устанавливается методом getConnection(). У метода getConnection() класса DriveManager имеются три формы, причем каждая форма возвращает интерфейс Connection (реализованный драйвером) и воспринимает URL (универсальный локатор ресурсов) в качестве первого аргумента формы:

```
jdbc:subprotocol:subname
```

Об использовании имени драйвера, URL и конкретной формы метода getConnection() см. документацию на соответствующий драйвер. Для большинства драйверов JDBC требуются дополнительные свойства соединения, предназначенные для определения конкретного экземпляра базы данных, к которому осуществляется доступ, обеспечения опознавания, установки параметров сеанса и т.д. Например, в большинстве драйверов Oracle требуется наличие строки подключения SQL*Net, имени пользователя и пароля. Приведем пример установки соединения с базой данных:

```
java.sql.Connection conn =DriverManager.getConnection
 ("jdbc:oracle:thin:@uran:1521:ORCL","scott","tiger");
 // установка соединения
```

После установки соединения при помощи объекта conn класса Connection апплет, сервлет или приложение может обращаться к базе данных для выполнения того или иного запроса. После того, как соединение больше не требуется, его можно закрыть методом close() класса Connection:

conn.close(); // закрытие соединения

1.2 Применение интерфейса DatabaseMetaData

Интерфейс java.sql.DatabaseMetaData имеет неоценимое значение для тех приложений, которые должны поддерживать несколько машин баз данных или несколько версий одной и той же RDBMS. Его можно использовать для определения имени и версии драйвера, конкретной RDBMS и ее версии, поддерживаемой грамматики SQL, уровня выделения транзакций, уровней процедурной поддержки и т.д. Интерфейс DatabaseMetaData возвращается из метода getMetaData() интерфейса Connection. В приведенном ниже фрагменте кода показан один из возможных вариантов использования этого интерфейса:

java.sql.DatabaseMetadata md=conn.getMetadata(); // получение информации о соединении

String driverName=md.getDriverName();

String driverVer=md.getDriverVersion();

String dbmsName=md.getDatabaseProductName();

String dbmsVer=md.getDatabaseProductVersion();

Приложение может использовать данные о поставщике и версии DBMS для динамического создания операторов SQL в соответствии с конкретной используемой RDBMS. Например, внешние соединения в значительной степени зависят от RDBMS. В интерфейсе JDBC предусмотрены управляющие последовательности для устранения этих несогласованностей, однако, к сожалению, поддержка в драйверах этих управляющих последовательностей оказывается несогласованной. Пример синтаксиса управляющих последовательностей интерфейса JDBC представлен в разделе, посвященном доступу к процедурам и функциям. В одних случаях, возможно, потребуется применить синтаксис управляющих последовательностей, тогда как в других более надежным может оказаться создание внутри приложения операторов SQL для конкретной DBMS или драйвера. Этот выбор в значительной степени зависит от числа продуктов и версий DBMS, которые необходимо поддерживать в приложении (даже если это известно заранее).

Этот вопрос может показаться незначительным, однако влияние различий в грамматике SQL и реализациях драйверов может оказаться существенным, если оно не будет учтено при разработке приложения. Если бы это было не так, тогда многие методы интерфейса DatabaseMetaData оказались бы просто не нужны.

1.3 Посылка статичных SQL-запросов

Перед тем как выполнить SQL-запрос, необходимо создать его создать. Объект запроса класса java.sql.Statement получается посредством вызова, например, метода createStatement() для экземпляра соединения. Затем после создания текста запроса егоследует передать базе данных, где он разбирается, оптимизируется и выполняется. Для посылки запроса используется один из методов интерфейса Statement:

ResultSet executeQuery(String sql);

int executeUpdate(String sql);

boolean execute(String sql);

Метод, выбираемый для посылки запроса базе данных, зависит от типа запроса и типа возвращаемого значения. Если запрос возвращает в виде результата некую таблицу строк данных, то необходимо использовать executeQuery(). В этом случае запрос - это чаще всего статический SQL-запрос SELECT. Текст запроса просто передается как строчный аргумент. Этот метод возвращает объект набора результатов java.sql.ResultSet в виде таблицы.

Если запрос не возвращает ничего (это не то же самое, что возвращение нуля строк) или возвращает целочисленное значение, как в случае с SQL-операторами INSERT, UPDATE или DELETE, нужно использовать executeUpdate(). Вызов возвращает целочисленное значение или нуль для инструкций, не возвращающих ничего.

Когда SQL-запрос возвращает более чем один результат, необходимо использовать execute() для выполнения этого запроса.

Приведем пример создания и выполнения запроса:

Statement stmt=conn.createStatement();

// создание запроса

// создание текста запроса (выдать результат в виде таблицы, в столбцах которой помещены

// а) фамилии, б) имена, в) оклады всех людей, информация о которых хранится в таблице PAYROLL)

String sqlQuery="select LAST, FIRST, SALARY from PAYROLL";

ResultSet rset=stmt.executeQuery(sqlQuery);

// выполнение запроса и получения результата запроса - таблицы выбранных строк

// использование результата запроса - объекта rset

. . . . . .

stmt.close(); // освобождение ресурсов запроса

Интерфейс ResultSet, входящий в состав интерфейса JDBC, как подразумевает его наименование, специально предназначен для выборки результатов (см. пункт о выборке результатов).

Приведем еще один пример создания и выполнения запроса. Этот запрос изменяет таблицу базы данных (обновляет данные) и, поэтому, требует фиксации изменений методом commit() для объекта соединения:

```
Statement stmt=conn.createStatement();
 // создание запроса
String sqlQuery="update PAYROLL set SALARY=SALARY*1.1 where SALARY < 1000";
// создание текста запроса
int=stmt.executeUpdate(sqlQuery);
 // выполнение запроса и получения результата запроса - число строк, в которых произведено обновление
// использование результата запроса - целого числа
. . . . . .
conn.commit();
 // фиксация изменений
stmt.close();
 // освобождение ресурсов запроса
```

Заметим, что если не требуется зафиксировать сделанные изменения, а, наоборот, необходимо отменить результаты выполнения запроса, следует воспользоваться следующим вызовом:

```
conn.rollback();
// отмена фиксации результатов выполнения запроса
```

1.4 Посылка параметризированных и частовыполняемых SQL-запросов

Интерфейс java.sql.Statement следует использовать, если запрос SQL является статичным. А для параметризованных запросов SQL или часто выполняемых запросов следует использовать интерфейс java.sql.PreparedStatement. В том случае, когда используется интерфейс PreparedStatement, имеется возможность выполнить предкомпиляцию операторов SQL и привязать план доступа к конкретной базе данных. Интерфейс PreparedStatement содержит методы установки значений параметров до выполнения. Это дает возможность многократно выполнять предварительно скомпилированные операторы SQL с разными значениями параметров, что может привести к существенному повышению производительности, в особенности при сложных запросах.

Чтобы проиллюстрировать указанные выше различные методы, допустим, что имеется приложение, используемое для отображения данных служащего (фамилию, имя, оклад) по номеру его социального страхования. Это можно осуществить с использованием интерфейса Statement или PreparedStatement. В приведенном ниже коде показано непосредственное выполнение операторов SQL с помощью интерфейса Statement:

```
// пусть txtSSN - это ссылка на поле ввода класса java.awt.TextField, который воспринимает
// номер социального страхования в качестве вводимых пользователем данных
String sqlQuery="select LAST, FIRST, SALARY from PAYROLL where SSN="+txtSSN.getText();
 // текст запроса
java.sql.Statement stmt=conn.createStatement();
// создание запроса
java.sql.ResultSet rset=stmt.executeQuery(sqlQuery);
 // выполнение запроса и получения результата запроса
// использование результата запроса - объекта rset
. . . . . .
stmt.close();
 // освобождение ресурсов запроса
```

Этот же запрос может быть параметризирован и выполнен следующим образом:

```
// комментарии - аналогичные
String sqlQuery="select LAST, FIRST, SALARY from PAYROLL where SSN=?";
// текст запроса
java.sql.PreparedStatement pstmt=conn.prepareStatement(sqlQuery);
 // создание запроса
pstmt.setLong(1,java.lang.Long.parseLong(txtSSN.getText()));
// установка параметра
java.sql.ResultSet rset=pstmt.executeQuery();
// выполнение запроса
// использование результата запроса - объекта rset
. . . . . .
```

Интерфейс PreparedStatement позволяет повторно выполнять этот запрос с новым параметром любое число раз, создавая новый набор ResultSet:

pstmt.setLong(1,java.lang.Long.parseLong(txtSSN.getText()));

// установка параметра

java.sql.ResultSet rset=pstmt.executeQuery();

// выполнение запроса

// использование результата запроса - объекта rset

. . . . . .

Когда запрос pstmt уже не нужен, его следует закрыть, освобождая связанные с ним ресурсы:

pstmt.close(); // освобождение ресурсов запроса

Метод setLong() является одним из методов интерфейса PreparedStatement, используемых для задания значений параметров. Каждый из этих методов воспринимает индекс параметров с отсчетом от единицы и значение параметра. Например, метод setString() воспринимает индекс параметров и класс java.lang.String.

К сожалению, не все драйверы поддерживают все типы данных интерфейса JDBC и не одинаково выполняют подготовленные операторы SQL. Некоторые драйверы могут задержать интерпретацию и установку параметров до тех пор, пока не начнется выполнение запроса. Это позволяет повысить производительность, однако может нарушить обработку исключений по причине задержки активизации некоторых исключений. Например, исключение SQLException может быть активизировано в том случае, если один из методов set() вызывается с типом данных, несовместимым с соответствующим столбцом базы данных. Если интерпретация и установка параметров задержаны, то это же исключение не будет активизировано до тех пор, пока не будет вызван метод executeQuery().

1.5 Выборка результатов

Рассмотрим теперь, как выбрать результаты запроса, если они выдаются в виде набора java.sql.ResultSet. Интерфейс ResultSet, входящий в состав интерфейса JDBC, как подразумевает его наименование, специально предназначен для выборки результатов.

Интерфейс Statement, входящий в состав интерфейса JDBC, представляет собой основной интерфейс взаимодействия с базой данных. Он расширяется за счет интерфейсов CallableStatement и PreparedStatement. Интерфейс ResultSet можно получить через интерфейс Statement или PreparedStatement (интерфейс CallableStatement в основном используется для процедурных расширений и рассматривается в последующих разделах.).

Интерфейс Statement следует использовать для создания интерфейса ResultSet только в том случае, если запрос SQL является статичным. А для параметризованных запросов SQL или часто выполняемых запросов следует использовать интерфейс PreparedStatement. В том

случае, когда используется интерфейс PreparedStatement, имеется возможность выполнить предкомпиляцию операторов SQL и привязать план доступа к конкретной базе данных. Интерфейс PreparedStatement содержит методы установки значений параметров до выполнения. Это дает возможность многократно выполнять предварительно скомпилированные операторы SQL с разными значениями параметров, что может привести к существенному повышению производительности, в особенности при сложных запросах.

Независимо от того, осуществляется ли непосредственное (с помощью интерфейса Statement) или подготовленное (с помощью интерфейса PreparedStatement) выполнение операторов SQL, возвращаемый интерфейс ResultSet оказывается одним и тем же. Доступ к строке осуществляется с помощью метода next() интерфейса ResultSet. Этот метод возвращает логическое значение, которое является истинным до тех пор, пока не будут считаны все строки. Кроме того, он может активизировать исключение SQLException, если возникнет ошибка.

После того как курсор будет установлен на строку, значения столбцов выбираются с помощью одного из методов get() интерфейса ResultSet (getStringO, getInt() и др.). Каждый из этих методов воспринимает для выборки индекс столбца с отсчетом от единицы или имя столбца. Следующий код служит расширением примера запроса, представленного в приведенных выше фрагментах кода:

```
String lastName="";
String firstName="";
float salary=0;
while(rset.next())// цикл по строкам результата запроса
{
// получение значений столбцов результата запроса
lastName=rset.getString(1); // или lastName=rset.getString("LAST");
firstName=rset.getString(2); // или firstName=rset.getString("FIRST");
salary=rset.getFloat(3); // или salary=rset.getFloat("SALARY");
// выполнить что-либо по использованию полученных значений
. . . . . .
}
rset.close(); // освобождение результата запроса
```

Рекомендуется закрывать результирующий набор, как только в нем отпадает необходимость. Метод close() применяется также в интерфейсах Statement и PreparedStatement. Однако, возможно, потребуется оставить интерфейсы Statement открытыми для последующих

выполнений операторов SQL, поскольку их закрытие освобождает все связанные с ними ресурсы. Если ссылка на интерфейс Statement становится недействительной, метод close() вызывается автоматически в процессе сборки мусора. Вызов метода close() до определения недействительности ссыкли гарантирует немедленное освобождение внешних ресурсов.

1.6 Применение интерфейса ResultSetMetaData

Интерфейс java.sql.ResultSetMetaData используется для получения сведений о результирующем наборе и его столбцах во время выполнения. Этот интерфейс может стать эффективным инструментальным средством при разработке многократно используемых компонентов отображения, когда характер результирующего набора не известен на этапе проектирования. Интерфейс ResultSetMetaData может быть получен из любой допустимой ссылки на интерфейс ResultSet:

```
java.sql.ResultSetMetadata rsetInfo=rset.getMetadata(); // получение информации о результате запроса
```

Интерфейс ResultSetMetaData предоставляет методы определения числа и имен столбцов, типов данных, ширины элементов отображения и т.п. Следующий фрагмент кода можно использовать для сбора информации о заголовках и размерах столбцов, что способствует отображению результатов запроса в виде координатной сетки:

```
int numCols=rsetInfo.getColumnCount(); // количество столбцов результата запроса
String colNames[]=new String[numCols];
int colSizes[]=new int[numCols];
int colTypes[]=new int[numCols];
for(int i=0; i<numCols; i++) // цикл по столбцам результата запроса
{
// получение информации о столбцах результата запроса
colNames[i]=rsetInfo.getColumnLabel(i+1); // индексы столбцов в JDBC начинаются с 1
colSizes[i]=rsetInfo.getColumnDisplaySize(i+1);
colTypes[i]=rsetInfo.getColumnType(i+1);
}
```

Метод getColumnType() возвращает целое значение, представляющее собой тип данных SQL. Константы типа данных SQL определены в классе java.sql.Types. В контексте приведенного выше примера этой информацией можно воспользоваться для определения способа отображения столбца с выравниванием по левому или по правому краю. Следующий фрагмент кода служит расширением приведенного выше примера для выборки значений столбцов в каждой строке:

```
int numCols=rsetInfo.getColumnCount();
String colValues[]=new String[numCols];
int i=0;
while(rset.next()); // цикл по строкам результата запроса
{
for(int i=0; i<numCols; i++) // цикл по столбцам результата запроса
{
colValues[i]=rset.getString(rsetInfo.getColumnLabel(i+1)); // получение значения столбца результата запроса

// выполнить что-либо по использованию полученного значения этого столбца
. . . . . .
}
}
```

В приведенном выше фрагменте кода метод getString() используется для получения значений всех столбцов. Этот метод обычно можно использовать независимо от типа данных столбца, поскольку в большинстве драйверов поддерживается преобразование любого типа данных в его строковое представление. С другой стороны, сведения о типе данных могут быть использованы в приложении для сохранения значений столбцов в более подходящей структуре с последующим применением соответствующего метода get() к каждому столбцу в зависимости от его типа данных. Необходимость в этом может возникнуть, если данные используются в математических расчетах, однако нередко более удобной оказывается выборка значений столбцов в виде строк в целях их последующего отображения.

При использовании метода getlnt() и других числовых методов может потребоваться специальная обработка пустых значений. Так, метод wasNull() может быть использован для того, чтобы определить, является ли пустым значение последнего считанного столбца:

```
double comm=rset.getDouble("COMM");
if(rset.wasNull())
// выполнить обработку пустого значения (например, установить соответствующий признак)
. . . . . .
```

1.7 Доступ к хранимым функциям и процедурам

Доступ к хранимым процедурам и функциям выполняется с помощью интерфейса java.sql.CallableStatement и синтаксиса управляющих последовательностей SQL. Интерфейс CallableStatement расширяет интерфейс PreparedStatement, который, в свою очередь, расши-

ряет интерфейс Statement. Интерфейс CallableStatement существует для того, чтобы предоставлять дополнительные методы, специально предназначенные для вызова хранимых процедур и функций и обработки выходных параметров.

Синтаксис управляющих последовательностей SQL для вызова хранимой процедуры из интерфейса JDBC имеет следующую общую форму:

{call proc_name(arg1,arg2,...)}

А вызов функции, выполняемой через интерфейс JDBC, имеет следующую общую форму:

{?=call func_name(arg1,arg2,...)}

Ссылку на объект CallableStatement можно получить непосредственно из интерфейса Connection, почти так же как ссылку на объект PreparedStatement. Хотя объекты PreparedStatement можно получить с помощью метода preparedStatement(), объекты CallableStatement можно получить с помощью метода prepareCall(). Допустим, что существует хранимая процедура Oracle с именем delete_rows,предназначенная для удаления из некоторой таблицы тех записей, в которых одно из полей больше одного числа, но меньше другого (оба эти числа передаются через параметры). Код JDBC для выполнения операции вставки данных может иметь следующий вид:

```
int min=10; String max="20";
String querySql="{call delete_rows (?,?)}";// текст запроса
CallableStatement cstmt = conn.prepareCall(query,Sql); // создание запроса
cstmt.setInt(1,min); // установка значения первого аргумента процедуры
cstmt.setString(2,max); // установка значения второго аргумента процедуры
cstmt.executeUpdate(); // выполнение запроса
conn.commit(); // фиксация изменений
cstmt.close(); // освобождение ресурсов запроса
```

В действительности эту процедуру можно было бы выполнить с помощью интерфейса PreparedStatement, а не интерфейса CallableStatement, поскольку в этом случае нужны только те методы класса CallableStatement, которые унаследованы от класса PreparedStatement. Однако рекомендуется использовать интерфейс CallableStatement для выполнения процедур, что позволяет свести к минимуму влияние изменений в базовой процедуре. Интерфейс CallableStatement необходимо использовать для доступа к выходным параметрам, которые рассматриваются в следующем разделе.

1.8 Применение выходных параметров

Нередко в процедурах и функциях Oracle полезным оказывается применение выходных параметров. Интерфейс CallableStatement, входящий в состав интерфейса JDBC, предоставляет методы выборки этих значений. Допустим, что операция удаления данных из таб-

лицы delete_rows, рассмотренная в предыдущем разделе, написана в виде пакетной функции и что это функция возвращает количество удаленных строк. Для примера допустим далее, что значение второй входной параметр является также и выходным параметром этой функции (пусть, например, там передается начальное количество строк таблицы, из которой производится удаление). В приведенном ниже фрагменте кода показан вызов этой функции и выборка выходных параметров:

```
int min=10, max=20;
String querySql="{?= call delete_rows (?,?)}";// текст запроса
CallableStatement cstmt = conn.prepareCall(query,Sql); // создание запроса
cstmt.registerOutParameter(1,java.sql.Types.INTEGER);
 // регистрация типа возвращаемого функцией значения
cstmt.setInt(2,min); // установка значения первого аргумента
cstmt.setInt(3,max); // установка значения второго аргумента
cstmt.registerOutParameter(3,java.sql.Types.INTEGER);
// регистрация типа значения, которое возвращается через второй аргумент функции
cstmt.executeUpdate(); // выполнение запроса
int del=cstmt.getInt(1); // получение возвращаемого функцией значения
int rows=cstmt.getInt(3); // получение значения, возвращаемого во втором аргументе функции
conn.commit(); // фиксация изменений
cstmt.close(); // освобождение ресурсов запроса
```

Что же касается интерфейса JDBC, то функция Oracle является процедурой, имеющей, по меньшей мере, один выходной параметр. Не существует специальной обработки возвращаемого функцией значения. Аргументами метода registerOutParameter() являются индекс параметра с отсчетом от единицы и тип данных SQL этого параметра. Значения выходных параметров выбираются после выполнения с помощью соответствующего метода get() (getStringQ, getF1oat() и др.). Используемый метод get() должен быть совместим с типом данных, передаваемым методу registerOutParameterO. Методы get() класса CallableStatement аналогичны методам get() класса ResultSet, которые воспринимают индекс столбца в качестве своего аргумента. Кроме того, подобно соответствующему методу класса ResultSet, метод wasNulI() позволяет определить, был ли пустым последний считанный выходной параметр.

Входные и выходные параметры INOUT должны быть заданы и зарегистрированы. Тип данных SQL, передаваемый методу registerOutParameter(), должен соответствовать используемому методу set(). Параметр INOUT может быть задан и зарегистрирован так, как, например, в следующем примере:

cstmt.setInt(3,max);

cstmt.registerOutParameter(3,java.sql.Types.INTEGER);

2. Обработка исключений JDBC

Обработка исключений была опущена в предыдущих примерах, поскольку этот вопрос требует отдельного рассмотрения. Почти всякий вызов интерфейса JDBC, который выполняется в приложении, способен активизировать исключение, и поэтому он должен быть заключен в блок **try.** В интерфейсе JDBC определены три исключения в иерархической взаимосвязи. Эта иерархия сверху вниз выглядит следующим образом:

SQLException

SQLWarning

DataTruncation

Исключение SQLException активизируется методами интерфейса JDBC в том случае, если в базе данных возникает ошибка. Существует ряд методов, которые могут быть вызваны для выборки информации об ошибке:

- Метод int getErrorCodeO возвращает код ошибки конкретной базы данных
- Метод String getSQLState() возвращает состояние X/Open SQLState, связанное с ошибкой
- Метод String getMessage() наследуется из класса java.lang.Exception (возвращает описание ошибки)

Исключения JDBC могут быть связаны в цепочку, поэтому, если во время выполнения одной операции возникает более одной ошибки, информация обо всех ошибках сохраняется. Метод getNextException() возвращает ссылку на следующее исключение в цепочке или пустое значение, если исключений больше нет.

SQLWarning представляет собой такое исключение, которое не активизируется, но связывается в цепочку с конкретным объектом ResultSet или объектом оператора, к которому оно применяется. Как и в случае с исключениями SQLException, в цепочку может быть связано более одного исключения SQLWarning. Метод getNextWarning() класса SQLWarning возвращает следующее предупреждение в цепочке или пустое значение, если предупреждений больше нет. Например, при обработке результирующего набора в приложении наличие предупреждений можно проверить с помощью метода getWarnings() следующим образом:

java.sql.SQLWarning warn=rset.getWarnings();

```
while(warn!=null)
{
// сделать что-нибудь с предупреждением
. . . . .. . . . .
warn=warn.getNextWarning();
}
```

Исключение DataTruncation является особым случаем исключения SQLWarning, которое применяется к значениям столбцов и параметров. Исключение DataTruncation может быть связано в цепочку с объектом результирующего набора, однако оно активизируется при его применении к объекту оператора. Оно предоставляет дополнительные методы определения индекса столбца или параметра, к которому оно применяется, фактической и ожидаемой длины и т.д. При связывании исключения DataTruncation в цепочку с результирующим набором отличить его от других предупреждений можно на основании значения "01004" его состояния SQLState. Приведенный выше фрагмент кода можно расширить для проверки усечения данных:

```
int colNum, javaLen, dbLen;
java.sql.SQLWarning warn=rset.getWarnings();
while(warn!=null)
{
if(warn.getSQLState()=="01004")
{
colName=((DataTruncation)warn).getIndex();
javaLen=((DataTruncation)warn).getTransferSize();
dbLen=((DataTruncation)warn).getDataSize();
// сделать что-нибудь с этой информацией
. . . . .. . . . .
}
else
{
// это другое исключение java.sql.SQLWarning
. . . . .. . . . .
}
warn=warn.getNextWarning();
}
```

Методы getTransferSize() и getDataSize() применяются к длине столбца или параметра интерфейса JDBC, а также столбца или параметра базы данных. Исключение DataTruncation, скорее всего, будет активизировано тогда, когда значение параметра в результате окажется слишком длинным. В этом случае оно не связывается в цепочку в виде предупреждения, но активизируется в виде исключения, которое должно быть перехвачено:

```
int parmName, javaLen, dbLen;
try
{
cstmt.executeUpdate();
}
catch(SQLException e)
{
while(e!=null)
{
if(e.getSQLState()=="01004")
{
parmName=((DataTruncation)warn).getIndex();
javaLen=((DataTruncation)warn).getTransferSize();
dbLen=((DataTruncation)warn).getDataSize();
// сделать что-нибудь с этой информацией
. . . . .. . . . .
}
else
{
// это другое исключение SQLException
. . . . .. . . . .
}
e=e.getNextWarning();
}
}
```

Исключение DataTruncation может быть перехвачено как исключение SQLException, поскольку оно происходит от исключения SQLException. А когда оно перехватывается как исключение SQLException, то последнее необходимо явно привести к исключению DataTruncation, прежде чем получить доступ к методам этого интерфейса.

Обработка перехваченного исключения SQLException в значительной степени зависит от конкретных потребностей приложения. В некоторых случаях может оказаться полезной просто повторная активизация перехваченного исключения или активизация исключения специально для конкретного приложения, чтобы его можно было обработать на более высоком уровне. При активизации исключения из блока catch следует рассмотреть ряд вопросов. Для тех читателей, которые знакомы с языком Java, очевидным является первое положение: активизация исключений из вложенных блоков try — далеко не самая удачная идея. Рассмотрим следующий пример, в котором активизируется исключение, являющееся расширением класса java.lang.Exception и определенное в приложении как DBException:

```
try
{
// некоторые операции получения объектов Java, присвоения значений и т.д.
. . . . . .
try
{
// некоторые вызовы интерфейса JDBC
}
catch(SQLException sqle)
{
// выполнить очистку и активизировать исключение, чтобы его можно было активизировать на более высоком уровне
. . . . . .
throw new DBException(sqle.getMessage(),sqle.getErrorCode());
}
}
catch(java.lang.Exception e)
{
// выполнить что-нибудь
. . . . . .
}
```

Новое исключение DBException, активизированное из внутреннего блока, перехватывается во внешнем блоке, поскольку исключение DBException является потомком класса java.lang.Exception. Добиться требуемого результата можно с помощью нескольких перехватов для внешнего блока try:

```
try
```

```
{
// некоторые операции получения объектов Java, присвоения значений и т.д.
// некоторые вызовы интерфейса JDBC
. . . . . .
}
catch(SQLException sqle)
{
// выполнить очистку и активизировать исключение, чтобы его можно было активизировать на более высоком уровне
. . . . . .
throw new DBException(sqle.getMessage(),sqle.getErrorCode());
}
catch(java.lang.Exception e)
{
// выполнить что-нибудь
. . . . . .
}
```

Исключения должны перехватываться от низшего к высшему уровню иерархии. Если исключение java.lang.Exception перехватывается до исключения DBException, то блок DBException не будет достигнут. В предыдущем примере информация из первого перехваченного исключения SQLException копируется в новое исключение DBException. Если же несколько исключений связываются в цепочку, тогда эта информация не сохраняется. Цепочка может быть сохранена в том случае, если исключение DBException расширяет класс java.lang.Exception с помощью метода SetNextException():

```
catch(Exception sqle)
{
DBException first=new DBException(sqle.getMessage(),sqle.getErrorCode());
sqle=sqle.getNextException();
while(sqle!=null)
{
first.setNextException(sqle);
}
throw first;
}
```

Для перехвата исключения SQLException нередко требуются дополнительные действия, особенно в контексте транзакции. Допустим, что в приложении определен метод, который выполняет вставку строки в три отдельные таблицы в течение одной транзакции. Если при этом любая из операций вставки потерпит неудачу, должен быть осуществлен откат всей транзакции. В этом случае следует выдать команду отката в обработчике исключений. Это лишь один простой пример множества ситуаций, в которых исключение SQLException требует выполнения очистки. Следует иметь в виду, что в результате вызовов интерфейса JDBC распределяются внешние ресурсы и изменяется состояние базы данных, и поэтому обработчики исключений нужно проектировать соответствующим образом.

3. Отладка приложений JDBC

Отладка приложения JDBC оказывается более сложной, чем отладка простого апплета Java, сервлета или автономного приложения, поскольку она включает в себя дополнительные уровни программного обеспечения для взаимодействия с базой данных. Дело еще более усложняется тем, что часть драйвера JDBC может быть реализована в виде двоичной библиотеки, которая недоступна для отладчика JDB, являющегося стандартным инструментальным средством отладки, входящим в состав пакета JDK.

К счастью, в интерфейсе JDBC предусмотрен механизм просмотра операторов, обрабатываемых драйвером, а также информации, возвращаемой из базы данных. Для регистрации информации драйвера в файле или в стандартном выводе можно воспользоваться методом setLogStream() класса DriverManager:

DriverManager.setLogStream(System.out);

Метод setLogStream() класса DriverManager воспринимает любой объект класса java.io.PrintStream. Предоставляемая информация включает вызываемый метод JDBC, значения параметров и возвращаемые данные. В некоторых случаях предоставляется дополнительная информация для конкретного драйвера, которая может оказаться еще более полезной. Например, драйверы Oracle могут регистрировать базовые функции, параметры и возвращаемые значения интерфейса OCI, обрабатываемые драйвером в результате вызова интерфейса JDBC. Зарегистрированную таким образом информацию можно использовать для определения вызовов вне заданной последовательности, синтаксических ошибок SQL и других программных ошибок. В некоторых случаях она позволяет также обнаружить недокументированные средства драйвера.

Кроме того, можно воспользоваться инструментальным средством JDBCTest только на языке Java, разработанным компаниями JavaSoft и Intersolv для проверки драйверов JDBC. Оно содержит графический интерфейс и систему меню для выполнения вызовов API-интерфейса JDBC. Вызовы интерфейса JDBC, которые приводят к ошибкам в приложениях,

могут быть выполнены через этот интерфейс, чтобы определить, была ли данная ошибка вызвана драйвером или кодом приложения. Это инструментальное средство проверки может оказаться особенно полезным при выборе одного или нескольких драйверов для проектирования и разработки приложения. Например, нельзя заранее сказать, что драйвер OracleS будет поддерживать абстрактные типы данных и коллекции OracleS с помощью методов set0bject() и get0bject(). Эти весьма специфические для RDBMS средства не обязательно должны быть полностью совместимы с интерфейсом JDBC.

К сожалению, средство JDBCTest может не работать с драйверами только на языке Java. Большинству драйверов только на языке Java требуется код для установления взаимодействия с серверным компонентом перед регистрацией драйвера и установлением соединения. Многие из этих драйверов имеют нестандартные реализации общепринятых методов, превращая тем самым JDBCTest просто в непригодное для указанных выше целей инструментальное средство. Компания JavaSoft предоставляет более автоматизированное инструментальное средство проверки (программу проверки интерфейса JDBC), а также ряд тестовых комплектов, используемых вместе с этой программой проверки. Это инструментальное средство является более гибким, поскольку оно работает под управлением файла конфигурации и позволяет выполнять разработку специализированных проверочных комплектов с минимальными усилиями. Надлежащая проверка на предшествующем проектированию этапе даст возможность заранее обнаружить присущие конкретному драйверу ограничения, что позволит избежать многих проблем в процессе разработки.

4. Сервлет, работающий с информацией из базы данных

Проверить работу сервлета *example_db* (исходный текст см. ниже), объяснить, что происходит в блоке try-catch. В заголовках уровня 2 генерируемого html-файла указать, какое действие происходит в каждой секции (ненужное вычеркнуть).

```
/*---------- Пример 1. Файл example_db.java ----------*/
package examples; // пакет для сервлета-примера
// способы вызова клиентом класса example.example_db через строку адреса окна браузера
//          http://avanta.vvsu.ru/servlets/examples.example_db
// предполагается, что сервлет находится в директории сервлетов на WWW-сервере avanta.vvsu.ru
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

```
import java.sql.*;
public class example_db extends HttpServlet
{
public void doGet (HttpServletRequest request,HttpServletResponse response) throws ServletException, IOException
{
// передача типа генерируемого сервлетом содержимого
response.setContentType("text/html; charset=windows-1251");
// получение выходного потока для вывода генерируемого содержимого
Writer out=new BufferedWriter(new OutputStreamWriter(response.getOutputStream(),"Cp1251"));
// начало формирования результата в формате html-файла
out.write("<HTML><HEAD><TITLE>Пример сервлета</TITLE></HEAD><body>\n");
//  формирование содержимого
out.write("<h1>Пример работы сервлета, использующего JDBC</h1>\n");
//---------- Что происходит в этом блоке try-catch? ----------
try
{
String query;
Statement stmt;
PreparedStatement pstmt;
CallableStatement cstmt;
ResultSet rset;
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
java.sql.Connection
conn=DriverManager.getConnection("jdbc:oracle:thin:@uran:1521:ORCL","scott","tiger");
// ---------------------------------------------------------------------
out.write("<h2>Результат работы <<статичного запроса,параметризированного запроса, хранимой процедуры(или функции)>>(ненужное удалить!!!)</h2>\n");
query="select CITY,COUNTRY,POPULATION from WORLD_CITIES order by COUNTRY";
```

```
stmt = conn.createStatement ();
rset = stmt.executeQuery (query);
while (rset.next ())
{
out.write("<p><font color=\"green\">");
out.write(rset.getString(2)+", "+rset.getString(1)+", "+rset.getString(3));
out.write("</font></p>\n");
}
rset.close();
stmt.close();
// ----------------------------------------------------------------------
out.write("<h2>Результат работы <<статичного запроса,параметризированного запроса, хранимой процедуры (или функции)>>(ненужное удалить!!!)</h2>\n");
query="select CITY,POPULATION from WORLD_CITIES  where COUNTRY=? order by CITY";
pstmt = conn.prepareStatement (query);
pstmt.setString(1,"Australia");
rset = pstmt.executeQuery ();
while (rset.next ())
{
out.write("<p><font color=\"blue\">");
out.write(rset.getString(1)+" -- "+rset.getString(2));
out.write("</font></p>\n");
}
rset.close();
pstmt.close();
// ----------------------------------------------------------------------
out.write("<h2>Результат работы <<статичного запроса,параметризированного запроса, хранимой процедуры (или функции)>>(ненужное удалить!!!)</h2>\n");
query="{?=call summa(?,?)}";
cstmt = conn.prepareCall(query);
cstmt.registerOutParameter(1,java.sql.Types.INTEGER);
cstmt.setInt(2,7);
```

```
cstmt.setString(3,"8");
cstmt.executeUpdate();
int res=cstmt.getInt(1);
out.write("<p><font color=\"teal\">Итог ..?чего?.. = "+res+"</font></p>\n");
cstmt.close();
conn.close();
}
catch(SQLException e)
{
out.write("<p>Обращение к базе данных вызвало следующее исключение: "+e.toString());
}
finally
{
// завершение формирования результата и закрытие потока
out.write("</body></html>\n");
out.flush(); out.close();
return;
}
}
}
/*--------------------------*/
```

Задания к лабораторной работе

Задание1. Проверить и объяснить работу сервлета example_db, рассматриваемого в данной главе в качестве примера и отмеченного курсивом.

Задание 2. Дать ответы на контрольные вопросы.

Контрольные вопросы

1. Как происходит соединение с базой данных?
2. Как выбрать результаты запроса?
3. Для чего используется интерфейс ResultSetMetaData?
4. Как выполняется доступ к хранимым функциям и процедурам?
5. Почему применение выходных параметров полезно в процедурах и функциях?
6. Какие дополнительные функции включает в себя отладка приложений JDBC?

ПРИЛОЖЕНИЕ 1

JAVA-ФАЙЛ ПРОСТЕЙШЕГО АППЛЕТА И
HTML-ДОКУМЕНТ СО ССЫЛКОЙ НА НЕГО

Данное приложение содержит код простейшего апплета и html-файл со ссылкой на него. Данные тексты могут использоваться как шаблоны при создании класса апплетов, следует только предварительно изменить в них слово *ИмяКласса* на имя создаваемого класса.

Листинг Java-файла:

```
import java.applet.*;
import java.awt.*;
//==============================================
// Основной класс для апплета ИмяКласса
//==============================================
public class ИмяКласса extends Applet
{
        //-----------------------------------------------
        public ИмяКласса ()
        {// Сделать: Добавьте сюда код конструктора
        }
        //-----------------------------------------------
        public String getAppletInfo()
        {return "Name: Applet\r\n" +"";
        }
        //-----------------------------------------------
        public void init()
        {resize(320, 240); // установка размера окна апплета
                // Сделать: Добавьте сюда код инициализации
        }
        //-----------------------------------------------
        public void destroy()
        {// Сделать: Добавьте сюда код завершения работы апплета
        }
        //-----------------------------------------------
        public void paint(Graphics g)
        {// Сделать: Добавьте сюда код перерисовки окна апплета
        }
        //-----------------------------------------------
        public void start()
        {// Сделать: Добавьте сюда код, который должен
                // работать при запуске апплета
```

```
        }
        //-------------------------------------------------
        public void stop()
        {// Сделать: Добавьте сюда код, который должен
                // работать при остановке апплета
        }
        // Сделать: Добавьте сюда код, необходимый для работы
        // создаваемого специализированного апплета
  }
```

Листинг HTML-файла:

```
  <html>
        <head>
                <title> ИмяКласса </title>
        </head>
        <body>
                <hr>
                <applet
                        code= ИмяКласса.class
                        name= ИмяКласса
                        width=320
                        height=240 >
                </applet>
                <hr>
        </body>
  </html>
```

ПРИЛОЖЕНИЕ 2

JAVA-ФАЙЛ АППЛЕТА, ПРИНИМАЮЩЕГО ПАРАМЕТРЫ, И HTML-ДОКУМЕНТ СО ССЫЛКОЙ НА НЕГО

Данное приложение содержит код апплета, принимающего параметры, и html-файл со ссылкой на него. Данные тексты могут использоваться как шаблоны при создании класса апплетов, следует только предварительно изменить в них слово ИмяКласса на имя создаваемого класса, а также проверить соответствие передаваемых и принимаемых параметров.

Листинг Java-файла:

```
  import java.applet.*;
  import java.awt.*;
  //==============================================
  // Основной класс для апплета ИмяКласса
  //==============================================
  public class ИмяКласса extends Applet
```

```
{
        // Поля класса для хранения значений параметров
        //---------------------------------------------------
        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 ИмяКласса()
        {       // Сделать: Добавьте сюда код конструктора
        }
        //---------------------------------------------------
        public String getAppletInfo()
        {       return "Name: Applet\r\n" +"";
        }
        //---------------------------------------------------
        public String[][] getParameterInfo()
        {       String[][] info =
                {
                { PARAM_String_1, "String", "Parameter description" },
                { PARAM_String_2, "String", "Parameter description" },
                }
                return info;
        }
        //---------------------------------------------------
        public void init()
        {       String param;
                // Параметр с именем String_1
                param = getParameter(PARAM_String_1);
                if (param != null)  m_String_1 = param;
                // Параметр с именем String_2
                param = getParameter(PARAM_String_2);
                if (param != null)  m_String_2 = param;
                //-----------------------------------------------
                resize(320, 240);
                // Сделать: Добавьте сюда код инициализации
        }
        //---------------------------------------------------
```

```
        public void destroy()
        {// Сделать: Добавьте сюда код завершения работы апплета
        }
        //-------------------------------------------------------
        public void paint(Graphics g)
        {// Сделать: Добавьте сюда код перерисовки окна апплета
        }
        //-------------------------------------------------------
        public void start()
        {// Сделать: Добавьте сюда код, который должен
                // работать при запуске апплета
        }
        //-------------------------------------------------------
        public void stop()
        {// Сделать: Добавьте сюда код, который должен
                // работать при остановке апплета
        }
        // Сделать: Добавьте сюда код, необходимый для работы
        // создаваемого специализированного апплета
  }
```

Листинг HTML-файла:

```
  <html>
        <head>
                <title> ИмяКласса </title>
        </head>
        <body>
                <hr>
                <applet
                        code= ИмяКласса.class
                        name= ИмяКласса
                        width=320
                        height=240 >
                        <param name=String_1 value="First string">
                        <param name=String_2 value="Second string">
                </applet>
                <hr>
                </body>
  </html>
```

ПРИЛОЖЕНИЕ 3

JAVA-ФАЙЛ АППЛЕТА, ОБРАБАТЫВАЮЩЕГО ПРОСТЫЕ СОБЫТИЯ МЫШИ, И HTML-ДОКУМЕНТ СО ССЫЛКОЙ НА НЕГО

Данное приложение содержит код апплета, обрабатывающего простые события мыши, и html-файл со ссылкой на него. Данные тексты могут использоваться как шаблоны при создании класса апплетов, следует только предварительно изменить в них слово ИмяКласса на имя создаваемого класса.

Листинг Java-файла:

```
import java.applet.*;
import java.awt.*;
//==============================================
// Основной класс для апплета ИмяКласса
//==============================================
public class ИмяКласса extends Applet
{
        //-------------------------------------------------
        public ИмяКласса()
        {// Сделать: Добавьте сюда код конструктора
        }
        //-------------------------------------------------
        public String getAppletInfo()
        {return "Name: Applet\r\n" +"";
        }
        //-------------------------------------------------
        public void init()
        {resize(320, 240); // установка размера окна апплета
                // Сделать: Добавьте сюда код инициализации
        }
        //-------------------------------------------------
        public void destroy()
        {// Сделать: Добавьте сюда код завершения работы апплета
        }
        //-------------------------------------------------
        public void paint(Graphics g)
        {// Сделать: Добавьте сюда код перерисовки окна апплета
        }
        //-------------------------------------------------
        public void start()
        {// Сделать: Добавьте сюда код, который должен
                // работать при запуске апплета
```

```
}
//---------------------------------------------------
public void stop()
{// Сделать: Добавьте сюда код, который должен
        // работать при остановке апплета
}
//---------------------------------------------------
public boolean mouseDown(Event evt, int x, int y)
{// Сделать: Добавьте сюда код, который должен
        // работать при нажатии клавиши мыши
        return true;
}
//---------------------------------------------------
public boolean mouseUp(Event evt, int x, int y)
{       // Сделать: Добавьте сюда код, который должен
        // работать при отпускании клавиши мыши
        return true;
}
//---------------------------------------------------
public boolean mouseDrag(Event evt, int x, int y)
{// Сделать: Добавьте сюда код, который должен
// работать при перемещении мыши с нажатой клавишей
        return true;
}
//---------------------------------------------------
public boolean mouseMove(Event evt, int x, int y)
{// Сделать: Добавьте сюда код, который должен
// работать при перемещении мыши с ненажатой клавишей
        return true;
}
//---------------------------------------------------
public boolean mouseEnter(Event evt, int x, int y)
{// Сделать: Добавьте сюда код, который должен
// работать при входе указателя мыши в окно апплета
        return true;
}
//---------------------------------------------------
public boolean mouseExit(Event evt, int x, int y)
{// Сделать: Добавьте сюда код, который должен
// работать при выходе указателя мыши из окна апплета
```

```
            return true;
        }
        // Сделать: Добавьте сюда код, необходимый для работы
        // создаваемого специализированного апплета
    }
```

Листинг HTML-файла:

```
    <html>
        <head>
            <title> ИмяКласса </title>
        </head>
        <body>
            <hr>
            <applet
                code= ИмяКласса.class
                name= ИмяКласса
                width=320
                height=240 >
            </applet>
            <hr>
        </body>
    </html>
```

ПРИЛОЖЕНИЕ 4

JAVA-ФАЙЛЫ АППЛЕТА ДВОЙНОГО НАЗНАЧЕНИЯ И HTML-ДОКУМЕНТ СО ССЫЛКОЙ НА НЕГО

Данное приложение содержит код апплета двойного назначения и html-файл со ссылкой на него. Данные тексты могут использоваться как шаблоны при создании класса апплетов, следует только предварительно изменить в них слово ИмяКласса на имя создаваемого класса.

Листинг Java-файла класса апплета:

```
    import java.applet.*;
    import java.awt.*;
    import MainWndFrame; // импорт класса рамки (фрейма)
    //=============================================
    // Основной класс для апплета ИмяКласса
    //=============================================
    public class ИмяКласса extends Applet
    {
        // Признак режима работы программы:
        // true/false - приложение/апплет
```

```
//---------------------------------------
private boolean m_fStandAlone = false;
//---------------------------------------
public static void main(String args[])
{       // Создать рамку (фрейм) для апплета
        MainWndFrame frame = new MainWndFrame("Title");
        // До изменения размеров фрейма отобразить его.
        // Это необходимо для того, чтобы метод insert()
        // выдавал правильные значения
        frame.show(); frame.hide();
        frame.resize(frame.insets().left + frame.insets().right + 320,
                frame.insets().top + frame.insets().bottom + 240);
// Создание объекта апплета, связывание апплета и фрейма
        ИмяКласса applet_Combi = new ИмяКласса();
        frame.add("Center", applet_Combi);
        // Установление признака режима работы - приложение
        applet_Combi.m_fStandAlone = true;
        // Вызов методов ааплета для его запуска
        applet_Combi.init();
        applet_Combi.start();
        // Отображение окна фрейма
        frame.show();
}
//---------------------------------------
public ИмяКласса()
{// Сделать: Добавьте сюда код конструктора
}
//---------------------------------------
public String getAppletInfo()
{       return "Name: CombiApplet\r\n" +"";
}
//---------------------------------------
public void init()
{       resize(320, 240);
// Сделать: Добавьте сюда код инициализации
}
//---------------------------------------
public void destroy()
{// Сделать: Добавьте сюда код завершения работы апплета
}
```

```
        //-----------------------------------------
        public void paint(Graphics g)
        {// Сделать: Добавьте сюда код перерисовки окна апплета
        }
        //-----------------------------------------
        public void start()
        {// Сделать: Добавьте сюда код, который должен
                // работать при запуске апплета
        }
        //-----------------------------------------
        public void stop()
        {// Сделать: Добавьте сюда код, который должен
                // работать при остановке апплета
        }
        // Сделать: Добавьте сюда код, необходимый для работы
        // создаваемого специализированного апплета
}
```

Листинг Java-файла класса фрейма для апплета:

```
import java.awt.*;
//=============================================
// Этот класс действует как окно, в котором отображается апплет,
// когда он запускается как обычное приложение
//=============================================
class MainWndFrame extends Frame
{
        // Конструктор класса
        //-----------------------------------------
        public MainWndFrame(String str)
        {       super (str);
                // Сделать: Добавьте сюда код конструктора
        }
        //-----------------------------------------
        public boolean handleEvent(Event evt)
        {       switch (evt.id)
                {// при закрытии окна завершается работа приложения
                        //------------------------------
                        case Event.WINDOW_DESTROY:
                // Сделать: Добавьте сюда код, который должен
                                // работать при остановке приложения
                                dispose(); // удаление окна
```

```
                    System.exit(0); // завершение приложение
                    return true;
                default:
                    // передача сообщения на обработку
                    // методу базового класса
                    return super.handleEvent(evt);
            }
        }
    }
```

Листинг HTML-файла:

```
<html>
    <head>
        <title> ИмяКласса </title>
    </head>
    <body>
        <hr>
        <applet
            code= ИмяКласса.class
            name= ИмяКласса
            width=320
            height=240 >
        </applet>
        <hr>
    </body>
</html>
```

ПРИЛОЖЕНИЕ 5

JAVA-ФАЙЛЫ АППЛЕТА ДВОЙНОГО НАЗНАЧЕНИЯ, ОБРАБАТЫВАЮЩЕГО СООБЩЕНИЯ ОТ МЫШИ, И HTML-ДОКУМЕНТ СО ССЫЛКОЙ НА НЕГО

Данное приложение содержит код апплета двойного назначения, обрабатывающего сообщения от мыши, и html-файл со ссылкой на него. Данные тексты могут использоваться как шаблоны при создании класса апплетов, следует только предварительно изменить в них слово ИмяКласса на имя создаваемого класса.

Листинг Java-файла класса апплета:

```
import java.applet.*;
import java.awt.*;
import MainWndFrame; // импорт класса рамки (фрейма)
//=============================================
// Основной класс для апплета ИмяКласса
//=============================================
```

```
public class ИмяКласса extends Applet
{
	// Признак режима работы программы:
	// true/false - приложение/апплет
	//---------------------------------------------
	private boolean m_fStandAlone = false;
	//---------------------------------------------
	public static void main(String args[])
	{	// Создать рамку (фрейм) для апплета
		MainWndFrame frame = new MainWndFrame("Title");
		// До изменения размеров фрейма отобразить его.
		// Это необходимо для того, чтобы метод insert()
		// выдавал правильные значения
		frame.show(); frame.hide();
		frame.resize(frame.insets().left + frame.insets().right + 320,
				frame.insets().top + frame.insets().bottom + 240);
	// Создание объекта апплета, связывание апплета и фрейма
		ИмяКласса applet_Combi = new ИмяКласса();
		frame.add("Center", applet_Combi);
		// Установление признака режима работы - приложение
		applet_Combi.m_fStandAlone = true;
		// Вызов методов ааплета для его запуска
		applet_Combi.init();
		applet_Combi.start();
		// Отображение окна фрейма
		frame.show();
	}
	//---------------------------------------------
	public ИмяКласса()
	{	// Сделать: Добавьте сюда код конструктора
	}
	//---------------------------------------------
	public String getAppletInfo()
	{	return "Name: CombiApplet\r\n" +"";
	}
	//---------------------------------------------
	public void init()
	{	resize(320, 240);
		// Сделать: Добавьте сюда код инициализации
	}
```

```
//---------------------------------------------
public void destroy()
{// Сделать: Добавьте сюда код завершения работы апплета
}
//---------------------------------------------
public void paint(Graphics g)
{// Сделать: Добавьте сюда код перерисовки окна апплета
}
//---------------------------------------------
public void start()
{// Сделать: Добавьте сюда код, который должен
        // работать при запуске апплета
}
//---------------------------------------------
public void stop()
{// Сделать: Добавьте сюда код, который должен
        // работать при остановке апплета
}
//---------------------------------------------
public boolean mouseDown(Event evt, int x, int y)
{// Сделать: Добавьте сюда код, который должен
        // работать при нажатии клавиши мыши
        return true;
}
//---------------------------------------------
public boolean mouseUp(Event evt, int x, int y)
{// Сделать: Добавьте сюда код, который должен
        // работать при отпускании клавиши мыши
        return true;
}
//---------------------------------------------
public boolean mouseDrag(Event evt, int x, int y)
{// Сделать: Добавьте сюда код, который должен
// работать при перемещении мыши с нажатой клавишей
        return true;
}
//---------------------------------------------
public boolean mouseMove(Event evt, int x, int y)
{// Сделать: Добавьте сюда код, который должен
// работать при перемещении мыши с ненажатой клавишей
```

```
            return true;
        }
        //---------------------------------------------
        public boolean mouseEnter(Event evt, int x, int y)
        {// Сделать: Добавьте сюда код, который должен
            // работать при входе указателя мыши в окно апплета
            return true;
        }
        //---------------------------------------------
        public boolean mouseExit(Event evt, int x, int y)
        {// Сделать: Добавьте сюда код, который должен
        // работать при выходе указателя мыши из окна апплета
            return true;
        }
        // Сделать: Добавьте сюда код, необходимый для работы
        // создаваемого специализированного апплета
    }
```

Листинг Java-файла класса фрейма для апплета:

```
    import java.awt.*;
    //=============================================
    // Этот класс действует как окно, в котором отображается апплет,
    // когда он запускается как обычное приложение
    //=============================================
    class MainWndFrame extends Frame
    {
        // Конструктор класса
        //---------------------------------------------
        public MainWndFrame(String str)
        {   super (str);
            // Сделать: Добавьте сюда код конструктора
        }
        //---------------------------------------------
        public boolean handleEvent(Event evt)
        {   switch (evt.id)
            {// при закрытии окна завершается работа приложения
                //-----------------------------
                case Event.WINDOW_DESTROY:
            // Сделать: Добавьте сюда код, который должен
                    // работать при остановке приложения
                    dispose(); // удаление окна
```

```
                                        System.exit(0); // завершение приложение
                                        return true;
                                default:
                                        // передача сообщения на обработку
                                        // методу базового класса
                                        return super.handleEvent(evt);
                        }
                }
        }
```

Листинг HTML-файла:

```
<html>
        <head>
                <title> ИмяКласса </title>
        </head>
        <body>
                <hr>
                <applet
                        code= ИмяКласса.class
                        name= ИмяКласса
                        width=320
                        height=240 >
                </applet>
                <hr>
        </body>
</html>
```

ПРИЛОЖЕНИЕ 6

JAVA-ФАЙЛ АППЛЕТА ДВОЙНОГО НАЗНАЧЕНИЯ, РЕАЛИЗУЮЩЕГО ИНТЕРФЕЙС RUNNABLE, И HTML-ДОКУМЕНТ СО ССЫЛКОЙ НА НЕГО

Данное приложение содержит код апплета двойного назначения, реализующего интерфейс Runnable, и html-файл со ссылкой на него. Данные тексты могут использоваться как шаблоны при создании класса апплетов, следует только предварительно изменить в них слово ИмяКласса на имя создаваемого класса.

Листинг Java-файла класса апплета:

```
import java.applet.*;
import java.awt.*;
import MainWndFrame; // импорт класса рамки (фрейма)
//=============================================
// Основной класс для апплета ИмяКласса
//=============================================
```

```
public class ИмяКласса extends Applet implements Runnable
{
        // задача, которая будет одновременно с апплетом
        private Thread m_Task = null;
        // Признак режима работы программы:
        // true/false - приложение/апплет
        //----------------------------------------------
        private boolean m_fStandAlone = false;
        //----------------------------------------------
        public static void main(String args[])
        {// Создать рамку (фрейм) для апплета
                MainWndFrame frame = new MainWndFrame("Title");
                // До изменения размеров фрейма отобразить его.
                // Это необходимо для того, чтобы метод insert()
                // выдавал правильные значения
                frame.show(); frame.hide();
                frame.resize(frame.insets().left + frame.insets().right + 320,
                        frame.insets().top + frame.insets().bottom + 240);
        // Создание объекта апплета, связывание апплета и фрейма
                ИмяКласса applet_Combi = new ИмяКласса();
                frame.add("Center", applet_Combi);
                // Установление признака режима работы - приложение
                applet_Combi.m_fStandAlone = true;
                // Вызов методов ааплета для его запуска
                applet_Combi.init();
                applet_Combi.start();
                // Отображение окна фрейма
                frame.show();
        }
        //----------------------------------------------
        public ИмяКласса()
        {// Сделать: Добавьте сюда код конструктора
        }
        //----------------------------------------------
        public String getAppletInfo()
        {       return "Name: CombiApplet\r\n" +"";
        }
        //----------------------------------------------
        public void init()
        {       resize(320, 240);
```

```
// Сделать: Добавьте сюда код инициализации
}
//-----------------------------------------------
public void destroy()
{// Сделать: Добавьте сюда код завершения работы апплета
}
//-----------------------------------------------
public void paint(Graphics g)
{// Сделать: Добавьте сюда код перерисовки окна апплета
        g.drawString("Running: " + Math.random(), 10, 20);
}
//-----------------------------------------------
public void start()
{       // если задача еще не создана, апплет создает
        // новую задачу как объект класса Thread
        if (m_Task == null)
        {       m_Task = new Thread(this); // создание задачи
                m_Task.start(); // запуск задачи
        }
        // Сделать: Добавьте сюда код, который должен
        // работать при запуске апплета
}

//-----------------------------------------------
public void stop()
{       // когда пользователь покидает страницу,
        // метод stop() класса Thread останавливает задачу
        if (m_Task != null) // если задача была создана
        {       m_Task.stop(); //остановка задачи
                m_Task = null; // сброс ссылки на задачу
        }
        // Сделать: Добавьте сюда код, который должен
        // работать при остановке апплета
}
// Метод, который работает в рамках отдельной задачи.
// Он вызывает периодическое обновление окна апплета
//-----------------------------------------------
public void run()
{       // выполняем обновление окна в бесконечном цикле
        while (true)
```

```
                {       try
                        {       // вызов функции обновления
                                repaint();
                // Сделать: Добавьте сюда код, который должен
                                // здесь работать в рамках задачи
                                //выполнение небольшой задержки
                                Thread.sleep(50);
                        }
                        catch (InterruptedException e)
                {/// Сделать: Добавьте сюда код, который должен
                        // здесь работать при генерации исключения
                        // если при выполнении задержки произошло
                        // исключение, останавливаем работу апплета
                                stop();
                        }
                }
        }
        // Сделать: Добавьте сюда код, необходимый для работы
        // создаваемого специализированного апплета
}
```

Листинг Java-файла класса фрейма для апплета:

```
import java.awt.*;
//=============================================
// Этот класс действует как окно, в котором отображается апплет,
// когда он запускается как обычное приложение
//=============================================
class MainWndFrame extends Frame
{
        // Конструктор класса
        //---------------------------------------------
        public MainWndFrame(String str)
        {       super (str);
        // Сделать: Добавьте сюда код конструктора
        }
        //---------------------------------------------
        public boolean handleEvent(Event evt)
        {       switch (evt.id)
                {/// при закрытии окна завершается работа приложения
                        //-------------------------------
                        case Event.WINDOW_DESTROY:
```

```
                    // Сделать: Добавьте сюда код, который должен
                                    // работать при остановке приложения
                                    dispose(); // удаление окна
                                    System.exit(0); // завершение приложение
                                    return true;
                            default:
                                    // передача сообщения на обработку
                                    // методу базового класса
                                    return super.handleEvent(evt);
                    }
            }
    }
```

Листинг HTML-файла:

```
    <html>
            <head>
                    <title> ИмяКласса </title>
            </head>
            <body>
                    <hr>
                    <applet
                            code= ИмяКласса.class
                            name= ИмяКласса
                            width=320
                            height=240 >
                    </applet>
                    <hr>
            </body>
    </html>
```

ПРИЛОЖЕНИЕ 7

САМОСТОЯТЕЛЬНОЕ ГРАФИЧЕСКОЕ JAVA-ПРИЛОЖЕНИЕ

Данное приложение содержит код самостоятельного графического Java-приложения. Данные тексты могут использоваться как шаблоны при создании класса приложения, следует только предварительно изменить в них слово ИмяКласса на имя создаваемого класса.

Листинг Java-файла класса приложения:

```
    import java.awt.*;
    //=============================================
    // Основной класс для приложения ИмяКласса
    //=============================================
    public class ИмяКласса
```

```
{       public static void main(String args[])
        {       // Создать рамку (фрейм)
                MainWndFrame frame = new MainWndFrame("Title");
                // Отображение окна фрейма
                frame.show();
        }
        // Сделать: Добавьте сюда код, необходимый для работы
        // создаваемого специализированного приложения
}
//=============================================
// Класс главного фрейма для приложения
//=============================================
class MainWndFrame extends Frame
{
        // Конструктор класса
        //-------------------------------------------
        public MainWndFrame(String str)
        {       super (str);
                // Сделать: Добавьте сюда код конструктора
                resize(320,240); // задание необходимых размеров окна
// Сделать: здесь можно определить различные параметры фрейма,
// например, форму курсора, меню, добавить компоненты и др.
        }
        //-------------------------------------------
        public boolean handleEvent(Event evt) // обработчик событий
        {       switch (evt.id)
                {// при закрытии окна завершается работа приложения
                        //---------------------------
                        case Event.WINDOW_DESTROY:
                // Сделать: Добавьте сюда код, который должен
                                // работать при остановке приложения
                                dispose(); // удаление окна
                                System.exit(0); // завершение приложение
                                return true;
                        default:
                                // передача сообщения на обработку
                                // методу базового класса
                                return super.handleEvent(evt);
                }
        }       }
```

Библиографический список

1. Аарон И. Волш. Основы программирования на Java для World Wide Web. Киев: "Диалектика", 1996.
2. Вебер Д.. Технология Java в подлиннике. С.-П.: "BHV-СанктПетербург", 1997.
3. Гослинг Дж., Арнольд К. Язык программирования Java. С.-П.: Издательский дом "Питер", 1997.
4. Джамса К. Изучи сам Java сегодня. Мн: ООО "Попурри", 1996.
5. Нотон П. Java: справочное руководство. М.: Восточная книжная компания, 1996.
6. Родли Дж. Создание Java-апплетов. К: НИПФ "ДиаСофт Лтд", 1996.
7. Томас М., Пател П., Хадсон А., Болл Д. Секреты программирования для Internet на Java. С.-П.: Издательский дом "Питер", 1997.
8. Фролов А.В., Фролов Г.В. Библиотека системного программиста, тт.30, 32. М.: "ДИАЛОГ-МИФИ", 1997.
9. Холзнер С. Visual J++ 1.1 с самого начала. С.-П.: Издательский дом "Питер", 1997.
10. Эферган М. Java: справочник. С.-П.: Издательский дом "Питер", 1997.

СОДЕРЖАНИЕ

Ирина Михайловна Акилова,
доцент кафедры "Информационных и управляющих систем",

Лариса Владимировна Чепак

доцент кафедры "Информационных и управляющих систем", к.т.н.

Елена Николаевна Архипова
ведущий программист отдела информационных систем и корпоративных приложений Владивостокского государственного университета экономики и сервиса

Технология программирования. Программирование на языке JAVA. Учебное пособие.
