Бондарев В. М.

Г 32.162.(07)
Б 81

Учебное пособие по программированию на JAVA

519.682(07)
Б81

В. М. Бондарев

Учебное пособие по программированию на Java

БІБЛІОТЕКА
ХНУРЕ
ІНВ. №

Харьков «СМИТ»
2003

Содержание

Введение....4

Часть 1. Основы программирования на Java....5

1. Первое знакомство....5
2. Операции и операторы....9
3. Массивы....18
4. Классы....21
5. Наследование и полиморфизм....25
6. Строки....33
7. Обработка исключений....41
8. Интерфейсы и внутренние классы....47
9. Ввод и вывод....55
10. Контейнерные классы....66

Часть 2. Продвинутые средства языка....74

1. Ассоциативные контейнеры....74
2. Многопоточное программирование....81
3. Взаимодействие потоков....90
4. Сериализация объектов....101
5. Рефлексия....105
6. Загрузка классов....109
7. Системное программирование....113
8. Программный интерфейс JDBC....119
9. Сжатие и разбор данных....130
10. Документация....136

Часть 3. Графический интерфейс пользователя....142

1. Приложения с оконным интерфейсом....142
2. Обработка событий в AWT....149
3. Библиотека SWING....155
4. Компоненты и контейнеры....160
5. Некоторые компоненты библиотеки SWING....172
6. Текстовые поля и модель документов....180
7. Формы и диалоги....191
8. Рисование в SWING....201
9. Битовые изображения....209
10. Отображение таблиц....217
11. Отображение деревьев....225
12. Апплеты....232

Часть 4. Разработка Internet-приложений....244

1. Протокол HTTP....244
2. Что такое сервлет....251
3. Запросы и ответы....261
4. Сеансы и переадресация....269
5. Апплеты в роли клиентов сервлетов....276
6. Серверные страницы JAVA....282
7. Переадресация и включение....290
8. Использование BEAN-компонентов....292

Введение

В истории программирования нет другого языка, популярность которого росла так же быстро, как росла популярность Java. Не прошло пяти лет с момента появления Java, как он стал обязательной частью фундаментального образования программиста и предметом для изучения в вузах и техникумах. Множество программистов, ранее писавших программы на Бейсике, Паскале, C++, захотели изучить его и связать с ним свое программистское будущее.

Помимо независимости от программно-аппаратной платформы, Java привлекает простотой и продуманностью своих библиотек. И это понятно, ведь в их создании участвует коллективный разум лучших программистов. Спецификации новых программных интерфейсов Java открыты для обсуждения и, прежде чем станут стандартом, подвергаются всесторонней проверке и анализу.

Книга состоит из четырех частей:

1. Основы программирования на Java;
2. Продвинутые средства языка;
3. Графический интерфейс пользователя;
4. Разработка Internet-приложений.

Первая часть посвящена основам языка, его объектной модели и нескольким программным пакетам, без которых нельзя написать мало-мальски интересную программу.

Во второй части рассматриваются инструменты, необходимые для создания реальных, программ: контейнеры, многопоточность, средства доступа к базам данных.

Третья часть целиком посвящена разработке настольных приложений с графическим интерфейсом пользователя на основе библиотек AWT и Swing.

Четвертая часть необходима тем, кто намерен разрабатывать сетевые приложения для Интернет и локальных сетей.

Книга не предназначена для первоначального знакомства с программированием и предполагает, что читатель уже пробовал писать программы на каком-то из процедурных языков, но на каком именно особого значения не имеет.

Часть 1. Основы программирования на Java

1. ПЕРВОЕ ЗНАКОМСТВО

1.1. История возникновения

Java был задуман Джеймсом Гослингом, Патриком Ноутоном, Крисом Вартом, Эдом Франком и Майком Шериданом из компании Sun Microsystems Inc. в 1991 г. Через 1,5 года была создана первая рабочая версия, которая называлась Oak. В 1995 г. язык был переименован в Java.

Первым побуждением для создания Java был не Интернет, а потребность в независимом от платформы языке для программирования бытовых электронных устройств (микроволновых печей, стиральных машин, дистанционных пультов управления, мобильных устройств и т.п.).

Когда Java уже сформировался, появились Internet и всемирная паутина WWW, которым был необходим независимый от платформы полнофункциональный язык программирования. Это заставило создателей Java сосредоточиться не на микропроцессорах, а на Internet-программировании.

Своим успехом язык Java обязан Internet.

1.2. Области применения Java

- Локальные консольные приложения.
- Локальные приложения с графическим интерфейсом.
- Разработка программных компонентов.
- Работа с базами данных.
- Internet-программирование на стороне клиента - апплеты.
- Internet-программирование на стороне сервера – сервлеты, серверные страницы JSP.
- Разработка масштабируемых серверных приложений уровня предприятия – EJB.

Технология Java поддерживается и совершенствуется с помощью механизма, известного как Java Community Process (JCP), и представляющего взаимодействие группы из более чем 400 компаний, организаций и частных лиц. JCP предполагает создание групп экспертов, которые сотрудничают в целях определения новых спецификаций и усовершенствования уже существующих. Система принятия решений с

помощью голосования гарантирует, что Java остается единой и общей платформой для всех компаний.

Java-приложения могут выполняться в любых операционных системах: системах уровня предприятия, таких, как Unix, Linux, OS/390, Windows 2000, или HP-UX; операционных системах для настольных компьютеров, таких как MacOS, Windows или Linux; а также в операционных системах для мобильных устройств, таких как PalmOS или Symbian's EPOC.

Технология Java является открытой и построена на внутриотраслевых стандартах для программного обеспечения. Любой желающий может загрузить и изучать код Java-платформы.

Java немного похож на C++, но не совместим с ним ни в какую сторону. Java не должен заменить C++, а будет сосуществовать с ним, т.к. предназначен для решения иных проблем.

1.3. Особенности языка

- Простой (нет указателей, освобождения памяти, перегрузки операций, шаблонов, множественного наследования).
- Объектно-ориентированный, в Java даже нет глобальных переменных или функций, есть только поля и методы классов.
- Архитектурно-нейтральный, т.е. не ориентирован на какую-то одну аппаратную или программную архитектуру.
- Устойчивый (проверяет выход за границу массива, не только предоставляет аппарат исключений, но и требует от программиста их обработки).
- Многопоточный, средства работы с потоками встроены в язык.
- Интерпретируемый, выполнение программы происходит путем интерпретации частично откомпилированного кода.
- Распределенный (позволяет выполнять удаленные вызовы методов).
- Динамический (использует информацию о типах и отражение).

Главной же особенностью Java является независимость программ от той программно-аппаратной среды, в которой они будут выполняться. Это достигается разделением процесса исполнения программы на два этапа. На первом этапе (компиляция) исходный код переводится в нормализованную форму - байт-код. На втором этапе (интерпретация) байт-код мелкими порциями превращается в инструкции процессора, и эти инструкции тут же выполняются. Для каждой известной платформы существует свой интерпретатор байт-кода, который называют виртуальной машиной Java - JVM (Java Virtual Machine).

1.4. Java Development Kit

Так называют инструментарий разработчика Java-приложений. Он состоит из библиотек, которые может и должен использовать всякий программист, и набора программ-утилит, таких как компилятор (javac), интерпретатор (java), отладчик (jdb), архиватор (jar), документатор (javadoc) и других.

В эволюции Java имеется несколько поворотных пунктов, которые можно представить как различные версии JDK.

Версии JDK

JDK 1.0 - первая версия языка.
JDK 1.1 – новая обработка событий, вложенные классы.
JDK 1.2 (Java 2) - появились сервлеты, новые контейнеры.
JDK 1.3 - самая распространенная в настоящее время.
JDK 1.4 - последняя версия. В стандартной редакции содержится 2738 классов и интерфейсов (для сравнения JDK 1.0 имел всего 212 классов и интерфейсов).

В версии JDK 1.2 изменения были столь значительны, что, начиная с нее, язык стали называть Java 2.

В настоящее время имеется три редакции JDK:
J2ME - Micro - ограниченная версия Java для устройств.
J2SE - Standard - последняя версия 1.4.
J2EE - Enterprise - последняя версия 1.3.1.

Все это можно бесплатно загрузить с сайта компании Sun Microsystems http://java.sun.com, причем там имеются варианты JDK для всех основных операционных систем.

1.5. Установка JDK

Для установки JDK достаточно:
1) разместить утилиты JDK в каталоге, например, C:\JDK\BIN;
2) добавить путь к этим утилитам в переменную среды PATH;
3) разместить библиотечные классы JDK в каталоге, например, C:\JDK\LIB;
4) добавить путь к этим классам в переменную среды CLASSPATH.

Но лучше воспользоваться специальной установочной программой, которая имеется в составе JDK.

1.6. Hello Java!

Для того, чтобы проверить правильность установки JDK, напишем и выполним простую программу, которая выдаст на экран сообщение "Hello Java!"

Сделаем это по шагам.

Шаг 1. Подготовим собственно программу в текстовом файле Hello.java.

⇓

```
public class Hello
{
  public static void main(String[] args) {
      System.out.println("Hello Java!");
  }
}
```

Шаг 2. Создадим рабочий каталог, сделаем его текущим и поместим туда файл с программой.

Шаг 3. Откомпилируем приложение командой

>javac Hello.java

Имя файла совпадает с именем класса и чувствительно к регистру, т.е. прописные и строчные буквы различаются. В результате в рабочем каталоге должен появиться файл Hello.class, который содержит нашу программу, откомпилированную в так называемый байт-код.

Шаг 4. Выполним интерпретацию полученного байт-кода командой

>java Hello

Замечание. Чтобы интерпретатор нашел класс Hello, путь к классу надо указать в командной строке (опция –cp) или добавить в переменную среды CLASSPATH. Первое предпочтительней:

>java –cp D:\WORKDIR Hello

Если класс, как в нашем примере, находится в текущем каталоге, достаточно набрать

>java –cp ; Hello

Точка с запятой является разделителем путей в OS Windows. Одинокий разделитель и символизирует текущий каталог.

Цикл компиляции и выполнения обычно приходится повторять многократно, поэтому пользователям MS Windows может облегчить жизнь пакетный файл, подобный этому.

⇓

```
del  %1.class
javac %1.java
java -cp ;  %1
pause
```

1.7. Литература по Java

- П. Ноутон, Г. Шилд. Java-2. - СПб.: БХВ-Петербург, 2000. - 1072 с. – С языком Java лучше всего знакомиться по этой книге.
- Брюс Эккель. Философия Java. Библиотека программиста. - СПб.: Питер, 2001. - 880 с. – то же, но для людей с философским складом ума. Помимо языка, затрагиваются некоторые технологии, впрочем, весьма поверхностно.
- К. Арнолд, Дж. Гослинг, Д. Холмс. Язык программирования Java, 3-е изд. - М.: Издательский дом "Вильямс", 2001. - 624 с. – руководство по языку, которое лучше читать, когда первое знакомство с Java уже состоялось.

1.8. Задание

Установить на компьютере JDK версии не ниже 1.3. или убедиться, что они там уже установлены. Написать, откомпилировать и выполнить пробную программу, которая выводит на экран фамилию и имя автора.

2. ОПЕРАЦИИ И ОПЕРАТОРЫ

2.1. Простые типы данных

В языке Java есть две разновидности данных: простые типы и ссылочные. О ссылочных типах речь пойдет позже, а простые типы языка сведены в таблицу.

Тип	Значение	Размер	Константы
boolean	true или false	1 бит	true, false
char	символ Unicode	2 байта	‘a’, \u00ff
byte	целое со знаком	1 байт	
short	целое со знаком	2 байта	
int	целое со знаком	4 байта	
long	целое со знаком	8 байтов	10L, 10l
float	число с п.т. в формате IEEE 754	4 байта	20F, 20f
double	число с п.т. в формате IEEE 754	8 байтов	30D, 30d

Размер каждого типа определен в самом языке и не зависит от реализации. Сделаем несколько замечаний о некоторых типах.

Логические значения

boolean не являются целыми и не могут быть преобразованы в числовой тип или из него.

Кодировка символов

Символы в Java являются 16-разрядными символами Unicode. Первые 256 символов Unicode совместимы с кодировкой ISO8859-1.

Поскольку не все символы можно отобразить на экране, применяются Escape-последовательности Unicode.

\uxxxx – где xxxx – четыре 16-ичных цифры – код символа.

Применяются также Escape-последовательности в стиле C:

\ooo, \t, \n, \", \\

где: ooo - означает трехзначное восьмеричное число, \t - символ табуляции, \n - перевод строки, \" изображает кавычку, а \\ - обратную косую черту.

Замечание. Escape-последовательности Unicode могут находиться в любом месте программы, а не только в строковых константах.

⇓

```
int \u0069;  // означает то же, что int i;
```

Преобразование символов в числа может давать отрицательные значения.

⇓

```
int n = (short)'\uf000';
```

Вещественные числа

Представлены в формате IEEE (Institute of Electrical and Electronics Engineers), который для вещественного числа float отводит 1 бит на знак, 8 битов на порядок и 23 бита на мантиссу. Порядок смещен на -127, т.е. нуль означает 2 в -127-й степени, а мантисса заключена в диапазоне от 1.0 до 2.0, причем старшая единица не записывается в разрядную сетку. Такое представление дает диапазон чисел от 3.4E-38 до 3.4E+38.

Числа формата double хранятся аналогичным образом, но распределение битов между порядком и мантиссой 11:52.

В результате выполнения операций над числами с плавающей точкой могут получиться особые значения:

- плюс бесконечность4
- минус бесконечность;
- минус ноль;
- не число (NaN – Not a Number).

Число NaN не сравнимо ни с чем, даже с самим собой.

При делении на ноль целых чисел генерируется исключение ArithmeticException (об исключениях будет сказано в разделе 7), при делении на ноль вещественных чисел исключение не возникает, а результат будет равен плюс или минус бесконечности.

2.2. Операции

Любая операция характеризуется приоритетом и ассоциативностью. Все операции вырабатывают значение.

Приоритет определяет порядок выполнения операций в выражении, когда нет скобок. Ассоциативность говорит, какая из двух рядом стоящих операций с равным приоритетом выполняется раньше, левая или правая. Например, операция вычитания левоассоциативна, поэтому

$$6 - 3 - 2 = (6 - 3) - 2 = 1.$$

Если бы операция вычитания была правоассоциативна, то

$$6 - 3 - 2 = 6 - (3 - 2) = 5.$$

Все операции, имеющиеся в Java, сведены в таблицу.

Приоритет	Оператор	Тип операнда	Ассоциативность	Название операции
1	++	арифметический	П	пре- и постинкремент
1	--	арифметический	П	пре- и постдекремент
1	+, -	арифметический	П	унарные + и -
1	~	целый	П	битовое дополнение
1	!	булев	П	отрицание
1	(type)	любой	П	приведение типа
2	*, /, %	арифметический	Л	умножение, деление, остаток от деления
3	+, -	арифметический	Л	сложение, вычитание
3	+	строковый	Л	конкатенация строк
4	<<	целый	Л	сдвиг влево

4	>>	целый	Л	сдвиг вправо с размножением знак. бита
4	>>>	целый	Л	сдвиг вправо с размножением нуля
5	<, <=	арифметический	Л	меньше, меньше или равно
5	>, >=	арифметический	Л	больше, больше или равно
5	instanceof	объект, тип	Л	сравнение типов
6	==	простой	Л	равенство
6	!=	простой	Л	неравенство
6	==	объект	Л	равенство ссылок
6	!=	объект	Л	неравенство ссылок
7	&	целый	Л	битовое AND
7	&	булев	Л	булево AND (по полной схеме)
8	^	целый	Л	битовое XOR
8	^	булев	Л	булево XOR (по полной схеме)
9	|	целый	Л	битовое OR
9	|	булев	Л	булево OR (по полной схеме)
10	&&	булев	Л	булево AND (по краткой схеме)
11	||	булев	Л	булево OR (по краткой схеме)
12	? :	булев, любой, любой	П	оператор условия
13	=	переменая, любой	П	присваивание
13	*=, += ...	переменная, любой	П	присваивание с операцией

Сделаем несколько замечаний относительно некоторых операций.

Деление целых чисел дает целый же результат.

⇓

```
System.out.println(12 / 5);   // напечатает 2.
```

Преобразование числовых типов выполняется неявно, если переводит более узкий тип в более широкий (расширяющее преобразование типа). Противоположное сужающее преобразование должно выполняться явно.

Исключение составляет случай, когда преобразуется небольшая целая константа к типу short, byte или char.

Обычные операторы сдвига (<< и >>) сохраняют знаковый бит. Оператор >>> знаковый бит не сохраняет, а заменяет нулем.

Примеры сдвигов.

⇓

```
int i = 2;
i << 1  = 4
i >> 1  = 1
i >>> 1 = 1

int i = -2;
i << 1  = -4
i >> 1  = -1
i >>> 1 = 2147483647
```

Последний результат объясняется тем, что отрицательные числа хранятся в Java в дополнительном коде (разряды соответствующего положительного числа инвертируются и к результату прибавляется 1).

⇓

```
 1 = <000…0001>
-1 = <111…1110> + <1> = <111…1111>
-2 = <111…1101> + <1> = <111…1110>
-2 >>  1 = <111…1111> = -1
-2 >>> 1 = <011…1111> = 2147483647
```

Операции деления нацело **x / y** и взятия остатка от деления **x % y** определены так, чтобы выполнялось тождество

(x / y) * y + x % y = x.

2.3. Комментарии

Java допускает комментарии в исходном коде программы:
многострочные в стиле языка C

⇓

```
/* Любое количество любых строк
лишь бы там не было сочетания
звездочки и косой черты  */
```

однострочные в стиле языка C++

⇓

```
// все написанное до конца строки - комментарий
```

комментарии документатора

⇓

```
/** Многострочный комментарий,
  * который войдет в
  * программный документ. */
```

2.4. Управляющие конструкции

Отметим, что управляющие конструкции Java схожи с операторами языка C, но есть некоторые отличия.

Оператор if / else

Схема условного оператора такова:

if (условие)
оператор 1
[else
оператор 2]

Смысл условного оператора такой же, как и в других языках программирования. В отличие от языка C, условие всегда должно иметь логическое значение, поэтому пишут

if (i == 0) …
а не
if (!i) …
как в C.

Операторы while и do / while

Схемы для оператора цикла следующие:

while (условие)
оператор;
и
do
оператор
while (условие);

Условием служит такое же логическое выражение, как в операторе **if.**

В первом цикле условие проверяется до выполнения вложенного оператора (цикл с предусловием), во втором - после (цикл с постусловием). Это ведет к тому, что вложенный оператор первого цикла может не

выполниться ни разу, а оператор второго цикла обязательно будет выполнен хотя бы раз.

Примеры циклов.

⇓

```
// Цикл с предусловием
float sum = 0, x = 1;
while(x < 100) {
  sum += 1 / x;
  x++;
}

// То же, но с постусловием
sum = 0; x = 1;
do {
  sum += 1 / x;
  x++;
while(x < 100)

// Идиома бесконечного цикла
while (true) {
  // оператор
}
```

Оператор for

Оператор for имитирует такой же оператор языка С.

for (выражение_инициализации; [выражение1] ; [выражение2])
оператор

Смысл оператора for передается следующим псевдокодом, который делает то же, что оператор for.

⇓

```
выражение_инициализации;
while (выражение1) {
  оператор
  выражение 2;
}
```

Примеры циклов.

⇓

```
// Делает то же, что цикл с предусловием из предыдущего примера
float sum = 0;
for (float x = 1; x < 100; x++ ) {
  sum += 1 / x;
}

// Идиома бесконечного цикла
for (;;) {
  // оператор
}
```

Операторы break и continue

Чтобы прервать последовательность повторений любого цикла в языке С используется оператор break.

Пример. Цикл прекратит повторяться, когда x превысит 99.

⇓

```
for (float x = 1; ; x++ ) {
  if (x > 99)
      break;
  sum += 1 / x;
}
```

Чтобы перейти к следующей итерации цикла, не завершая текущую, в язъыке С используется оператор continue.

Пример. Второй десяток слагаемых в сумму не войдет.

⇓

```
for (float x = 1; x < 100; x++ ) {
  if ( 10 < x && x < 21)
      continue;
  sum += 1 / x;
}
```

Оба эти оператора есть в языке Java, но, кроме того, для работы с вложенными циклами имеется вариант этих операторов с меткой. Метка L помечает блок кода, внутри которого располагаются операторы "break L;" или "continue L;".

Оператор "break L;" - передает управление оператору, следующему за помеченным блоком.

Оператор "continue L;" - продолжает помеченный цикл (инструкция continue используется только в циклах);

Пример. Применение оператора break для досрочного выхода из вложенных циклов.

⇓

```
for (int i1 = 0; i1 < 10; i1++) {
L: for (int i2 = 0; i2 < 10; i2++) {
     for (int i3 = 0; i3 < 10; i3++) {
       if (i3 == 6)
         break L;
     }
   }
   // Сюда попадет управление после выполнения оператора "break L;"
}
```

Оператор return

Обеспечивает возврат из вызываемой функции в вызывающую.

2.5. Задание

1. Написать программу, которая находит наибольший общий делитель двух целых положительных чисел.
2. Написать программу, которая находит сумму цифр произвольного целого числа.
3. Написать программу проверки того, что заданное число X – простое (т.е. не делится без остатка ни на какие числа, кроме себя и 1). Число X задавать в коде программы.
4. Сосчитать сумму ряда 1! - 2! + 3! – 4! + 5! - … + n! для заданного числа n > 0. Чем шире диапазон возможных значений n, тем лучше. Число n задавать в коде программы.
5. Подсчитать, сколько шестизначных цифр имеют равную сумму трех первых и трех последних цифр.

Замечание. Все задания, которые вы здесь видите, можно выполнять в форме функции main(), как в примере "Hello Java" из первого раздела.

⇓

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

БІБЛІОТЕКА
ХНУРЕ

```
        // Код, решающий задачу
        ...
        System.out.println(значение результата);
    }
}
```

3. МАССИВЫ

3.1. Ссылочные типы данных

Все типы, которые не являются простыми, называются ссылочными. Это оттого, что в переменных этих типов хранятся не сами значения, а только ссылки на них (ссылкой называется адрес памяти).

Ссылочные типы Java делятся на три группы: массивы, классы, строки.

При работе со ссылочными типами нужно помнить следующее.

- Операторы "=" и "==" присваивают и сравнивают ссылки, а не значения. Для копирования и сравнения значений используют другие методы.
- Значение простого типа не может быть преобразовано к ссылочному типу.
- Значение ссылочного типа не может быть преобразовано к простому типу.
- Специальное значение **null** представляет пустую ссылку.

3.2. Одномерные массивы

Чтобы воспользоваться массивом в программе, необходимо:

1) определить ссылку на массив элементов заданного типа;
2) разместить в памяти необходимое количество элементов.

После размещения в памяти элементы всегда инициализируются - по умолчанию или при помощи списка инициализации. Количество элементов массива не может быть изменено, но переменной можно присвоить ссылку на другой массив.

Примеры объявления и размещения массивов.

⇓

```
// Объявление ссылки на массив, размещение элементов в памяти и
// инициализация их нулями по умолчанию
byte[] buffer = new byte[1024];
```

```
// То же, но объявление массива похоже на C++
byte buffer[] = new byte[1024];

// Ссылка на массив только объявляется
float[] f;

// В памяти размещается массив и ссылка инициализируется его адресом
f = new float[300];

// Объявление, размещение и явная инициализация при помощи списка
int fib[] = {1, 2, 3, 5, 8, 13};
```

Размер массива хранится в поле **length**. Это единственное поле массива и оно - только для чтения.

Пример. Заполнить массив значениями.

⇓

```
for( int i = 0; i < m.length; i++)
  m[i] = i;
```

В Java при каждом обращении к элементу массива проверяются границы массива, и при их нарушении выбрасывается исключение ArrayIndexOutOfBoundsException.

3.3. Многомерные массивы

Многомерный массив – это массив, элементами которого служат массивы меньшей размерности, например, элементами двухмерного массива служат одномерные массивы.

Поскольку одномерный массив в Java представлен ссылкой, двухмерный массив является массивом ссылок на одномерные массивы.

Замечание. Это позволяет организовывать “непрямоугольные” массивы, которые, впрочем, редко находят применение.

Пример. Объявление многомерных массивов.

⇓

```
byte m2[][] = new byte [256][16];
byte m3[][][] = new byte [100][][];
```

Замечание. В Java при объявлении многомерных массивов не обязательно задавать сразу все размерности, можно только левые.

Многомерные массивы инициализируют при помощи вложенных списков инициализации.

Пример. Объявление, размещение и инициализация «треугольного» массива.

⇓

```
int m[][] = {{1}, {2,3}, {4,5,6}};
```

3.4. Массивы-параметры и возвращаемые значения

Массивы могут быть параметрами функций и теми значениями, которые функции возвращают.

Для примера определим функцию, которая принимает целый массив и возвращает другой массив, который является инвертированным аргументом (в инвертированном массиве элементы расположены в обратном порядке).

⇓

```
static int[] inversion (int[] a)
{
 int len = a.length;
 int[] b = new int[len];
 for (int i = 0; i < len; i++)
      b[i] = a[len - i + 1];
 return b;
}
```

3.5. Анонимные массивы

При вызове методов и передаче им параметров часто применяют анонимные, т.е. безымянные, массивы.

Вызовем функцию из предыдущего примера, дав ей анонимный массив в качестве аргумента.

⇓

```
int[] x = inversion(new int[] {1,2,3,4,5});
```

3.6. Задание

1. Разместить в памяти массив из 20 элементов и заполнить его рядом Фиббоначчи: 1, 1, 2, 3, 5, 8, 13, 21, …. В этом ряду каждое следующее число является суммой двух предыдущих.
2. Создать целый массив из 100 элементов и заполнить его простыми числами: 2, 3, 5, 7, 11, 13, 17, ….
3. Создать двухмерный массив символов и заполнить его буквами 'Ч' и 'Б' в шахматном порядке.
4. Создать целый шестимерный массив с двумя значениями в каждом измерении. Заполнить массив числами из начала натурального ряда: 1, 2, 3, …. Сказать, сколько потребуется чисел?
5. Создать "треугольный" массив из 10 строк и заполнить его биномиальными коэффициентами (треугольник Паскаля)

```
   1
  1 2 1
 1 3 3 1
1 4 6 4 1
```

4. КЛАССЫ

4.1. Классы и объекты

Мы предполагаем, что читатель уже знаком с основами объектно-ориентированного программирования, поэтому сосредоточимся на особенностях объектной модели языка Java.

Напомним, что класс это такой программный тип, который определяет не только данные, но и функции, применимые к этим данным. Говорят, что данные задают состояние объекта, а функции – его поведение.

Переменные такого программного типа называют экземплярами класса или объектами.

Объекты в Java всегда размещаются в динамической памяти при помощи оператора **new**.

Пример. Объявить класс Rectangle и создать объект этого класса.

⇓

```
class Rectangle {
  int x1, y1, x2, y2;
}
```

```
public class Try {
  public static void main(String[] args) {
      Rectangle r = new Rectangle();
      r.x1 = 10; r.y1 = 10; r.x2 = 100; r.y2 = 100;
  }
}
```

В файле может быть объявлено несколько классов, но только один со спецификатором public - открытый. Имя файла и имя открытого класса должны совпадать.

4.2. Конструкторы

В конструкторе принято выполнять только инициализацию полей объекта и ничего больше.

Пример. Добавить два конструктора в класс Rectangle.

⇓

```
class Rectangle {
  int x1, y1, x2, y2;

  public Rectangle() {
      x1 = -1; y1 = -1; x2 = 0; y2 = 0;
  }

  public Rectangle(int x1, int y1, int x2, int y2) {
      this.x1 = x1;
      this.y1 = y1;
      this.x2 = x2;
      this.y2 = y2;
  }
}
```

Если конструктор в классе не определен, компилятор создает пустой конструктор без параметров. Если программист явно определил хотя бы один конструктор, то пустой конструктор создаваться не будет.

Из одного конструктора можно вызвать другой конструктор того же класса при помощи синтаксиса **this(аргументы).** Этот вызов должен быть первым оператором в коде конструктора.

В создании объекта всегда участвует какой-нибудь из его конструкторов. Конструктор вызывается после выполнения явной или неявной инициализации полей.

Замечание. Инструкции инициализации это программный код, расположенный внутри определения класса, но за пределами конструкторов и других методов. В предыдущем примере мы могли бы инициализировать поля прямоугольника вместе с их объявлением.

⇓

```
int x1 = -1, y1 = -1, x2 = 0, y2 = 0;
```

4.3. Поля и методы объекта

Поля объекта хранят все его данные или, говоря иначе, его состояние. Методы объекта - это функции, из которых доступны поля объекта.

Пример. Добавить два метода в класс Rectangle: move() – передвинуть и isInside() – проверка попадания точки в прямоугольник.

⇓

```
class Rectangle {
  int x1, y1, x2, y2;

  // Не будем повторять код конструкторов
  public Rectangle() { …}

  // Передвинуть прямоугольник на dx по оси x и на dy по оси y
  public void move(int dx, int dy) {
      x1 += dx;
      x2 += dx;
      y1 += dy;
      y2 += dy;
  }

  // Если точка (x,y) находится внутри прямоугольника, возвращает
true
  public boolean isInside (int x, int y) {
      return x1 < x && x < x2 && y1 < y && y < y2;
    }
}
```

4.4. Ссылка this

В каждом классе предопределена ссылка this, которая ссылается на объект – экземпляр данного класса. На какой именно экземпляр ссылается this? Именно на тот, чей метод использовал эту ссылку.

Ссылка this необходима, когда надо вызвать метод объекта другого класса и передать ему данный объект в виде параметра.

Можно использовать ссылку this для уточнения имени поля или метода. Так в предыдущем примере в теле конструктора мы писали

this.x1 = x1;

чтобы отличить поле класса от одноименного параметра метода.

Вышеупомянутый доступ из методов объекта к его полям также обеспечивается тем, что при вызове методам неявно передается дополнительный параметр this.

4.5. Статические поля и методы

Поля и методы, принадлежащие классу, а не его экземплярам, объявляются при помощи модификатора **static.** Доступ к ним осуществляется при помощи имени класса.

Существуют классы, которые имеют только статические методы, например, java.lang.System из библиотеки Java.

Пример. Класс Circle (окружность) со статическим полем, содержащим приближенное значение числа "пи".

⇓

```
// Объявление класса
class Circle {
  final static double PI = 3.14159;
  …
}
…
// За пределами объявления константа доступна через имя класса
d = Circle.PI * r;
```

Идиома "final static" означает константу, существующую в памяти в одном экземпляре.

Статическим методам не передается ссылка this, по этой причине им не доступны поля и методы экземпляров. Из статических методов можно обращаться только к статическим полям и вызывать только статические методы. Это не мешает вызывать статические методы и поля из обычных методов.

Инициализация статических полей происходит однажды при загрузке класса. Сложная инициализация делается при помощи статических инициализаторов - блоков **static {операторы}**, расположенных внутри объявления класса.

4.6. Задание

1. Объявить класс "Окружность" с полями: x, y - координаты центра окружности, radius - радиус окружности и методами:
   передвинуть окружность на dx и dy;
   проверить попадание заданной точки внутрь данной окружности;
   проверить попадание другой окружности внутрь данной;
   вывести на экран параметры окружности.
2. Объявить класс "Вектор" для хранения ссылок на объекты. Класс должен иметь следующие поля:
   массив ссылок, который может расти;
   количество ссылок в массиве.
   Класс должен иметь следующие методы:
   очистить весь массив;
   добавить ссылку в массив;
   получить i-й элемент;
   удалить i-й элемент;
   вывести значения массива на экран.
3. Объявить класс "Матрица" с полем - двухмерным массивом вещественных чисел и методами: сложение с другой матрицей, умножение на число, умножение на другую матрицу, транспонирование, вывод на печать.

5. НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ

5.1. Наследование

При наследовании производный класс расширяет и изменяет возможности базового класса.

Пример. Объявить класс Figure - абстрактную фигуру, которая имеет две координаты и которую можно перемещать по плоскости.

⇓

```
class Figure {
  int x, y;

  public Figure(int x, int y) {
```

```
        this.x = x;
        this.y = y;
    }

    public void move (int dx, int dy) {
        x += dx;
        y += dy;
    }

    public void print() {
        System.out.print("Figure: " + x + "  " + y);
    }
}
```

Пример. Объявить класс Rect - прямоугольник, расширяющий класс Figure. Экземпляры Rect имеют ширину и высоту и их параметры можно распечатывать.

⇓

```
class Rect extends Figure{
  int w, h;

  public Rect(int x1, int y1, int x2, int y2) {
      super(x1, y1);
      this.w = x2 - x1;
      this.h = y2 - y1;
  }

  public void print() {
      System.out.print("Rectangle: " + x + "  " + y + "  " + w + "  " + h);
  }
}
```

Пример. Объявить класс Circle - окружность, расширяющий класс Figure. Экземпляры Circle имеют радиус и их тоже можно распечатывать.

⇓

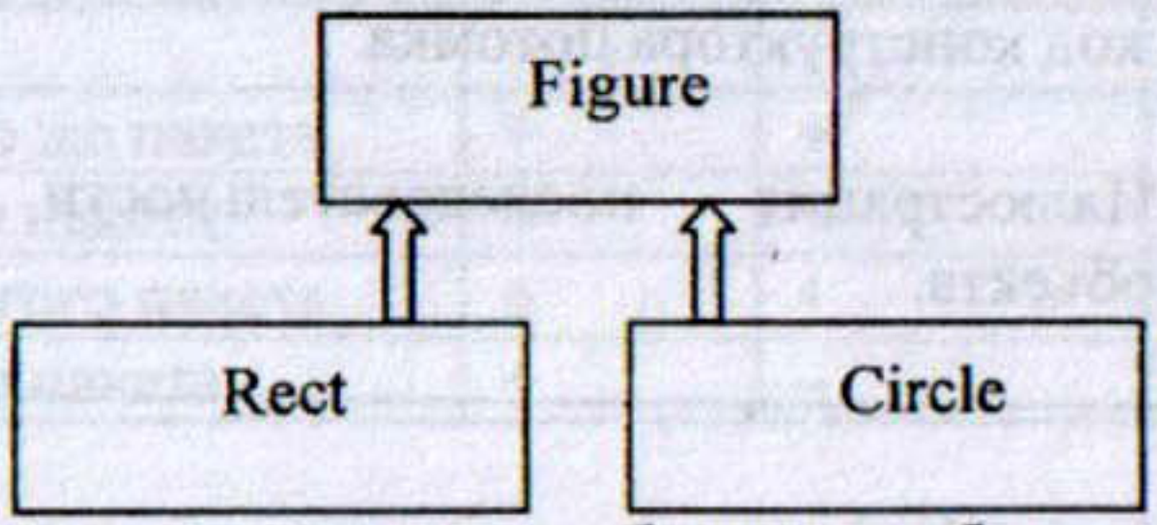
```
class Circle extends Figure{
  int r;

  public Circle(int x, int y, int r) {
      super(x, y);
      this.r = r;
```

```
    }

    public void print() {
        System.out.print("   Circle: " + x + "  " + y + "  " + r);
    }
}
```

Теперь мы имеем структуру классов, которую на диаграмме можно изобразить так.



В программе можно создавать объекты любого из трех классов и вызывать их методы.

⇓

```
...
Figure figure = new Figure(10, 10);
figure.move(100,100);
figure.print();
Rect rect = new Rect(10,10,100,200);
...
```

Если в производном классе D объявить поле f, одноименное с полем базового класса, то объект D будет содержать два поля, причем производное имя будет скрывать базовое.

Обратиться из метода производного класса к полю f базового можно так:

super.f

Если в производном классе объявить метод m(), одноименный с методом базового класса, то:

- при совпадении параметров производный метод перекроет базовый;
- при несовпадении параметров производный метод перегрузит базовый.

Замечание. Перекрытый метод будет скрыт новым методом в производном классе, перегруженный метод будет виден наряду с новым методом.

Перекрытый метод базового класса можно вызвать из метода производного класса при помощи синтаксиса **super.m().**

5.2. Конструкторы и наследование

Из конструктора потомка, прежде всего, вызывается конструктор предка, после этого выполняются инструкции инициализации объекта-потомка, а затем - код конструктора потомка.

Пример. Иллюстрация последовательности действий при конструировании объекта.

⇓

```
class A {
  { System.out.print(1); }
  A(){  System.out.print(2);  };
}

class B extends A {
  { System.out.print(3); }
  B(){ System.out.print(4);  };
}
```

При создании объекта класса B будет напечатано "1234".

Желаемый конструктор предка вызывается при помощи синтаксиса **super(аргументы).** Если это не сделано, компилятор сам вставит вызов **super()** в байт-код, и базовый класс обязан такой конструктор иметь.

Замечание. Если конструкторы класса не доступны, он не сможет наследоваться.

Завершение жизни объекта

В Java объект существует, пока есть ссылки на него. Если ссылок нет, объект будет уничтожен сборщиком мусора, но когда это произойдет, предсказать нельзя.

5.3. Инкапсуляция и модификаторы доступа

Инкапсуляция - это сокрытие деталей реализации внутри класса. Инкапсуляция очень важна для разработки программ, т.к. избавляет программиста от несущественной для него в этот момент информации.

В Java существует несколько степеней инкапсуляции (уровней доступа), и все они перечислены в таблице в порядке возрастания открытости.

Член класса доступен для ...	public	protected	default	private
Того же класса	+	+	+	+
Подкласса из того же пакета	+	+	+	–
Класса из того же пакета	+	+	+	–
Подкласса из другого пакета	+	+	–	–
Класса из другого пакета	+	–	–	–

При наследовании уровень доступа к перекрываемому методу может быть оставлен прежним или расширен, но не сужен! Это вытекает из того постулата Java, что в контексте программы ссылка на предка может быть заменена ссылкой на потомка и программа при этом останется правильной.

5.4. Класс Object

Все классы Java (даже массивы) происходят от класса java.lang.Object. Методы класса Object можно разделить на две категории: методы поддержки многопоточности и прикладные методы. Методы для многопоточности будут рассмотрены позже, а прикладные методы приведены в таблице.

Прикладные методы	Описание
boolean equals(Object o)	Проверяет идентичность объектов (тождественность проверяется операцией ==)
int hachCode()	Возвращает значение хэш-кода. Используется для хранения объекта в некоторых контейнерах
Class getClass()	Возвращает объект - описатель класса
String toString()	Сопоставляет объекту строку символов
Object clone()	Клонирует объект при помощи побитового копирования полей
void finalize()	Деинициализирует объект. Вызывается перед уничтожением объекта сборщиком мусора

5.5. Полиморфизм

Ссылку на объект производного класса можно поместить в переменную, имеющую базовый тип.

⇓

```
Figure x = new Rect(10, 10, 100, 200);
Figure y = new Circle(10, 10, 100);
```

Если теперь мы будем вызывать метод print() для x и y, то распечатается «Rect» и «Circle», а не «Figure», как можно было бы ожидать. Иными словами, хотя x и y ссылаются на базовый тип объекта, вызываться будут методы производных классов, именно тех, к которым относятся реально существующие объекты - для ссылки x это Rect, а для y это Circle. Это и есть полиморфизм.

Пример. Создать массив из прямоугольников и окружностей.

⇓

```
Figure [] figures = { new Rect(10,10,100,200)
                      new Rect(10,10,100,200),
                     new Circle(10,10,100),
};
```

Передвинуть и распечатать все элементы массива.

⇓

```
for (int i = 0; i < figures.length; i++) {
  figures[i].move(40, 40);
  figures[i].print();
}
```

При желании мы можем проверить тип каждого элемента при помощи оператора **instanceof**. Оператор **x instanceof Y** возвращает истину, если объект x относится к классу Y, расширяет класс Y или реализует интерфейс Y.

Пример. Определить количество окружностей в массиве фигур.

⇓

```
int countCircle = 0;
for (int i = 0; i < figures.length; i++) {
  if (figures[i] instanceof Circle)
    countCircle++;
}
```

Замечание. При правильном использовании полиморфизма надобность в операторе instanceof возникает редко.

5.6. Раннее и позднее связывание

Механизм полиморфизма основан на позднем связывании, когда адрес вызываемого метода определяется в момент выполнения программы, а не во время компиляции, как при раннем связывании.

И для метода move(), и для методов print() выполняется именно позднее связывание. Если для метода print() это оправдано полиморфизмом, то для метода move() это лишь вызывает дополнительный расход времени при выполнении программы. Чтобы заставить компилятор выполнять раннее связывание, достаточно объявить метод со спецификатором final; final-методы не предназначены для перекрытия, поэтому полиморфизм для них не нужен.

5.7. Абстрактные методы и классы

Метод Figure.print() в последнем примере носит формальный характер. Сам он непосредственно не работает, но удалить его нельзя, иначе в программе печати компилятор не оттранслирует строку

⇓

```
figures[i].print();
```

Тело этого метода можно сделать любым, даже совсем удалить. Утратив тело, метод Figure.print() перейдет в категорию абстрактных методов и должен объявляться в классе Figure с модификатором abstract.

⇓

```
abstract public void print();
```

Класс, который имеет хотя бы один абстрактный метод, сам становится абстрактным.

⇓

```
abstract class Figure {
...
```

Экземпляры абстрактного класса создавать в программе нельзя, такие классы служат только для наследования.

5.8. Преобразование типа

Преобразование между типами, связанными отношением наследования, может быть *восходящим*, от потомка к предку, или *нисходящим*, от предка к потомку (предки на диаграмме классов обычно располагаются выше потомков).

Восходящее преобразование выполняется неявно.

Нисходящее преобразование выполняется при помощи операции приведения типа:

D d = (D) b.

Если преобразование невозможно, создается исключительная ситуация, поэтому сначала можно проверить допустимость преобразования при помощи оператора проверки типа instanceof.

⇓

```
// B - базовый класс  D - производный класс
B b = new D();
D d;
if (b instanceof D)
    d = (D) b;
```

Полиморфизм делает явное преобразование типов ненужным!

5.9. Задание

Определите систему из 4-х классов: "Точка", "Прямоугольник", "Треугольник", "Изображение".

Точка должна быть абстрактным классом с абстрактным методом вывода на экран, методом для перемещения и абстрактным методом масштабирования - умножения значений координат на масштабный коэффициент.

Прямоугольник и Треугольник должны наследовать Точке.

Изображение должно наследовать Прямоугольнику и служить средством для хранения фигур. Изображение должно допускать операции вывода на экран, перемещения, масштабирования и сложения изображений.

Совет: для хранения элементов изображения используйте массив точек.

6. СТРОКИ

Для работы с текстом во всех языках программирования традиционно используются строки. В языке Java они являются объектами класса String.

После создания объекта String его нельзя изменять (наращивать, укорачивать, заменять отдельные буквы). Когда нужна изменяемая строка, создается объект другого класса - StringBuffer, речь о котором пойдет позже.

Оба класса строк определены в пакете java.lang и не могут наследоваться. Наследованием пожертвовали ради эффективности.

6.1. Конструкторы строк

Как и другие библиотечные типы, представляющие значения, строки имеют много конструкторов.

Конструктор без параметров создает строку, в которой нет ни одного символа - пустую строку.

⇓

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

Конструкторы, которые создают строки из массива символов или его части.

⇓

```
char chars[] = {'a', 'b', 'c', 'd', 'e', 'f'};
String s1 = new String(chars);          // s1 = "abcdef"
String s2 = new String(chars, 2, 3);   // s2 = "cde"
```

Конструктор копирования может создать копию строки-аргумента.

⇓

```
String s3 = new String(s2);
```

Строки можно создавать из массива байтов, так же как и из массива символов.

⇓

```
byte bytes [] = {65, 66, 67, 68, 69, 70};
String s5 = new String(bytes);          // s5 = "ABCDEF"
String s6 = new String(bytes, 2, 3);   // s6 = "CDE"
```

Пример "Ввод строки с клавиатуры".

Без дополнительного программирования мы можем ввести с клавиатуры только массив байтов, это делает метод System.in.read(имя_массива).

⇓

```
// Зарезервировать массив для приема введенных с клавиатуры байтов
byte[] b = new byte[100];
// Ввести байты и вернуть количество введенных байтов
int n = System.in.read(b);
// Создать строку из начальной части массива b
String s = new String(b, 0, n - 1);
```

Если поместить данный код в метод main(), компилятор сообщит об ошибке, связанной с необработанным исключением. Исключения будут рассмотрены позже, а пока, чтобы программа компилировалась, дополните заголовок метода main() фразой "throws java.io.IOException".

⇓

```
public static void main(String[] args) throws java.io.IOException { … }
```

Строковые константы записываются в виде последовательности символов, заключенных в двойные кавычки. Каждая такая константа представляет собой полноценный объект класса String, можно даже вызывать его методы.

⇓

```
String s = "abc";
int n = "abc".length();
```

6.2. Методы класса String

Длина строки

Метод

⇓

```
int length( )
```

возвращает длину строки (количество символов в строке).

Соединение строк

Выполняется при помощи операции "+".

⇓

```
String s1 = "lang" + "uage" ;   // s1 = language
```

Если строка соединяется с не строковым значением простого или ссылочного типа, последнее преобразуется в строку.

⇓

```
String s2 = 3 + " pigs" ;          // s2 = "3 pigs"
String s3 = "four = " + 2 + 2;     // s3 = "four = 22"
String s4 = "four = " + (2 + 2);   // s4 = "four = 4"
```

Преобразование строк

Когда значение должно быть преобразовано в строку, вызывается статический метод **String.valueOf()**. Он перегружен для всех простых типов.

Пример. Преобразование в строку числа "пи".

⇓

```
String s = String.valueOf(3.14159);
```

Для объектов метод String.valueOf() вызывает метод **Object.toString()**, который следует перекрывать в пользовательских классах.

Пример. Переопределение метода toString() для класса Rect.

⇓

```
public String toString() {
    return "Rect: " + width + " x " + height;
}
```

Такое переопределение позволит выводить на экран объекты Rect.

⇓

```
System.out.print( new Rect(10, 10, 100, 100));
```

Преобразование строк в данные простых типов выполняют методы parseXxx() соответствующих оболочечных классов, о которых будет сказано в конце этого раздела.

Извлечение символов

char **charAt**(int index) - возвращает символ строки, стоящий в позиции index.

void **getChars**(int sourceStart, int sourceEnd, char target[], int targetStart) - существующий массив target заполняется частью строки, начиная с позиции sourceStart включительно и заканчивая позицией sourceEnd исключительно. Параметр targetStart задает место в массиве, с которого начнется его заполнение.

char[] **toCharArray**() - создается массив символов и заполняется символами строки.

byte[] **getBytes**() - создается массив байт и заполняется символами строки. Символы предварительно преобразуются в байты, поэтому количество байт массива будет равно количеству символов строки.

byte[] **getBytes**(String encoding) - то же, но с изменением кодировки.

Пример. Создать и заполнить символьный и байтовый массивы символами строки.

⇓

```
String s = String.valueOf(3.14159);
char[] c = s.toCharArray();
byte[] b = s.getBytes();
```

Сравнение строк

boolean **equals**(Object other) - производит посимвольное сравнение строки this со строкой other с учетом регистра символов.

boolean **equalsIgnoreCase**(String other) - то же, но без учета регистра символов.

boolean **regionMatches**(int startIndex, String other, int otherStartIndex, int numChars) - сравнивает между собой два участка строк this и other.

boolean **regionMatches** (boolean ignoreCase, int startIndex, String str2, int str2StartIndex, int nimChars) - то же, но без учета регистра символов.

boolean **startsWith**(String substr) - проверяет, начинается ли строка this с подстроки substr.

boolean **endsWith**(String substr) - проверяет, заканчивается ли строка this подстрокой substr.

int **compareTo**(String other) - позволяет узнать, какая строка больше и возвращает отрицательное число, если строка this меньше, чем other, ноль, если строки совпадают, и положительное число, если строка this больше, чем строка other.

Пример. Даны две строки. Найти самую большую непрерывную общую подстроку.

⇓

```
String si = "aaabbbcccddd", sj = "sssbcccdfff";
int imax = 0, nmax = 0;
for (int i = 0; i < si.length() - 1; i++)
    for (int n = 1; i + n < si.length(); n++)
        for (int j = 0; j + n < sj.length(); j++)
            if (si.regionMatches(i, sj, j, n) && n > nmax) {
                imax = i; nmax = n;
            }
```

После выполнения этого кода переменная imax содержит начало вхождения искомой подстроки в строку si, а переменная nmax - длину искомой подстроки.

Поиск строк

Определить, в каком месте строки в нее входит символ или другая строка, позволяет серия методов indexOf():

int **indexOf**(int ch) - возвращает место первого вхождения в строку this символа ch;

int **lastIndexOf**(int ch) - возвращает место последнего вхождения в строку this символа ch;

int **indexOf**(String substr) - возвращает место первого вхождения в строку this подстроки substr;

int **lastIndexOf**(String substr) - возвращает место последнего вхождения в строку this подстроки substr;

Есть еще четыре таких же метода, но с дополнительным параметром int startIndex - начальная позиция поиска.

Модификация строк

Как вам уже известно, строку изменить нельзя, но можно создать другую строку, равную части исходной строки.

String **substring**(int beginIndex) - создает новую строку, равную части строки this, начинающейся с позиции beginIndex.

String **substring**(int beginIndex, int endIndex) - создает новую строку, равную участку строки this, начиная с позиции beginIndex включительно и заканчивая позицией endIndex исключительно.

Пример. Дано предложение "Алгоритмы+Данные=Программы" или другое подобной же структуры ("Саша+Маша=Любовь"). Составить предложение "Алгоритмы=Программы-Данные".

⇓

```
String source = "Алгоритмы+Данные=Программы";
// Определяем позицию символа '+'
int p1 = source.indexOf('+');
// Определяем позицию символа '='
int p2 = source.indexOf('=');
// Вырезаем первое слово
String alg = source.substring(0, p1);
// Вырезаем второе слово
String dat = source.substring(p1 + 1, p2);
// Вырезаем третье слово
String prg = source.substring(p2 + 1);
// Сшиваем все по-новому
String target = alg + "=" + prg + "-" + dat;
```

Прочие методы

String **concat**(String s) - присоединяет строку s к строке this.

String **trim**() - удаляет ведущие и завершающие пробельные символы из строки this (пробельными символами считаются: ' ', \r, \n, \t).

String **replace**(char original. char replacement) - заменяет все вхождения символа original символом replacement.

String **toLowerCase**() - изменяет регистр символов в строке, делая все буквы строчными.

String **toUpperCase**() - изменяет регистр символов в строке, делая все буквы прописными.

Напоминаем, что эти методы, равно как и другие, не изменяют строку this, а создают новый объект-строку и возвращают ссылку на него.

6.3. Изменяемые строки

Чтобы получить изменяемую строку, надо создать объект класса java.lang.StringBuffer.

У этого класса нет многого, из того, что имеет класс String, зато есть несколько версий методов:

append() – добавления;
delete() – удаления;
insert() - вставки подстроки в строку this.

Кроме того, имеются методы **setCharAt()** и **deleteCharAt()** для замещения и удаления отдельных символов.

Пример. Удалить из строки s1 все вхождения подстроки s2.

⇓

```
String s1 = "qwe12qwe345qwe678qwe90qwe";
String s2 = "qwe";
StringBuffer s = new StringBuffer(s1);
int p = -1;
while ( (p = s.toString().indexOf(s2)) >= 0 ) {
    s.delete(p, p + s2.length());
}
s1 = s.toString();
```

6.4. Классы-оболочки для простых типов

В том же пакете java.lang определены классы-оболочки для простых типов:

Boolean, Byte, Character, Double, Float, Integer, Long, Short и даже Void.

Все они похожи друг на друга. Рассмотрим эти классы на примере класса Integer.

Конструкторы

Integer(int) позволяет создать объект вокруг значения int. Это значение будет сохраняться внутри объекта неизменным.

Другой конструктор, **Integer**(String) создает объект из строки, если она является записью целого числа.

Методы объекта

byte **byteValue**(), double **doubleValue**() и т.п. служат для преобразования хранимого значения в иные простые типы данных.

Методы класса

int **parseInt**(String s) и

int **parseInt**(String s, int radix) разбирают строку и возвращают записанное в ней число. Второй метод делает это для любых систем счисления от 2 до 16.

Integer **valueOf**() делает то же, но получают объект Integer, а не значение простого типа int.

Символические статические константы MIN_INTEGER и MAX_INTEGER содержат наименьшее и наибольшее из целых чисел.

Замечание.

В классах Double и Float объявлены символические статические константы POSITIVE_INFINYTI, NEGATIVE_INFINYTI и NaN.

Пример. Выполнить преобразование целого числа 123 в строку.

⇓

```
String s1 = String.valueOf(123);          // 1-й способ - рекомендуемый
String s2 = new Integer(123).toString();  // 2-й способ
String s3 = "" + 123;                     // 3-й способ
```

Хотя третий способ выглядит короче, по эффективности он сильно проигрывает двум первым.

Пример. Выполнить обратное преобразование: строку - в целое число.

⇓

```
int i1 = Integer.parseInt("123");          // 1-й способ - рекомендуемый
int i2 = (new Integer("123")).intValue();  // 2-й способ
```

6.5. Задание

1. Ввести предложение с клавиатуры. Заменить все вхождения одного слова в предложение другим словом.

Совет: Для ввода строки с клавиатуры воспользуйтесь примером, приведенным в начале раздела.

2. В строке находится одно слово. Изменить порядок букв в слове на обратный.
3. Даны два слова, каждое в отдельной строке. Распечатать все буквы, которые присутствуют в одном слове и отсутствуют во втором.

4. Дан массив строк. Упорядочить строки в массиве по алфавиту.
5. Ввести предложение с клавиатуры, разобрать его на отдельные слова и напечатать их в столбик. Словом считать последовательность букв, ограниченную пробелами, знаками препинания, началом или концом строки. Для решения задачи создать специальный класс с конструктором, получающим строку-предложение, и соответствующими методами.

7. ОБРАБОТКА ИСКЛЮЧЕНИЙ

7.1. Оператор throw

Механизм исключений Java предназначен для обработки нештатных ситуаций, возникающих во время выполнения программы. Например, программа собирается читать файл, а его на диске нет. Или пользователь должен ввести с клавиатуры число, ошибается и вводит нечисловые символы.

Если в программе возникает такая ситуация, программист должен возбудить исключение оператором **throw.**

Пример. Перед делением целых чисел проверить, не равен ли нулю делитель и, если это так, "выбросить" объект исключения.

⇓

```
if (y == 0)
    throw new Exception();
z = x / y;
```

Чтобы объект мог выполнить роль исключения, его тип должен расширять класс java.lang.Throwable.

Стандартная реакция исполняющей системы на выброшенное исключение - выдать сообщение о нем и снять программу. Чтобы перехватить исключение в программе, надо воспользоваться охраняемым участком.

7.2. Охраняемый участок

Охраняемый участок представляет собой блок кода, которому предшествует слово try. В этом блоке возможно появление исключений.

За охраняемым участком всегда следует блок обработки исключений. Блок обработки похож на функцию с именем catch и одним параметром.

При возникновении исключения выполнение блока try немедленно прерывается, и управление передается блоку catch.

⇓

```
try {
    // охраняемый участок
}
catch (Exception e) {
    // блок обработки исключения e
}
```

За блоком try могут следовать несколько блоков catch. Для обработки исключения выбирается первый из подходящих блоков.

Подходящим считается блок, в котором тип параметра совпадает или является подклассом для класса выброшенного исключения.

Чтобы в одном блоке catch перехватить все исключения, нужно задать тип параметра Exception.

7.3. Типы исключений

Корневая часть дерева исключений выглядит так:

```
java.lang.Throwable
    |
    +--java.lang.Error
    |
    +--java.lang.Exception
        |
        +--java.lang.RuntimeException
```

Все исключения, которые выбрасывает виртуальная машина или выполняемая программа при помощи оператора throw, происходят от класса Throwable. Только потомки этого класса могут быть пойманы блоком catch.

Класс Throwable имеет 2 конструктора: без параметров и с одним строковым параметром. При создании объект этого класса получает "фотографию" стека вызовов поцедур и, возможно, строку сообщения.

Некоторые из методов класса Throwable приведены в таблице.

Метод	Описание
Throwable fillInStackTrace()	Заносит в объект фотографию стека. Полезен при повторном возбуждении

	исключения
String getMessage()	Извлекает сообщение из исключения
printStackTrace()	Выводит фотографию стека в стандартный поток

Класс Trowable имеет подклассы, которые не добавляют никаких методов:

- Error - исключения, которые не следует ловить в программе - ошибки виртуальной машины.
- Exception - исключения, которые ловятся. Это предок всех программных исключений.
- RunTimeException – исключения, свидетельствующие о таких ошибках, после которых продолжать работу программы, как правило, не целесообразно.

Пример "Ввод строки с клавиатуры - 2".

Метод System.in.read(b), который вводит данные с клавиатуры, способен выбросить исключение типа java.io.IOException. Обработаем это исключение, выдав в выходной поток сообщение и содержимое стека вызовов процедур.

⇓

```
byte[] b = new byte[100];
String s = null;
try {
    int n = System.in.read(b);
    s = new String(b, 0, n - 1);
} catch (java.io.IOException e) {
    // Выводим сообщение
    System.out.println("Ошибка: " + e.getMessage());
    // Выводим содержимое стека
    e.printStackTrace();
}
```

В пакетах java.lang, java.util, java.io, java.net и других описано большое количество подклассов исключения Exception. В исключении более важен его тип, а не содержащаяся в нем строка, поскольку тип исключений позволяет распределять их обработку по различным блокам catch.

Пример. Ввести два числа из командной строки, сложить и результат напечатать.

⇓

```
public static void main(String[] args) {
    int i0 = 0, i1 = 0;
    try {
        // При некорректной записи числа метод parseInt()
        // выбросит исключение NumberFormatException
        i0 = Integer.parseInt(args[0]);
        i1 = Integer.parseInt(args[1]);
    } catch (NumberFormatException ex) {
        System.err.println("Work cannot be done." + ex.);
        System.exit(1);
    }
    System.out.println(i0 + i1);
}
```

7.4. Блок finally

Часто возникает ситуация, когда необходимо что-то сделать независимо от того, было или не было возбуждено исключение, например, закрыть ранее открытый файл. Это действие выполняют внутри блока finally, расположенного после всех блоков catch.

Есть два основных способа использования блока finally, которые мы покажем схематически. В схемах выражение "пролог();" означает действия, выполняемые до того места в программе, где возможно исключение, а выражение "эпилог();" означает действия, выполняемые ниже места возможного исключения.

```
// Схема 1 - эпилог надо выполнять в любом случае

пролог();
try {
  // другие действия
} finally {
  эпилог();
}
```

```
// Схема 2 - эпилог надо выполнять, только если пролог успешно выполнен

Object val = null;
try {
  val = пролог();
  // другие действия
} finally {
```

```
  if (val != null)
    эпилог();
}
```

Перед входом в блок finally причина завершения блока try (естественное завершение, инструкция throw, brake или return) запоминается и после выхода из блока try восстанавливается. Восстановления не происходит, если блок finally сам завершается инструкцией throw, brake или return.

7.5. Спецификация исключений в методе

Исключения, возбуждаемые внутри метода, могут быть обработаны в этом же методе (путем заключения в блок try с последующим блоком catch) либо выпущены наружу. В этом случае им нужно "выписать пропуск" – перечислить в предложении throws в заголовке метода. Сам метод становится потенциальным источником исключений, и его вызов должен быть заключен в блок try, либо вызывающий метод обязан упомянуть этот тип исключения в своем предложении throws и т.д.

Поскольку за соблюдением этих правил следит компилятор, программист на Java уверен, что все исключительные ситуации в его программе будут обработаны.

Пример спецификации двух исключений в заголовке метода.

⇓

```
void myMethod() throws SomeException, ZeroException { …
```

Появление некоторого класса в списке throws подразумевает и все его подклассы.

Если метод переопределяется в потомке, он может выбрасывать те же исключения, что и метод предка или их подмножество. Метод потомка не может выбрасывать такие исключения, которые не способна поймать программа, вызвавшая метод предка, т.к. тогда нарушится принцип Java, что потомок может заменить предка в любом контексте.

Это не относится к конструкторам, т.к. конструкторы не наследуются.

Пример. Унаследовать класс и перекрыть метод, который может выбрасывать исключения.

⇓

```
class Exc1 extends Exception {}
class Exc2 extends Exception {}
```

```
class One {
  void f() throws Exc1, Exc2 {};
}

class Two extends One {
  void f() throws Exc2{};
}

public static void main(String[] args) throws Exc1, Exc2
{
  One one = new Two();
  one.f();
}
```

7.6. Исключения времени выполнения

В классе Exception есть подкласс RunTimeException, к которому относятся такие исключения как:

ArithmeticException - арифметическая ошибка;
ClassCastException - ошибка преобразования типов;
NullPointerException - использование нулевой ссылки;
IndexOutOfBoundsException - выход за границы массива;
ConcurrentModificationException - изменение данных другим потоком и ряд других.

С одной стороны источником этих исключений может быть любое выражение в программе, с другой стороны самый разумный способ их обработки, как правило, в том, чтобы выдать информацию об исключении и завершить программу (последнее как раз и сделает виртуальная машина, если мы сами откажемся от обработки исключения). Поэтому на исключения времени выполнения не распространяется правило спецификации в методе всех исключений, которые он способен выбросить но не обрабатывает.

7.7. Повторное возбуждение исключений

После обработки исключение можно возбудить повторно для продолжения обработки во внешнем блоке try.

Пример. Повторно возбудить исключение, обновив в нем информацию о стеке вызовов.

⇓

```
try {
  a = b / c;
```

```
} catch(ArithmeticThrowable e) {
  a = Number.MAX_VALUE;
  throw e.fillInStackTrace();
}
```

Если нужно возбудить исключение, не внося в него никаких изменений, применяют оператор throw без параметров.

В блоке обработки можно выбросить и новое исключение.

7.8. Задание

1. Создать статический метод, который преобразует строку в вещественное число. Успешному преобразованию не должны мешать никакие символы, находящиеся в строке до и после записи числа. Если в строке можно усмотреть несколько вещественных чисел, для преобразования нужно выбрать первое. Если преобразование невозможно, должно возбуждаться исключение типа AlasException. Тип AlasException определить в программе, унаследовав Exception.
2. Создать класс "Очередь", основанный на циклическом массиве объектов. Возбуждать исключения при неудаче операций добавления и извлечения из очереди. Типы возбуждаемых исключений определить самому.
3. Оснастить обработкой исключений решения задач из раздела "Строки".

8. ИНТЕРФЕЙСЫ И ВНУТРЕННИЕ КЛАССЫ

8.1. Интерфейсы

Интерфейс - это набор методов, объединенных общим предназначением.

Интерфейсы нужны, чтобы объявить о возможностях класса и выделить эти возможности из всего того, что умеет класс.

Пример. Определить интерфейс "Инверсия", состоящий из методов "инвертировать" и "сдвинуть_циклически". Методы изменяют состояние какого-то последовательного хранилища (контейнера) объектов.

⇓

```
public interface Inversion {
    void invert ();    // порядок элементов меняется на обратный
    void cshift ();    // сдвигается вправо циклически на 1 элемент
}
```

- Для определения интерфейса используется ключевое слово языка **interface**.
- Все методы интерфейса автоматически являются public.
- Интерфейс, объявленный в отдельном файле, должен быть объявлен как public.

8.2. Реализация интерфейса

Интерфейс следует рассматривать как некое соглашение, которое должен выполнять класс. Если класс выполняет такое соглашение, говорят, что он реализует интерфейс.

Пример. Реализовать интерфейс Inversion в классе "MyString".

⇓

```
public class MyString implements Inversion {
    // Хранилищем элементов является строка
    private String innerValue;

    // Порядок элементов меняется на обратный
    public void invert() {
        int l = innerValue.length();
        StringBuffer sb = new StringBuffer(innerValue);
        for (int i = 0; i < l; i++) {
            char c = innerValue.charAt(i);
            sb.setCharAt(l - i - 1, c);
        }
        innerValue = new String(sb);
    }

    // Все сдвигается вправо циклически на 1 элемент
    public void cshift () {
        innerValue = innerValue.substring(1) + innerValue.substring(0,1);
    }
}
```

- Ключевое слово **implements** показывает, что класс реализует интерфейс.
- Все методы интерфейса обязательно должны быть реализованы.
- Все методы интерфейса должны быть реализованы с уровнем доступа не ниже, чем в определении интерфейса, т.е. как публичные.
- Один класс может реализовать много интерфейсов.

- Бывают интерфейсы, которые не имеют ни одного метода. Они используются как метки классов, наличие такой метки проверяется оператором instanceof.

8.3. Поля интерфейса

Кроме методов, интерфейс может иметь статические поля.

Пример. Определить интерфейс Month, который по номеру месяца определяет число дней в нем. Для нумерации месяцев воспользоваться группой статических констант.

⇓

```
interface Month {
  int JANUARY = 1;
  int FEBRUARY = 2;
  int MARCH = 3;
  int APRIL = 4;
  int MAY = 5; // и т.д.

  String month (int no);
}
```

- Поля интерфейса автоматически считаются static и final, эти слова можно не писать.
- Поля не являются частью интерфейса (т.е. не нужно заботиться об их реализации), но они хранятся в статической области памяти, отведенной компилятором для данного интерфейса.

8.4. Наследование интерфейса

Наследование интерфейса позволяет сократить объявления больших интерфейсов. Класс, который реализует производный интерфейс, автоматически реализует и базовый.

Пример. Унаследовать интерфейс Composition от интерфейса Inversion.

⇓

```
interface Composition extends Inversion {
  void compose();
}
```

Как в классах, на наследование указывает ключевое слово extends.

В отличие от классов для интерфейсов допустимо множественное наследование

⇓

```
interface Iii extends Aaa, Bbb, Ccc{…
```

8.5. Внутренние классы

Классы, определенные внутри других классов, позволяют лучше структурировать программу.

Внутренние классы могут объявляться как внутри объявления внешнего класса, так и внутри его методов.

Все члены внешнего класса доступны из внутреннего класса. Это достигается за счет неявной ссылки на объект внешнего класса, которая всегда имеется в объекте внутреннего класса. Из-за этого объект внутреннего класса нельзя создать, если нет объекта внешнего класса.

Пример. Объявить класс InnerClass внутри класса OuterClass.

⇓

```
class OuterClass {

  public class InnerClass {
      void fun() {
          System.out.println("This is InnerClass.fun");
      }
  }
  …
}
```

Компилятор создает отдельный файл .class для каждого класса, внешнего или внутреннего. Имя файла для внутреннего класса выглядит так: OuterClass$InnerClass.class.

8.6. Создание объектов внутренних классов

Пример. Создать объект внутреннего класса в методе внешнего класса и там же его использовать.

⇓

```
class OuterClass {

  public class InnerClass {
      void fun() {
```

```
            System.out.println("Вызван метод InnerClass.fun()");
        }
    }

    public void fun() {
        System.out.println("Вызван метод OuterClass.fun()");
        InnerClass x = new InnerClass();
        x.fun();
    }
}
```

Очень часто объект внутреннего класса создается в методе внешнего класса и возвращается этим методом для дальнейшего использования.

⇓

```
class OuterClass {
  public class InnerClass {
      ...
  }

  InnerClass classFactory() {
      return new InnerClass();
  }
}
```

Такой проектный шаблон называют фабрикой классов.

Создадим объект внутреннего класса за пределами внешнего, воспользовавшись для этого фабрикой классов.

⇓

```
// Сначала надо создать объект внешнего класса.
OuterClass outer = new OuterClass();
// и только потом – экземпляр внутреннего
OuterClass.InnerClass inner = outer.classFactory();
```

- Объявлять ссылку на объект внутреннего класса нужно при помощи составного **имени OuterClass.InnerClass**.

Можно создать объект внутреннего класса за пределами внешнего, и без фабрики классов, при помощи расширенного синтаксиса оператора new.

⇓

```
OuterClass outer = new OuterClass();
OuterClass.InnerClass inner = outer.new InnerClass();
```

Если необходимо из внутреннего класса сослаться на объект внешнего класса, используется синтаксис **OuterClass.this.**

8.7. Реализация интерфейсов внутренними классами

Реализация интерфейса требует написания программного кода. Если уже существует класс, методы которого полностью или частично реализуют требуемый интерфейс, мы можем использовать готовый код, унаследовав этот класс. Проблема в том, что часто необходимо реализовывать несколько интерфейсов в одном классе, а наследовать можно лишь один класс. Решение проблемы – в реализации интерфейсов внутренними классами.

Пример. Определить интерфейс, закрытый внутренний класс, который его реализует, и метод-фабрику для производства объектов внутреннего класса.

⇓

```
// Интерфейс
interface MyInterface {
  void fun();
}

// Внешний класс
class OuterClass {

  // Закрытый внутренний класс
  private class InnerClass implements MyInterface {
    public void fun() {
      System.out.println("This is InnerClass.fun");
    }
  }

  // Фабрика интерфейсов
  MyInterface classFactory() {
    return (MyInterface) new InnerClass();
  }

}
```

Воспользоваться реализацией интерфейса можно так:

⇓

```
OuterClass outer = new OuterClass();
MyInterface inner = outer.classFactory();
inner.fun();
```

Внутренних классов может быть столько, сколько интерфейсов нужно реализовать. Внутренние классы могут наследовать уже готовую реализацию и, таким образом, служить эквивалентом множественного наследования в C++. Вместо нескольких предков есть несколько внутренних классов, наследующих код.

Пример реализации интерфейса

Предположим, имеется некоторое хранилище (контейнер) объектов. Итератором называется вспомогательный объект, позволяющий последовательно перебрать все объекты из хранилища.

Задача. Во внешнем классе имеется контейнер - массив, который содержит ссылки на объекты вперемежку с пустыми ссылками. Реализовать интерфейс Iterator с методами:

boolean hasNext() – проверяет, есть ли еще объекты в массиве;
Object next() – возвращает очередной объект из массива;
Сделать это при помощи закрытого внутреннего класса.

Объявляем интерфейс Iterator.

⇓

```
interface Iterator {
  boolean asNext();
  Object next();
}
```

Определим внешний класс с закрытым массивом, конструктором и методом для получения интерфейса Iterator

⇓

```
class OuterClass {

  private class InnerClass implements Iterator {
      // здесь будут поля и методы внутреннего класса
  }

  private Object arr[];
```

```
    public OuterClass(Object arr[]) {
        this.arr = arr;
    }

    public Iterator iterator () {
        return new InnerClass();
    }

}
```

Определим закрытый внутренний класс, реализующий интерфейс Iterator.

⇓

```
    private class InnerClass implements Iterator {
      private int curr = 0;

      public boolean hasNext() {
          for (int i = curr; i < arr.length; i++) {
              if (arr[i] != null)
                  return true;
          }
          return false;
      }

      public Object next() {
          while (curr < arr.length && arr[curr] == null)
              curr++;
          if (curr < arr.length) {
              return arr[curr++];
          }
          return null;
      }
    }
```

Создадим объект внешнего класса и заполним его несколькими значениями.

⇓

```
Object [] arr = {null, "abc", null, new Integer(123), null};
OuterClass outer = new OuterClass( arr );
```

Создадим итератор и используем его для распечатки значений массива.

⇓

```
for (Iterator i = outer.iterator(); i.hasNext();)
  System.out.println(i.next());
```

8.8. Задание

1. Создать класс "ЦелыйМассив", который хранит свои данные в массиве.
   а) Реализовать в нем интерфейс "Инверсия";
   б) Объявить интерфейс "Усреднение", состоящий из функций "среднееАрифметическое" и "медиана". Реализовать его в классе "ЦелыйМассив" наряду с интерфейсом "Инверсия".
2. Реализовать интерфейс "Усреднение" в виде внутреннего класса во внешнем класса "ЦелыйМассив". Внешний класс по-прежнему реализует интерфейс "Инверсия". Для создания объектов внутреннего класса определить отдельный метод.

9. ВВОД И ВЫВОД

Входная и выходная информация программы рассматривается в Java как упорядоченная последовательность (поток) байтов или символов.

Все средства ввода и вывода заключены в пакете java.io. В основе этих средств лежат 4 абстрактных класса:

InputStream, OutputStream – для байтовых потоков. Их называют потоками ввода и вывода;

Reader и Writer – для символьных потоков. Их называют потоками чтения и записи. Символьные потоки появились позже, и для работы с символами следует использовать именно их.

Две пары абстрактных классов образуют две отдельные иерархии.

9.1. Классы байтовых потоков

Класс InputStream

Абстрактный класс InputStream предоставляет минимальный набор методов для работы с входным потоком байтов.

Метод	Описание
int available()	Возвращает количество еще доступных байт потока

int read()	Вводит очередной байт. Возвращает значения от 0 до 255, если чтение невозможно, возвращает -1
int read(byte[] buf, int offset, int count)	Вводит байты в массив. Возвращает количество реально введенных байтов
long skip(long n)	Пропускает n байтов потока
void mark()	Ставит метку в текущей позиции потока
void reset()	Устанавливает текущую позицию на последнюю поставленную метку
void close()	Закрывает поток и высвобождает занятые системные ресурсы

Абстрактный класс InputStream служит предком всех байтовых входных потоков. При наследовании InputStream требуется, чтобы производный класс реализовал абстрактный метод read(), но часто перекрываются и другие методы.

Перечислим некоторых потомков класса InputStream.

- ObjectInputStream - поток объектов. Создается при сохранении объектов системными средствами (см. раздел "Сериализация объектов");
- SequenceInputStream - последовательное соединение нескольких входных потоков;
- ByteArrayInputStream - использует массив байтов как источник данных;
- PipedInputStream - совместно с PipedOutputStream обеспечивает обмен данными между двумя потоками выполнения (см. раздел "Многопоточное программирование");
- FileInputStream - обеспечивает чтение из файла;
- StringBufferInputStream - использует изменяемую строку StringBuffer как источник данных;
- FilterInputStream - абстрактный класс надстройки над классом InputStream.

Надстройка использует другой поток InputStream в качестве источника данных и перекрывает все методы класса InputStream, простыми вызовами соответствующих методов потока-источника. Любой потомок надстройки может перекрыть тривиальные методы надстройки и расширить набор ее методов.

В отличие от наследования надстраивание не ведет к появлению большого числа библиотечных классов. Так если мы имеем классы A1,

A2, …, An и хотим комбинировать их свойства путем наследования, мы вынуждены создать порядка n * n новых классов. Если делать то же путем надстраивания, понадобится всего n новых классов.

В пакете java.io имеется несколько потомков FilterInputStream, каждый из которых добавляет определенную функциональность к любому из входных потоков, например:

- PushBackInputStream – добавляет способность возвращать обратно в поток прочитанные байты;
- BufferedInputStream – добавляет буферизацию данных, что ускоряет процесс ввода;
- DataInputStream – добавляет способность читать значения простых типов. Имеет такие методы как readBoolean(), readInt(), readLong(), readFloat().

Класс OutputStream

Абстрактный класс OutputStream предоставляет минимальный набор методов для работы с выходным потоком байтов.

Метод	Описание
void write(int b)	Абстрактный метод записи в поток одного байта
void write(byte[] buf, int offset, int count)	Запись в поток массива байтов или его части
void flush()	Форсированная выгрузка буфера для буферизированных потоков. Если получателем служит другой поток, его буфер тоже сбрасывается
void close()	Закрытие потока и высвобождение системных ресурсов

Целый аргумент в методе write(int) использован потому, что часто приходится выводить результат арифметической операции над байтами, а он имеет целый тип.

Абстрактный класс OutputStream служит предком всех байтовых входных потоков. Некоторые потомки класса OutputStream.

- ObjectOutputStream - поток двоичных представлений объектов. Создается при сериализации.
- ByteArrayOutputStream - использует массив байтов как приемник данных.

- PipedOutputStream - вместе с PipedInputStream составляет пару потоков для обмена данными между потоками выполнения (threads).
- FileOutputStream - поток для записи в файл.
- FilterOutputStream - абстрактный класс надстройки.

Надстроками над потоком OutputStream служат следующие классы.

- PrintOutputStream – добавляет возможность преобразования простых типов данных в последовательность байтов. Делает это при помощи перегруженного метода print(), который преобразует и помещает их в выходной поток. Метод print() никогда не возбуждает исключение IOException, а записывает ошибки во внутренние переменные, которые проверяет метод checkError().
- BufferedOutputStream – буферизированный выходной поток. Ускоряет вывод.
- DataOutputStream - поток для вывода значений простых типов. Имеет такие методы как writeBoolean(), writeInt(), writeLong(), writeFloat() и т.п.

9.2. Классы Reader и Writer

Еще в JDK 1.1 были введены классы Reader и Writer, которые оперируют с Unicode-символами. Методы этих классов схожи с методами их байтовых аналогов.

Всюду, где возможно, надо пользоваться символьными классами, а где невозможно - байтовыми.

Некоторые потомки класса Writer приведены ниже.

- BufferedWriter - буферизированный выводной поток. Размер буфера можно менять, хотя размер, принятый по умолчанию, пригоден для большинства задач.
- CharArrayWriter - позволяет выводить символы в массив, как в поток.
- StringWriter - позволяет выводить символы в изменяемую строку, как в поток.
- PrintWriter - поток, снабженный операторами print() и println().
- PipedWriter - средство межпоточного общения.
- OutputStreamWriter – мост между классом OutputStream и классом Writer. Символы, записанные в этот поток, превращаются в байты. При этом можно выбирать способ кодирования символов.
- FileWriter - поток для записи символов в файл.
- FilterWriter – служит для быстрого создания пользовательских надстроек

Большинство потомков класса Reader симметричны потомкам класса Writer.

- BufferedReader - буферизированный вводной поток символов.
- CharArrayReader - позволяет читать символы из массива как из потока.
- StringReader - то же из строки.
- PipedReader - парный поток к PipedWriter.
- InputStreamReader – при помощи методов класса Reader читает байты из потока InputStream и превращает их в символы. В процессе превращения использует разные системы кодирования.
- FileReade - поток для чтения символов из файла.
- FilterReader – служит для создания надстроек.

Рассмотрим несколько типичных примеров применения потоков.

Пример. Вводить строки с клавиатуры и записывать их в файл на диске.

⇓

```
try {
  // Создаем буферизованный  символьный входной поток
  BufferedReader in = new BufferedReader(
      new InputStreamReader(System.in));

  // Используем класс PrintWriter для вывода
  PrintWriter out = new PrintWriter (new FileWriter("data.txt"));

  // Записываем строки, пока не введем строку "stop"
  while (true) {
      String s = in.readLine();
      if (s.equals("stop"))
          break;
      out.println(s);
  }
  out.close();

} catch (IOException ex) {
  // Обработать исключение
}
```

Многие конструкторы и методы потоков способны выбрасывать исключение java.io.IOException. В этом примере это делает конструктор FileWriter и метод in.readLine(). В последующих примерах мы будем обработку исключений опускать.

В статических полях класса System in и out представлены потоки стандартного ввода и вывода, которые по умолчанию связаны с клавиатурой и консолью. Поле in имеет тип InputStream, а поле out - тип PrintSream. Если поток out уже пригоден для вывода любых данных, то поток in приходится надстраивать. В предыдущем примере поток System.in надстроен потоком BufferedReader, а для перехода от байтового класса к символьному использован мост InputStreamReader.

Чтобы для вывода в файл использовать оператор println(), создан поток PrintWriter, который посылает данные в поток FileWriter.

Можно выводить непосредственно в поток FileWriter

⇓

```
FileWriter out = new FileWriter("111.txt");
```

но тогда вывод получится более низкоуровневым:

⇓

```
// sep - разделитель строк файла, принятый в данной операционной системе
String sep = System.getProperty("line.separator");
out.write(s + sep);
```

Пример. Читать строки из текстового файла и выводить их на экран.

⇓

```
// Создать поток для чтения строк
BufferedReader in = new BufferedReader( new FileReader("111.txt"));

// Читать и выводить строки на экран для демонстрации
String s;
while((s = in.readLine())!= null)
  System.out.println(s);

// Закрыть поток чтения
in.close();
```

Пример. Читать символы из строки "1234567890", находящейся в памяти.

⇓

```
StringReader in = new StringReader("1234567890");
int c;
```

```
while((c = in.read()) != -1)
  System.out.print((char)c);
```

При невозможности дальнейшего чтения из потока метод read() возвращает -1.

Для контроля все прочитанные символы выводятся на экран. Чтобы вывод имел вид символов, а не целых чисел, выполняется преобразование (char)c.

Пример. Выполнить вывод разнотипных данных в файл, а затем ввод данных из этого файла.

⇓

```
// Открыть поток для вывода данных в файл
DataOutputStream out =
  new DataOutputStream(
      new BufferedOutputStream(
          new FileOutputStream("Data.txt")));

// Вывести число, строку и закрыть файл
out.writeDouble(3.14159);
out.writeBytes("That was pi\n");
out.close();

// Открыть поток для ввода данных из файла
DataInputStream in =
  new DataInputStream(
      new BufferedInputStream(
          new FileInputStream("Data.txt")));

// Открыть поток для ввода строк из файла
BufferedReader inBuf =
  new BufferedReader(
      new InputStreamReader(in));

// Ввести число из потока и показать на экране
System.out.println(in.readDouble());

// Ввести строку из потока и показать на экране
System.out.println(inBuf.readLine());
```

9.3. Класс RandomAccessFile

Для работы с файлами произвольного доступа применяется класс RandomAccessFile с методом seek() для перемещения по файлу.

Он не участвует в рассмотренной иерархии, но реализует интерфейсы DataInput и DataOutput (те же, что реализованы классами InputStream и OutputStream).

Пример. Создать файл прямого доступа, выполнить запись в файл и чтение из файла

⇓

```
// Создать файл с возможностью чтения и записи "rw"
RandomAccessFile rf = new RandomAccessFile("rtest.dat", "rw");

// Записать в файл 10 чисел и закрыть файл
for(int i = 0; i < 10; i++)
  rf.writeDouble(i * 1.414);
rf.close();

// Открыть файл, записать в него еще одно число и снова закрыть
rf = new RandomAccessFile("rtest.dat", "rw");
rf.seek(5 * 8);
rf.writeDouble(47.0001);
rf.close();

// Открыть файл с возможностью только чтения "r"
rf = new RandomAccessFile("rtest.dat", "r");

// Прочитать 10 чисел и показать их на экране
for(int i = 0; i < 10; i++)
  System.out.println("Value " + i + ": " + rf.readDouble());

rf.close();
```

9.4. Класс File

Класс java.io.File совершенно не похож на остальные классы пакета, т.к. работает не с потоками данных, а с элементами файловой системы – каталогами и файлами.

Каждый объект File представляет абстрактный файл или каталог, возможно, и несуществующий.

Абстрактный путь, который заключает в себе объект File, состоит из не обязательного системно-зависимого префикса и последовательности

имен. Префикс выглядит по-разному в различных операционных системах: символ устройства "C:", "D:" в системе Windows, символ корневого каталога "/" в системе UNIX, символы "\\" в UNC и т.д. Каждое имя последовательности является именем каталога, а последнее имя может быть именем каталога или файла. Пустой абстрактный путь не имеет ни префикса, ни имен.

Хотя абстрактный путь не доступен непосредственно, всегда можно получить его строковое представление. Им является строка, которая начинается с префикса и состоит из последовательности имен. Имена разделены разделителем, принятым в данной операционной системе. Разделитель хранится в системном свойстве "file.separator" и в статическом поле separator класса File.

Путь может быть абсолютным или относительным. Абсолютный путь способен однозначно идентифицировать файл или каталог в файловой системе. Относительному пути для этого требуется другой абсолютный путь, который послужит для него "точкой отсчета". Для всех классов пакета java.io такой точкой отсчета является каталог из системного свойства "user.dir".

Объект File создается конструкторами:

- **File(String filePath),** где filePath – имя файла на диске;
- **File(String dirPath, String filePath),** здесь параметры dirPath и filePath вместе задают то же, что один параметр в предыдущем конструкторе;
- **File(File dirObj, String fileName),** вместо имени каталога выступает другой объект File.

Однажды созданный объект File нельзя изменить.

Основные методы класса File сведены в таблицу.

Метод	Описание
String getName()	Возвращает последнее имя пути
String getPath()	Возвращает строковое представление пути
String getParent()	Возвращает путь за вычетом последнего имени
boolean exists()	Проверяет, существует ли файл или каталог в действительности
boolean canWrite()	Проверяет, может ли приложение писать в файл
boolean canRead()	Проверяет, может ли приложение читать из файла
boolean isDirectory()	Возвращает true, если объект представляет каталог
boolean isFile()	Возвращает true, если объект представляет файл
long lastModified()	Возвращает время последней модификации в

	msec
long length()	Возвращает размер файла в байтах
boolean renameTo(File newFile)	Заменяет один абстрактный путь другим
boolean delete()	Удаляет файл или каталог, но не объект. Каталог удаляется, только если он пуст
boolean mkdir()	Создает каталог по заданному абстрактному пути
boolean mkdirs()	Создает каталог и все родительские каталоги, если это необходимо

Каталоги

Каталог - это особый файл, который содержит в себе список других файлов и каталогов.

Если метод isDirectory() возвращает истину, это значить, что объект File представляет каталог и можно вызывать метод

File[] listFiles();

который возвращает массив объектов File для файлов и каталогов, которые находятся в заданном каталоге.

Пример. Получить массив файлов и каталогов, которые находятся в рабочем (или текущем) каталоге.

⇓

```
File path = new File(".");
File[] list = path.listFiles();
for(int i = 0; i < list.length; i++)
  System.out.println(list[i].getName());
```

Строка "." обозначает относительный путь к рабочему каталогу. Абсолютный путь содержится в системном свойстве "user.dir", поэтому первую строку примера можно написать и так

⇓

```
File path = new File(System.getProperty("user.dir"));
```

Фильтры

В пакете java.io определен интерфейс FileFilter, который применяется для

проверки, подпадает ли объект File под некоторое условие. Интерфейс имеет единственный метод

boolean accept(File file);

который должен возвращать истину, если аргумент удовлетворяет условию, и ложь в противном случае.

Одна из модификаций метода listFiles() принимает в качестве аргумента объект FileFilter и возвращает уже профильтрованный массив из объектов.

Пример. Выбрать из текущего каталога лишь те файлы, которые содержат в своем последнем имени буквосочетание, заданное в командной строке.

⇓

```
public static void main(final String[] args) {
  File path = new File(".");

  // Получить массив объектов
  File[] list = path.listFiles(new FileFilter() {
      public boolean accept(File file) {
          String f = file.getName();
          return f.indexOf(args[0]) != -1;
      }
  });

  // Напечатать имена файлов
  for(int i = 0; i < list.length; i++) {
      System.out.println(list[i].getName());
  }
}
```

Интерфейс FileFilter реализован в примере анонимным внутренним классом. Локальные переменные, доступные во внутреннем классе, должны быть объявлены со спецификатором final. Это объясняет присутствие слова final в объявлении аргумента функции main().

9.5. Задание

1. Создайте статический метод для копирования файла с двумя параметрами: имя оригинального файла, имя файла-копии. Перегрузите метод для случая, когда файлы задаются не именами, а двумя объектами java.io.File. Сравните быстродействие метода на буферизированных и не буферизированных потоках.
2. Скопируйте содержимое одного символьного потока в другой поток, попутно делая замену символов. В качестве таблицы замен используйте две строки s1 и s2 одинаковой длины. В строке s1 должны

находиться заменяемые символы, а в соответствующих позициях строки s2 - заменяющие.

3. Создайте класс для ввода целых чисел с клавиатуры. Числа могут разделяться любым количеством пробельных символов (' ', \t, \n, \r).
4. Выведите на экран имена всех файлов, которые находятся в заданном каталоге и всех его подкаталогах.
5. Выполните предыдущее задание, но так, чтобы вывести лишь имена файлов с расширениями java, class и txt.

10. КОНТЕЙНЕРНЫЕ КЛАССЫ

Структуры, предназначенные для хранения однотипных данных, в программировании называют контейнерами. Простейшим контейнером является массив, но, именно в силу своей простоты, массивы могут сделать программу излишне сложной. В пакете java.util программист может найти целую библиотеку контейнеров, более совершенных, чем массив. Общим для всех контейнеров Java является то, что они предназначены для хранения потомков класса Object.

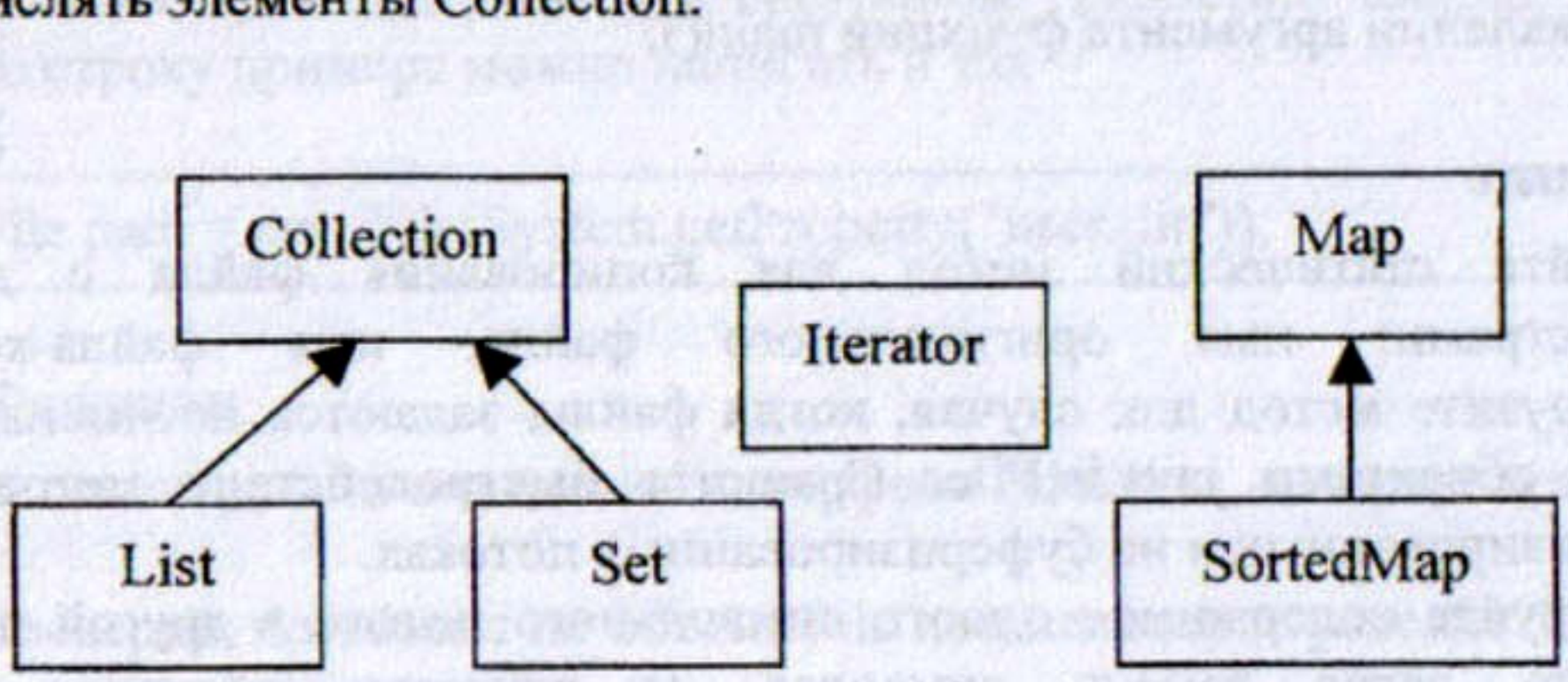
На вершине библиотеки контейнеров Java расположены два основных интерфейса, которые представляют два принципиально разных вида контейнеров:

- интерфейс Collection - множество;
- интерфейс Map - словарь или ассоциативный массив.

Интерфейс Collection расширяется двумя способами:

- интерфейс List - упорядоченное множество;
- интерфейс Set - неупорядоченное множество.

Вспомогательную роль играет интерфейс Iterator, который позволяет перечислять элементы Collection.



10.1. Интерфейс Collection

Основные методы интерфейса Collection перечислены в следующей таблице

Метод	Описание
int size()	Возвращает количество элементов коллекции
boolean contains(Object o)	Возвращает true, если элемент o, есть в коллекции
Iterator iterator()	Воздает итератор для коллекции и возвращает ссылку на него
boolean add(Object o)	Добавляет элемент в коллекцию. Возвращает true, если коллекция изменилась
boolean remove(Object o)	Удаляет одно вхождение элемента o в коллекцию. Возвращает true, если коллекция изменилась
void clear()	Удаляет все элементы из коллекции

Чаще всего для получения контейнерного объекта с интерфейсом Collection используется класс ArrayList.

10.2. Интерфейс Iterator

Для перемещения по коллекции используют итераторы. Интерфейс Iterator содержит минимально необходимые средства для этого.

Метод	Описание
boolean hasNext()	Проверяет, остались ли еще не пройденные элементы коллекции
Object next()	Возвращает очередной элемент. Сразу после создания итератора возвращает первый элемент. При отсутствии элементов выбрасывает исключение NoSuchElementException
void remove()	Удаляет элемент, который последним вернул метод next(). Этим методом следует пользоваться, когда проходишь коллекцию и по пути удаляешь элементы. Метод коллекции remove() в таком случае не пригоден

Пример "Печать коллекции". Создать коллекцию, заполнить ее и распечатать при помощи итератора.

⇓

```
// Создать коллекцию первоначальной емкостью в 10 элементов
Collection col = new ArrayList(10);
```

```
// Заполнить коллекцию объектами Integer
for (int i = 0; i < 10; i++)
  col.add(new Integer(i));

// Получить итератор и с его помощью распечатать коллекцию
Iterator it = col.iterator();
while (it.hasNext())
  System.out.print(it.next() + " ");
```

Вывести на экран все элементы коллекции можно и без итератора

⇓

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

но при этом нельзя будет управлять форматом вывода.

10.3. Интерфейс List

Если контейнер Collection является просто хранилищем, то в контейнере с интерфейсом List каждый элемент имеет порядковый номер - индекс. Номер элемента можно толковать и как позицию, занимаемую элементом в коллекции. Нумерация элементов ведется с нуля.

Здесь представлены лишь те методы интерфейса List, которых нет в интерфейсе Collection.

⇓

Метод	Описание
boolean add(int index, Object o)	Добавляет объект o в позицию index. Возвращает true, если добавление состоялось
Object remove(int index)	Удаляет элемент, стоящий в позиции index из коллекции
Object get(int index)	Возвращает ссылку на элемент, стоящий в позиции index
Object set(int index, Object o)	Заместить элемент в позиции index другим элементом - o
int indexOf(Object o)	Возвращает номер первого вхождения элемента o в коллекцию
int lastIndexOf(Object o)	Возвращает номер последнего вхождения элемента o в коллекцию
boolean isEmpty()	Возвращает true, если коллекция пуста

В пакете java.util имеется 2 класса, реализующих интерфейс List:

- ArrayList – в нем для хранения элементов используется массив;
- LinkedList – для хранения элементов используется двухсвязный список.

Пример "Печать коллекции-2". Создать коллекцию, заполнить ее и распечатать.

Если нам известно, что контейнер реализует интерфейс List, а не просто Collection, мы можем пройти все элементы контейнера и без итератора.

⇓

```
// Создать коллекцию первоначальной емкостью в 10 элементов
List col = new ArrayList(10);

// Заполнить коллекцию объектами Integer
for (int i = 0; i < 10; i++)
  col.add(new Integer(i));

// Распечатать коллекцию
for (int i = 0; i < col.size(); i++)
  System.out.print(col.get(i) + " ");
```

В классе LinkedList имеется ряд методов, которые не входят в интерфейс List:

- addFirst() и addLast() - добавить в начало и в конец списка;
- removeFirst() и removeLast() - удалить первый и последний элементы;
- getFirst() и getLast() - получить первый и последний элементы, которые позволяют использовать объекты LinkedList в качестве стека, очереди и дека (двусторонней очереди).

Специально для работы с List-коллекцией существует интерфейс ListIterator. Он расширяет Iterator методами:

- Object previous();
- boolean hasPprevious();
- int nextIndex();
- int previousIndex() и другими.

10.4. Интерфейс Set

Set (множество) - это интерфейс коллекции, которая не может содержать дубликаты элементов. Иными словами, если e1 и e2 элементы множества, то метод e1.equals(e2) возвращает false.

Хотя Set, добавляет лишь один метод к тем, что составляют интерфейс Collection -

boolean isEmpty();

который возвращает true, если список пуст, он по-иному толкует смысл некоторых других методов, например, add().

Добавление повторяющихся элементов не вызывает исключений, но они не попадают в множество.

В пакете java.util имеется 2 класса, реализующих интерфейс Set:

- HashSet – для хранения элементов использует хэш-таблицу;
- TreeSet – для хранения элементов использует двоичное дерево. Позволяет извлекать упорядоченную последовательность элементов.

Пример. Установить, какие символы входят в состав заданной строки.

⇓

```
// Исходная строка
String source = "qweroiqweurioqweuruqefyqdfygyrwqwtqtqtrqrt";
// Создать, пока пустое множество
Set set = new HashSet();

// Заполнить множество объектами Character
for (int i = 0; i < source.length(); i++) {
  char c = source.charAt(i);
  // Каждый символ, добавляемый во множество, выводим на экран
  if (set.add(new Character(c)))
      System.out.print(c);
}
```

10.5. Упорядочение элементов

Если о любой паре объектов одного типа о1 и о2 можно сказать, что один из них в определенном смысле меньше второго, то коллекцию из таких объектов можно сортировать.

Интерфейс Comparable

В отсортированных коллекциях элементы могут располагаться в их естественном порядке. Это возможно, если элементы имеют этот самый «естественный» порядок, иными словами, если класс элементов реализует интерфейс java.lang.Comparable.

Интерфейс Comparable состоит из единственного метода
public int compareTo(Object other).

Метод возвращает:
отрицательное число, если this < other;
ноль, если this = other;
положительное число, если this > other.

Дополнительным условием является то, что метод compareTo(other) возвращает 0 тогда и только тогда, когда метод equals(other) возвращает true.

Интерфейс Comparator

Если естественного порядка элементов нет или он есть, но нас не устраивает, можно задать в контейнере другой способ упорядочения. Для этого необходимо:

1. объявить класс, реализующий интерфейс java.util.Comparator;
2. создать объект этого класса;
3. передать этот объект в контейнер, например в качестве параметра конструктора.

Интерфейс Comparator включает метод
public int compare(Object o1, Object o2),
который возвращает такие же значения, как метод compareTo(Object other), если предположить, что o1 = this, а o2 = other.

10.6. Интерфейс SortedSet

Представляет собой множество с отношением строгого порядка на его элементах (все элементы должны быть сравнимы между собой). Расширяет интерфейс Set следующими методами.

⇓

Метод	Описание
Comparator comparator()	Возвращает объект Comparator или null, если используется естественная сортировка
Object first()	Возвращает наименьший элемент множества
Object last()	Возвращает наибольший элемент множества
SortedSet headSet(Object to)	Возвращает все элементы, меньшие, чем to
SortedSet tailSet(Object from)	Возвращает все элементы, большие, чем from
SortedSet subSet(Object from, Object to)	Возвращает подмножество элементов от from включительно до to исключительно

Интерфейс SortedSet реализуется классом TreeSet, построенном на базе упорядоченного двоичного дерева.

Интерфейс SortedSet реализуется классом TreeSet, построенном на базе упорядоченного двоичного дерева.

Пример. Создать множество строк, в котором бы они располагались в алфавитном порядке.

⇓

```
// Создаем пустое упорядоченное множество
SortedSet set = new TreeSet();
// Добавляем в него строки в произвольном порядке
set.add("Betha");
set.add("Gamma");
set.add("Alpha");
// Для проверки выведем первый элемент
System.out.println(set);       // это будет "Alpha"
```

Пример. Создать множество строк, в котором они будут располагаться в порядке обратном алфавитному.

Для решения задачи воспользуемся другим конструктором класса TreeSet, который получает компаратор в виде параметра. Остальной код предыдущего примера останется без изменений.

⇓

```
SortedSet set = new TreeSet(new Comparator() {
  public int compare(Object o1, Object o2) {
    return -((String)o1).compareTo(o2);
  }
});
```

10.7. Задание

1. Есть два списка объектов класса Integer. Образовать третий список, который будет содержать: а) объединение элементов двух списков; б) пересечение элементов; в) разность (те элементы, что присутствуют в первом списке и отсутствуют во втором).

2. Создать множество точек (объектов с двумя полями: x и y), в котором точки будут располагаться в порядке их удаленности от начала координат.

3. Имеется отсортированное множество строк и две строки s1 и s2 таких, что s1 < s2. Из заданного множества выбрать: а) полуоткрытый интервал строк [s1, s2); б) закрытый интервал [s1, s2]; в) открытый интервал (s1, s2).
4. Имеется английский текст, в котором слова разделены произвольным количеством пробельных символов. Сколько слов в нем?
5. Создать множество одномерных целых массивов, упорядоченных по количеству элементов в них. Меньшим считается тот массив, в котором элементов меньше.

Часть 2. Продвинутые средства языка

1. АССОЦИАТИВНЫЕ КОНТЕЙНЕРЫ

1.1. Интерфейс Map

Этот интерфейс называют ассоциативным массивом, а иногда словарем. Map осуществляет отображение (mapping) множества ключей на множество значений.

В таблице представлены методы интерфейса Map, которыми он расширяет интерфейс Collection.

Метод	Описание
void clear()	Очищает словарь
boolean containsKey(Object key)	Проверяет, есть ли в словаре ключ key
boolean containsValue(Object value)	Проверяет, есть ли в словаре значение value
Set entrySet()	Создает множество для просмотра пар*
boolean equals(Object o)	Проверяет эквивалентность другому словарю
Object get(Object key)	Определяет значение по заданному ключу
int hashCode()	Вырабатывает хэш-код карты = сумме всех кодов элементов
boolean isEmpty()	Проверяет словарь на пустоту
Set keySet()	Создает множество для просмотра ключей*
Object put(Object key, Object value)	Добавляет пару "ключ-значение"
void putAll(Map t)	Добавляет целый словарь
Object remove(Object key)	Удаляет пару с заданным ключом key
int size()	Возвращает количество элементов словаря
Collection values()	Создает коллекцию для просмотра значений*

* Множества и коллекция связаны со словарем. Изменения в словаре отражаются на них и наоборот.

Интерфейс Map реализован в классе HashMap. Класс HashMap сменил устаревший контейнер Hashtable.

Пример. Создать словарь, заполнить его строками, получить коллекции ключей, значений и пар и распечатать их.

⇓

```
// Создать пустой словарь
Map map = new HashMap();

// Заполнить его чем-нибудь
map.put("1", "111");
map.put("2", "222");
map.put("3", "333");
map.put("4", "333");

// Вывести словарь
System.out.println(map);

// Получить и вывести все ключи
Set keyset = map.keySet();
System.out.println("Set of keys: " + keyset);

// Получить и вывести все значения
Collection valcol = map.values();
System.out.println("Collection of values: " + valcol);

// Получить и вывести все пары
Set entryset = map.entrySet();
System.out.println("Set of entries: " + entryset);
```

На экране увидим:

```
Map: {4=333, 3=333, 2=222, 1=111}
Set of keys: [4, 3, 2, 1]
Collection of values: [333, 333, 222, 111]
Set of entries: [4=333, 3=333, 2=222, 1=111]
```

Внутренний интерфейс Entry

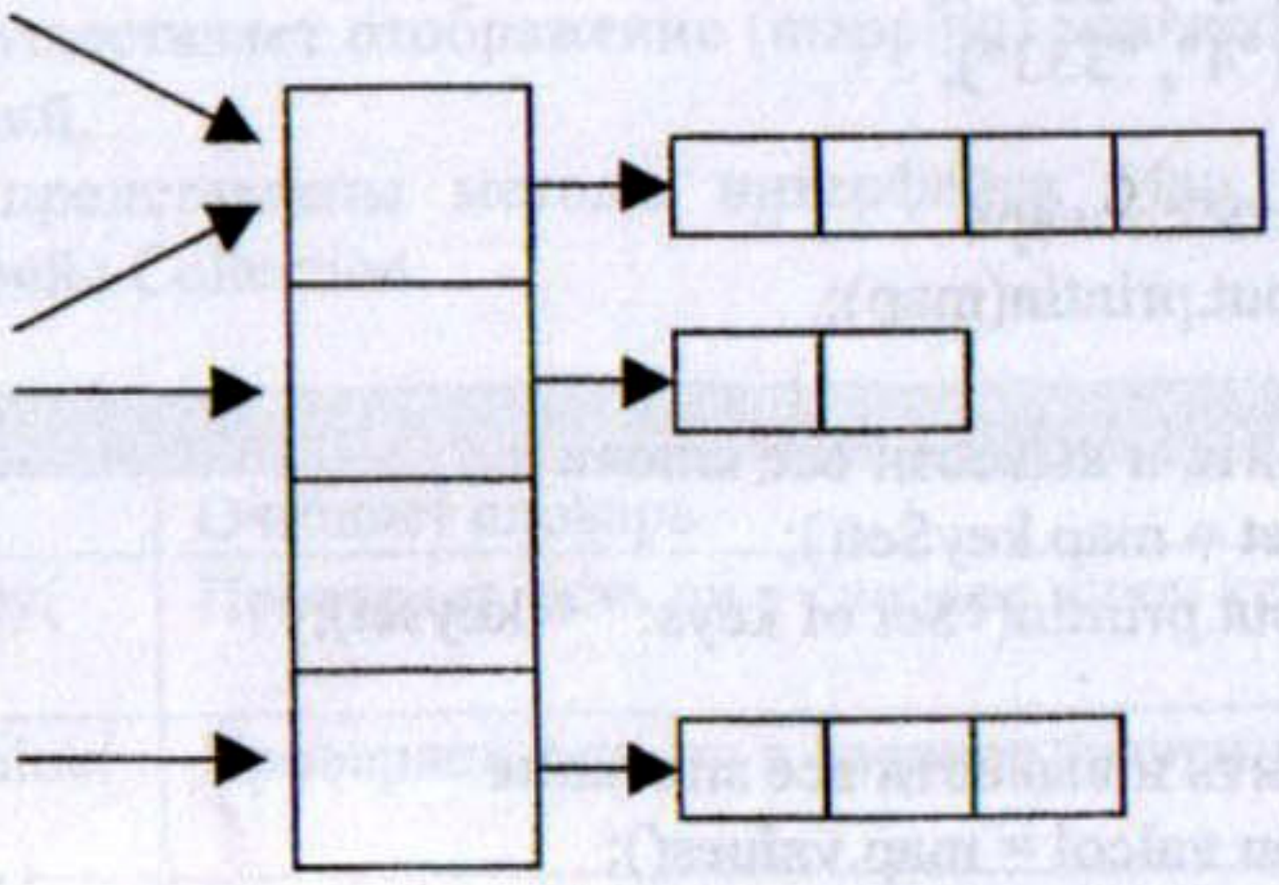
Интерфейс Map.Entry позволяет работать с объектом, который представляет собой пару (ключ, значение). Интерфейс содержит такие методы как:

- boolean equals(Object o) - проверяет эквивалентность двух пар;
- Object getKey() – возвращает ключ пары;
- Object getValue() – возвращает значение пары;

- Object setValue(Object value) – изменяет значение пары.

1.2. Хеширование

Хешированием называется такой способ хранения и доступа к данным, при котором пространство ключей вырожденно отображается на пространство адресов, т.е. разные ключи могут отобразиться в один адрес. По этому адресу находится список всех пар (ключ, значение), чьи ключи отобразились в этот адрес. Поиск нужного ключа в списке ведется последовательно.



Отображение ключа в адрес выполняется при помощи хэш-кода ключа, который возвращает метод объекта-ключа **int hashCode()**. Одинаковые ключи должны возвращать один и тот же хэш-код.

Помимо hashCode(), хеширование использует метод **boolean equals(Object)**, поэтому всякий объект, используемый в роли ключа, должен переопределять эти методы.

Пример. Определить класс ключа для хранения пар в таблице Map.

Предположим, что на ключ по какой-то причине наложено требование, чтобы он был целым числом, кратным 10.

⇓

```
class MyKey {
  final int i;
MyKey(int v) {
  if (v % 10 != 0)
        throw new IllegalArgumentException(String.valueOf(v));
    i = v;
  }

  public int hashCode() {
```

```
        return i / 10;
    }

    public boolean equals(Object o) {
        if ( !(o instanceof MyKey))
            return false;
        return i == ((MyKey)o).i;
    }
};
```

Значения hashCode() и equals() для класса Object вычисляются по адресу объектов и будут давать разные значения для идентичных ключей, поэтому их следует перекрыть.

Чтобы ключи нормально выводились на печать, необходимо перекрыть и метод toString().

⇓

```
public String toString() {
  return String.valueOf(i);
}
```

Воспользуемся созданным классом для заполнения контейнера HashMap.

⇓

```
Map map = new HashMap();
map.put(new MyKey(10), "111");
map.put(new MyKey(20), "222");
map.put(new MyKey(30), "323");
map.put(new MyKey(30), "333");
```

- Два ключа считаются одинаковыми, если и только если они вырабатывают одинаковый хэш-код и сравниваются методом equals().
- При помещении в словарь пар с одинаковыми ключами в словаре сохраняется только последняя пара, т.е. (30, 333).
- Тип значений может быть разным в одном и том же словаре.
- Тип ключей в одном словаре тоже может быть разным, например,

⇓

```
map.put("40", "444");
```

Производительность хеширования

Чтобы хеширование работало хорошо:

1) метод hashCode() должен равномерно отображать ключи на целые числа;
2) таблица должна иметь достаточный запас емкости.

При создании контейнера HashMap в конструкторе можно указать два параметра:

- емкость словаря - количество ячеек в таблице,
- коэффициент загрузки - долю заполненных ячеек, при которой емкость таблицы удваивается и производится рехеширование всех ключей. Опытным путем установили, что хеширование работает достаточно быстро при коэффициенте загрузки менее 0.8.

1.3. Класс Collections

Класс java.util.Collections это собрание статических методов для работы с коллекциями. С помощью его методов можно заполнять коллекции, сортировать их, искать элементы в коллекциях и делать другие операции. Основные методы этого класса приведены в таблице.

Метод	Описание
int binarySearch(List list, Object key)	Выполняет двоичный поиск ключа key в списке list*
copy(List dest, List src)	Копирует список src в список dest
void fill(List list, Object o)	Заполняет список list объектами o
Object max(Collection coll)	Находит наибольший элемент в коллекции coll*
Object min(Collection coll)	Находит наименьший элемент в коллекции coll*
void reverse(List list)	Инвертирует список list
void shuffle(List list)	Изменяет порядок элементов в списке list случайным образом
void sort(List list)	Упорядочивает список list естественным образом
void sort(List list, Comparator comp)	Упорядочивает список list при помощи компаратора comp
Collection synchronizedCollection(Collection coll)	Получает потокобезопасную коллекцию из коллекции coll*
Collection unmodifiableCollection(Collection coll)	Получает неизменяемую коллекцию из коллекции coll*

* Имеются аналогичные методы для всех контейнерных интерфейсов.

Пример. Сделать коллекцию неизменяемой.

⇓

```
Collection c = new ArrayList();
c.add("Celeron");
c.add("Dog");
c = Collections.unmodifiableCollection(c);

c.add("Error");   // метод add выбросит исключение
                  // UnsupportedOperationException
```

Чтобы воспрепятствовать разным потокам одновременно изменять содержимое контейнера, его можно сделать синхронизированным.

⇓

```
Collection c = Collections.synchronizedCollection(new ArrayList());
```

Контейнеры выбрасывают исключение ConcurrentModificationException, если происходит модификация их содержимого после того, как итератор начал проходить коллекцию.

Пример. Опасная ситуация: содержимое контейнера изменилось после получения его итератора.

⇓

```
Collection c = new ArrayList();
c.add("Celeron");
Iterator i = c.iterator();
c.add("Error");
```

Исключение возникнет, как только мы попытаемся воспользоваться пришедшим в негодность итератором.

⇓

```
i.next();      // выбросит исключение ConcurrentModificationException
```

1.4. Класс Arrays

Подобно классу Collections класс java.util.Arrays предоставляет дополнительный сервис для работы с массивами. Он состоит из четырех статических функций, которые перегружены для всех типов данных:

Метод	Описание
boolean equals(array1, array2)	Сравнивает два массива array1 и array2
void fill(array, value)	Заполняет все элементы массива array значением value
void sort(array)	Сортирует массив в естественном порядке
void sort(array, Comparator comp)	Сортирует массив при помощи компаратора comp
void binarySearch(array, key)	Выполняет двоичный поиск ключа key в массиве array

Пример. Создать целый массив, заполнить его данными и отсортировать.

⇓

```
import java.util.Arrays;
...
int [] m = new int[10];

Arrays.fill(m, 0, 3, 25);
Arrays.fill(m, 3, 7,  5);
Arrays.fill(m, 7, 10, 1);

Arrays.sort(m);
```

2-й и 3-й параметры функции fill определяют в массиве интервал заполнения, причем начало интервала входит в него, а конец – нет.

Простые типы сортируются только по возрастанию.

1.5. Задание

1 Определить класс MyString, который представляет собой неизменяемые строки, эквивалентные с точностью до регистра символов. При определении класса учесть, что его экземпляры будут использованы в качестве ключей словаря Map.

2 Построить частотный словарь текста, в котором каждому трехбуквенному сочетанию приписывается количество его вхождений в текст. Сочетания букв, которых в тексте нет, в словарь попадать не должны. Текст для анализа введите из файла. Сочетания в словаре не должны содержать пробелы и знаки препинания.

3 Разработать контейнерный класс «Телефонный справочник», который бы содержал произвольное количество абонентов. Каждый абонент имеет фамилию и один номер телефона. Фамилии разных

абонентов могут быть одинаковыми, номера телефонов – нет. Справочник должен выполнять поиск номеров телефонов по фамилии и поиск фамилии по номеру телефона.

2. МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ

Процесс - это выполняющаяся программа со своим собственным адресным пространством. Поток - это последовательность команд, исполняемых процессором. Каждый процесс может иметь один или более потоков. Механизм управления потоками (но не процессами) встроен в язык Java.

Преимущество многопоточности в том, что можно программировать длительные операции в отдельных потоках, не разбивая их на короткие отрезки. В однопоточной программе длительные операции "замораживали" бы диалог с пользователем и работу других потоков.

2.1. Создание потока путем расширения класса Thred

Чтобы создать поток вычислений, надо сконструировать объект класса, производного от Thread. После того как объект потока создан, его можно настроить, т.е. задать исходный приоритет, имя и т.д. Когда объект готов к работе, вызывается его метод start(), который порождает новый поток вычислений путем вызова другого метода объекта - run().

Когда метод run() возвращает управление, выполнение потока завершается.

Замечание. Метод start() можно вызвать только один раз, повторный вызов выбрасывает исключение IllegalThreadStateException.

Стандартная реализация метода run() не предполагает каких-либо действий. Чтобы заставить поток делать что-либо полезное, надо перекрыть метод run() в наследнике класса Thread.

Пример "Создание потока". Каждые полсекунды поток выводит сообщение на экран.

⇓

```
public class NewThread extends Thread {

  public void run() {
     try {
        for (int i = 5; i > 0; i-- ) {
           System.out.println("Дочерний поток: " + i);
```

```
                Thread.sleep(500);
            }
        } catch (InterruptedException ex) {
            // Исключение может выбросить метод sleep()
            System.out.println("Дочерний поток прерван.");
        }
        System.out.println("Дочерний поток завершен.");
    }
}
```

Из главного потока создадим дочерний поток.

⇓

```
(new NewThread()).start();
```

Это можно повторять многократно и каждый раз будет создаваться новый поток выполнения.

Имена потоков

Потоку можно присвоить имя – либо с помощью аргумента конструктора, либо методом setName(String). Текущее значение имени потока можно получить методом getName().

Имя потока не используется исполняющей системой и служит для удобства программиста. Тем не менее, поток обязан иметь имя, и, если оно не задано программистом, конструктор потока установит его сам.

Текущий поток

Ссылку на объект Thread текущего потока можно получить с помощью статического метода Thread.currentThread().

Пример. Вывести на экран имя потока, выполняющего метод main().

⇓

```
public static void main(String[] args) {
  System.out.println(Thread.currentThread().getName());
}
```

Все статические методы класса Thread относятся именно к текущему потоку.

Методы класса Thread

Основные методы класса Thread приведены в таблице.

Метод	Описание
void setName(String name)	Дать потоку имя name
String getName()	Получить имя потока
void setPriority(int priority)	Задать приоритет потока priority
int getPriority()	Получить приоритет потока
boolean isAlive()	Определить, выполняется ли еще поток this
void join()	Ожидание завершения потока this
void run()	Является точкой входа в поток
static void sleep(int msec)	Приостановить поток на msec миллисекунд
void start()	Запустить поток (вызывает run)

2.2. Создание потока путем реализации интерфейса Runnable

Наследование класса Thread мешает наследовать какой-либо другой класс, ведь множественного наследования в Java нет. Другой способ создавать потоки состоит в следующем.

1. Объявить класс MyThread, реализующий интерфейс Runnable, в составе этого интерфейса единственный метод - **void run()**.
2. Создать экземпляр класса MyThread.
3. Создать объект Thread, передав в конструктор экземпляр класса MyThread.
4. Вызвать метод start() объекта Thread.

Пример "Создание потока-2".

Создадим новый класс

⇓

```
public class MyThread implements Runnable {
  public void run() {
        ... тело метода такое же, как в предыдущем примере
  }
}
```

Из главного потока создадим дочерний поток.

⇓

```
(new Thread(new MyThread())).start();
```

Хотя мы явно не сохраняем ссылки на объект Thread, он делает это сам и поэтому не может быть удален сборщиком мусора.

2.3. Конфиденциальность

Рассмотрим некоторый сервер, который принимает задания, ставит их в очередь и обслуживает в порядке очереди.

⇓

```
public class MyServer implements Runnable {

  // Очередь запросов
  private LinkedList requests = new LinkedList();

  // Создать поток обслуживания в конструкторе
  public MyServer() {
      (new Thread(this)).start();
  }

  // Поставить работу в очередь
  public void addJob(Object job){
      requests.add(job);
  }

  // Процедура обслуживания
  public void run() {
      for(;;) {
          try {
              Object job = requests.removeFirst();
              System.out.println("[" + job + "]");
              Thread.sleep(100);  // сделать паузу
          } catch (Exception ex) {
          }
      }
  }
}
```

Поскольку метод run() принадлежит интерфейсу Runnable, он имеет уровень доступа public и может быть вызван кем угодно.

Чтобы воспрепятствовать этому, в метод run() можно вставить код, который получает ссылку на объект потока, вызвавшего run(), и сравнивает его с теми, которые имеют право вызывать run().

⇓

```
if (Thread.currentThread().getName().equals(…))
…
```

Более простое решение в том, чтобы не реализовать интерфейс Runnable, а объявить внутренний объект типа Runnable

⇓

```
public class MyServer {

  // Очередь запросов
  private LinkedList requests = new LinkedList();

  // Создание потока обслуживания
  public MyServer() {
      Runnable service = new Runnable() {
          public void run() {
              for(;;) {
                  try {
                      Thread.sleep(100);
                      Object job = requests.removeFirst();
                      System.out.println("[" + job + "]");
                  } catch (Exception ex) {
                  }
              }
          }
      };
      (new Thread(service)).start();
  }
}
```

2.4. Синхронизированные методы

Когда несколько потоков разделяют один ресурс, бывает необходимо синхронизировать их работу так, чтобы только один поток имел доступ к ресурсу в каждый момент времени.

Доступ к ресурсу регулируется при помощи монитора. Монитор – это объект, который используется для взаимоисключающей блокировки потоков (mutually exclusive lock). Когда один поток начинает выполнять любой из методов объекта-монитора (говорят, что поток входит в монитор), остальные, подойдя к монитору, останавливаются и ждут, пока монитор не освободится.

В Java монитором может служить любой объект, эта способность передается по наследству от класса Object.

Пример. Объектом служит банковский вклад. Поле balance содержит текущее количество денег на счету и не может быть меньше нуля.

Чтобы это условие не нарушилось при использовании объекта разными потоками, метод deposit() синхронизирован.

⇓

```
public class Account extends Thread {

  private long balance;     // текущая сумма вклада >= 0

  public Account (long value) {
      if (value > 0)
          balance = value;
      else
          balance = 0;
  }

  public synchronized void deposit(long value) {
      if (balance + value >= 0)
          balance += value;
  }

  public long getBalance() {
      return balance;
  }
  ...
}
```

Если бы метод deposit() не был синхронизирован, то между проверкой условия balance + value >= 0 и изменением баланса balance += value; мог вклиниться другой поток и изменить значение баланса.

Конструктор не должен быть синхронизированным, поскольку объект создается лишь однажды, а значит, только в одном потоке.

Замечание. Синхронизация еще один довод в пользу доступа к полям через методы. Непосредственный доступ к полям нельзя синхронизировать на стороне объекта.

Статические методы синхронизируются независимо от методов объекта. Монитором для них служит объект класса Class, который всегда создается для любого класса, используемого в программе.

2.5. Инструкция synchronized

Класс, экземпляр которого доступен многим потокам, может не иметь синхронизированных методов, и его исходный код может отсутствовать. В этом случае будет полезна инструкция synchronized. Она позволяет установить блокировку на участок кода, который использует некоторый объект (клиент объекта).

Пример. Предположим, что метод deposite() в классе Account не синхронизирован.

⇓

```
Account account = new Account(150);

synchronized (account) {
  account.deposite(-100);
}
```

Такой подход называется синхронизацией на стороне клиента. Недостатком синхронизации на стороне клиента является то, что она остается на совести клиента, и другие клиенты этого же объекта могут ею пренебречь.

Инструкция synchronized позволяет уменьшить объем синхронизированного кода и, тем самым, повысить эффективность многопоточной программы.

Пример. Синхронизирован не весь метод, а лишь его часть.

⇓

```
void deposit(long val) {
  synchronized (this) {
       if (balance + val >= 0)
           balance += val;
  }
  // что-нибудь еще
}
```

Инструкцию synchronized можно применять в различных проектных решениях, например, если объект имеет 2 группы методов и внутри каждой группы методы связаны, а между группами - нет, то можно применить 2 монитора, чтобы уменьшить блокировку.

⇓

```
class Worker {
```

```
Object lockA = new Object();
Object lockB = new Object();

// Методы группы A для взаимоблокировки применяют объект lockA
void methodFromA() {
    synchronized (lockA) {
    ...
    }
}

// Методы группы B для взаимоблокировки применяют объект lockB
void methodFromB() {
    synchronized (lockB) {
    ...
    }
  }
  ...
}
```

Замечание. Синхронизированный метод можно рассматривать как частный случай инструкции synchronized, когда монитором служит объект this.

Задача.

В заключение раздела рассмотрим полный код решения следующей задачи.

Имеется два потока. Каждый поток должен печатать свою собственную строку, переданную ему при создании.

В программе определены два класса: Concurrent и Worker.

Concurrent - класс потока. В конструкторе его экземпляр получает строку для печати и ссылку на объект Worker, который будет выполнять печать.

⇓

```
public class Concurrent implements Runnable {

  // Строка для печати
  private String msg;
  // Ссылка на исполнителя – объект, выполняющий печать
  private Worker worker;
```

```
// Конструктор, котрый запускает поток
public Concurr(String msg, Worker worker) {
    this.msg = msg;
    this.worker = worker;
    (new Thread(this)).start();
}

// Поток поручает всю работу объекту Worker
public void run() {
    worker.lazy(msg);
  }
}
```

Worker - класс для печати. Собственно печать выполняет метод lazy, который долго отдыхает после вывода каждого символа. Эта пауза нужна нам, чтобы на простом примере стали видны проблемы многопоточной работы.

⇓

```
class Worker {

  synchronized void lazy(String s) {
      for (int i = 0; i < s.length(); i++) {
          System.out.print(s.charAt(i));
          try {
              Thread.sleep(5);
          } catch (InterruptedException ex) {
          };
      }
  }
}
```

В методе main() создаются объекты и запускаются потоки.

⇓

```
public static void main(String[] args) {
  // Создать синхронизируемый объект
  Worker w = new Worker();

  // Запустить дочерние потоки
  Concurrent c1 = new Concurrent ("11111111111", w);
  Concurrent c2 = new Concurrent ("22222222222", w);
}
```

Если бы метод lasy был не синхронизирован, потоки печатали бы символы вперемешку.

2.6. Задание

1. Создать поток, который бы в течение примерно 5 сек печатал свое имя каждые полсекунды. Сделать это двумя способами - при помощи расширения класса Thread и при помощи реализации интерфейса Runnable.
2. Разработайте класс MyShadule, который получает массив интервалов времени в мсек и согласованный с ним массив сообщений, и выводит соответствующие сообщения на экран через заданные интервалы времени.
3. Создать класс с двумя отдельными счетчиками и объект этого класса. Создать несколько одинаковых потоков, каждый из которых повторяет следующее: сравнивает значение счетчиков и печатает результат сравнения, увеличивает первый счетчик, засыпает на 10 мсек, а затем увеличивает второй счетчик. Сравнить работу программы при условии, что упомянутый блок кода синхронизирован и не синхронизирован.
4. Написать программу, которая постарается создать как можно больше одновременно работающих потоков. Каждый поток естественно завершается после выполнения пустого цикла от 0 до Integer.MAX_VALUE/10. Проследите, как зависит количество одновременно работающих потоков от верхней границы цикла, и объясните эту зависимость.

3. ВЗАИМОДЕЙСТВИЕ ПОТОКОВ

3.1. Межпоточные связи

В предыдущем разделе потоки эксклюзивно использовали ресурсы, но не приостанавливали, не возобновляли и не прекращали работу друг друга, т.е. не взаимодействовали.

Для взаимодействия потоков применяются методы класса Object, которые можно разделить на методы ожидания и методы оповещения. Все они снабжены модификатором final и не могут быть переопределены в производных классах.

Метод	Описание
void wait(long timeout)	Переводит вызывающий поток в режим ожидания на время timeout
void wait()	То же, что wait(0)

void notify()	Пробуждает один из потоков, которые вызвали wait() на том же самом мониторе
void notifyAll()	Пробуждает все потоки. Первым будет выполняться поток с самым высоким приоритетом

Заметим, что все перечисленные методы следует вызывать лишь тогда, когда объект служит монитором, т.е. только из синхронизированного кода.

Перевод потока в режим ожидания

Метод wait() следует рассматривать как функцию, возврат из которой происходит не сразу, а после наступления одного их 4-х событий:

- вызова метода notify();
- вызова метода notifyAll();
- истечения интервала времени в timeout миллисекунд;
- вызова потоком собственного метода interrupt().

При вызове метода wait() блокировка с монитора снимается в отличие от метода sleep().

Типичный код для перевода потока в режим ожидания выглядит так.

⇓

```
synchronized void do() {
  while (условие ожидания)
      wait();
  // Сделать нечто, когда условие ожидания снимется
}
```

Оператор while нельзя заменять оператором if, поскольку поток может просыпаться от уведомлений, не связанных с условием его ожидания.

Типичный код для вывода потоков из режима ожидания выглядит так.

⇓

```
synchronized void changeCondition() {
  // Изменить условие ожидания
  // Послать уведомление всем ожидающим потокам
  notifyAll();
}
```

Если в момент вызова методов notify() и notifyAll() нет потоков в состоянии ожидания, уведомление, посланное методами, не запоминается.

Если выполнение метода wait() завершается ввиду истечения периода ожидания, поток "узнает" об этом только через оповещения, сделанные другими потоками. Если потоку необходимо узнать, подошло ли к концу время ожидания, он должен сделать это своими средствами. Задание периода ожидания следует рассматривать просто как страховку от бесконечного ожидания.

Задача.

Рассмотрим взаимодействие потоков на примере.

Есть производитель и потребитель данных - целых чисел. Имеется склад данных, способный хранить одно число, принимать и выдавать число со склада.

Необходимо так организовать работу системы, чтобы склад не переполнялся и чтобы потребитель не получал одно и то же число дважды.

Программа состоит из 3-х классов:
Producer - поток-производитель данных;
Consumer - поток-потребитель данных;
Store - склад.

Классы Producer и Consumer устроены просто.
⇓

```
class Producer implements Runnable {
 // Ссылка на объект-склад
 Store store;

 // Конструктор запускает поток-производитель
 Producer(Store store) {
     this.store = store;
     (new Thread(this)).start();
 }

 // Производитель отсылает числа на склад
 public void run() {
     for (int i = 0; ; i++)
         store.put(i);
 }
}

class Consumer implements Runnable {
 // Ссылка на объект-склад
```

```
Store store;

// Конструктор запускает поток-потребитель
Consumer(Store store) {
    this.store = store;
    (new Thread(this)).start();
}

// Потребитель получает числа со склада
public void run() {
    for (;;)
        store.get();
  }
}
```

Вся забота о синхронизации действий производителя и потребителя возлагается на объект-склад. Для этого:

1) класс Store имеет флаг hasData, который показывает, есть ли данные на складе;
2) если данных нет, надо заставить ждать поток, пришедший за данными;
3) если данные есть, надо заставить ждать поток, желающий сдать число на склад.

⇓

```
public class Store {
  // Хранимое на складе число
  private int n;
  // Признак наличия данных на складе
  private boolean hasData;

  synchronized int get() {
      // Если склад пуст, поток пришедший за данными, должен ждать
      while (!hasData) {
          try {
              wait();
          } catch (InterruptedException ex) {}
      }
  // Если число есть на складе, оно выдается
  hasData = false;
  notifyAll();
  System.out.println("Выдано со склада " + n);
  return n;
```

```
    }

    synchronized void put(int n) {
        // Если склад полон, поток принесший данные, должен ждать
        while (hasData) {
            try {
                wait();
            } catch (InterruptedException ex) {}
        }
        // Если склад пуст, число принимается на склад
        this.n = n;
        hasData = true;
        notifyAll();
        System.out.println("Поступило на склад " + n);
    }
}
```

В методе main() главный поток запускает два дочерних и тут же прекращается.

⇓

```
public static void main(String[] args) {
  Store store = new Store();
  Producer p = new Producer(store);
  Consumer c = new Consumer(store);
}
```

В результате получим следующий вывод:

```
        Поступило на склад 0
    Выдано со склада 0
    Поступило на склад 1
    Выдано со склада 1
    Поступило на склад 2
    Выдано со склада 2
        ...
```

Нетрудно убедиться, что если не вызывать методы wait() и notifyAll(), работа программы становится хаотической.

3.2. Приоритеты потоков

Потоки имеют приоритеты, которые определяют порядок передачи управления между ними. Правила передачи управления следующие.

- Если поток добровольно отказался от управления, оно переходит к потоку с самым высоким приоритетом.
- Если поток приостановлен потоком с более высоким приоритетом, то управление передается ему.
- Если приоритеты потоков равные, то поведение зависит от операционной системы.

Исходное значение приоритета потока совпадает с приоритетом потока-родителя. Приоритет может быть изменен вызовом метода Thread.setPriority() с аргументом из интервала MIN_PRIORITY-MAX_PRIORITY. По умолчанию приоритет потока равен NORM_PRIORITY (все константы определены в классе Thread).

Вообще говоря, непрерывно выполняющиеся потоки приложения должны обладать более низким приоритетом по сравнению с теми потоками, которые активизируются эпизодически.

Для потока, обновляющего экран задается приоритет, равный NORM_PRIORITY - 1, а потоку, обрабатывающему действия пользователя, задается приоритет, равный NORM_PRIORITY + 1. Лучше не впадать в крайности, назначая приоритеты, равные MIN_PRIORITY и MAX_PRIORITY.

Принудительное переупорядочение потоков

В классе Thread есть несколько методов, которые позволяют потоку уменьшить его потребность в процессорном времени.

- public static void **sleep**(long millis) throws InterruptedException - вынуждает текущий поток приостановить выполнение на указанный промежуток времени. Точность не гарантируется. Если во время спячки вызывается метод interrupt(), выбрасывается указанное исключение.
- public static void yield() - предлагает планировщику передать оставшуюся часть кванта времени следующему потоку. Планировщик может это предложение проигнорировать.

Замечание. Поскольку процесс планирования потоков всегда зависит от операционной системы, программисту на Java лучше избегать изменения приоритетов и вызова метода yield().

3.3. Взаимная блокировка потоков

В простейшем случае взаимная блокировка возникает, когда два потока используют два синхронизированных объекта.

Пусть первый поток вводит монитор в объект A, а второй поток вводит монитор в объект B.

После этого первый поток пытается ввести монитор в объект B, но переходит в ожидание, т.к. объект B занят вторым потоком. Второй поток не выполняется, он ждет освобождения объекта A. Получается, что оба потока не могут продолжать выполнение, возникает ситуация, которую иногда называют клинч или, более красочно, смертельное объятие.

Пример взаимной блокировки потоков.

⇓

```
public class DeadLocker extends Thread{
  Object a, b;
  public DeadLocker (Object a, Object b, String name) {
      super(name);
      this.a = a;
      this.b = b;
  }

  public void run() {
      // ввод монитора a
      synchronized (a) {
          System.out.println(getName() + ": step A");
          // передача процессора другому потоку
          sleep(0);
          // ввод монитора b
          synchronized (b) {
              System.out.println(getName() + ": step B");
          };
      };
  }
}
```

Для демонстрации взаимной блокировки напишем следующий код.

⇓

```
// Объекты a и b, которые послужат мониторами
Object a = new Object(),
Object b = new Object();

// Первый поток получает объекты a и b
new DeadLocker(a, b, "First").start();
```

```
// Второй поток получает те же объекты в другом порядке
new DeadLocker (b, a, "Second").start();
```

Этот код напечатает:

First: step A
Second: step A,

после чего потоки войдут в клинч.

Если вызов метода sleep(0) заменить на вызов метода a.wait(), клинча не произойдет.

3.4. Завершение потока

Поток продолжает действовать (метод потока isAlive() возвращает true), пока не произойдет одно из 3-х событий:

- метод run() естественно завершится;
- работа метода run() будет прервана;
- будет вызван метод потока destroy().

Вызов метода destroy() немедленно разрушает поток. Все блокировки, сделанные потоком остаются, поэтому другие потоки могут легко попасть в состояние бесконечного ожидания.

Многими системами метод destroy() не поддерживается и его применение вызывает исключение java.lang.NoSuchMethodError в вызывающем потоке. Применять для завершения потока метод destroy() нельзя.

Корректного прекращения работы потока можно добиться, поместив в цикл метода run() проверку флага завершения.

⇓

```
public class MyThread extends Thread {

  // Флаг завершения и метод для его установки
  private boolean stopFlag = false;

public void stopIt() {
    stopFlag = true;
}

// Поток, который печатает свое имя каждые полсекунды
public void run() {
    while (!stopFlag) {
```

```
            System.out.println(getName());
            try {
                sleep(500);
            }
            catch (InterruptedException ex) {
            }
        }
    }
}
```

Этот подход не учитывает того, что в момент желаемой остановки поток MyThread может выполнять длительный метод sleep() или wait(). Поэтому в классе Thread имеется механизм прерывания, состоящий из методов:

- public void interrupt();
- public static boolean interrupted();
- public boolean isInterrupted().

Метод interrupt() устанавливает состояние потока в значение "прерван" (сравни с установкой флага stopFlag = true). Это состояние диагностируется методами isInterrupted() и interrupted(), но последний еще и сбрасывает его. Теперь код примера упростится.

⇓

```
public class MyThread extends Thread{

    public void run() {
        while (!interrupted()) {
            System.out.println(getName());
            try {
                sleep(500);
            }
            catch (InterruptedException ex) {
                interrupt();
            }
        }
    }
}
```

Повторный вызов метода interrupt() в блоке catch необходим потому, что метод sleep(), выбросив исключение, так же очищает состояние потока "прерван", как и метод interrupted().

Стандартная обработка прерываний от методов sleep() и wait() обязана включать вызов метода interrupt() текущего потока.

⇓

```
try {
  …
  sleep(500);
}
catch (InterruptedException ex) {
  …
  Thread.currentThread.interrupt();
}
```

Замечание. До Java 2 для приостановки, перезапуска и остановки потока использовали методы suspend(), resume() и stop(). Эти методы не снимают блокировки, установленные потоком, и поэтому не используются.

3.5. Ожидание завершения потока

Если главный поток решил дожидаться завершения работы дочернего потока, он может сделать это, организовав цикл с проверкой условия isAlive() для объекта дочернего потока.

Чтобы ожидание не было активным, в цикле можно выполнять метод sleep()

⇓

```
public static void main(String[] args) throws Exception{
  MyThread thread = new MyThread();
  thread.start();
  while (thread.isAlive())
      Thread.sleep(100);
  System.out.println("Конец");
}
```

Понятно, что в среднем половина отрезка времени, указанного в аргументе метода sleep(), пропадает зря, поэтому метод sleep() надо заменить методом wait(), для чего нам потребуется синхронизированный код и объект в качестве монитора. Для этой цели не подойдет никакой объект, кроме объекта дочернего потока, т.к. именно он получит уведомление, когда поток завершится.

⇓

```
synchronized (thread) {
  while (thread.isAlive())
```

```
        thread.wait();
    }
```

Иными словами, если в качестве монитора синхронизированного кода использовать посторонний объект, его метод wait() будет выполняться неограниченно долго.

Остается оформить цикл ожидания в виде отдельного метода дочернего потока

⇓

```
public class MyThread extends Thread {
    ...
    synchronized void myJoin() throws InterruptedException {
        while (isAlive()) {
            wait();
        }
    }
```

и сказать, что такой полезный метод уже имеется в классе Thread и называется join().

3.6. Завершение работы приложения

Каждое приложение начинает работу с создания потока, который выполняет метод main(). Если приложение не создает других потоков, оно завершается, когда завершается метод main().

Все потоки, которые могут быть созданы в приложении, делятся на два вида: пользовательские и демоны. Незавершенные потоки-демоны разрушаются автоматически, когда заканчивается последний из пользовательских потоков.

Приложение завершается после окончания работы последнего пользовательского потока или после вызова метода exit() классов System или Runtime. При вызове метода exit() все незавершенные потоки немедленно разрушаются, а не получают естественного завершения.

3.7. Описатель volatile

Обращение к любым данным, допускающим изменение несколькими потоками, должно выполняться в синхронизированном коде.

Если один поток пишет, а остальные только читают, синхронизация не обязательна, но возникает опасность чтения не из оригинала, а из копии переменной, которую может создать компилятор при оптимизации кода.

Пример. Код, который периодически обновляет значение на экране.

⇓

```
currentValue = 5;
for (;;) {
  display.showValue(currentValue);
  Thread.sleep(1000);
}
```

Компилятор сочтет, что раз переменная не меняется внутри цикла, ее можно заменить константой, равной ее первоначальному значению.

Чтобы избежать подмены оригинала копиями, код надо синхронизировать или объявить переменную как volatile.

⇓

```
volatile int currentValue;
```

3.8. Задание

1. Разработать пример взаимной блокировки двух потоков, в котором используются только два объекта - оба потоки.
2. Создать поток, который печатал бы текущее время каждую секунду. Предусмотреть возможность его завершения путем ввода любого символа с клавиатуры.
3. Запустить стороннее приложение. Спустя 5 сек прервать его. Для контроля времени создать отдельный поток. Для работы с приложением воспользоваться классами java.lang.Runtime и java.lang.Process.

4. СЕРИАЛИЗАЦИЯ ОБЪЕКТОВ

Сериализация позволяет превратить объект в поток байтов, чтобы, когда понадобится, полностью восстановить объект из потока. Под словом поток здесь всюду понимается поток данных, а не поток выполнения, как в предыдущем разделе.

Сериализация необходима для сохранения объектов в постоянной памяти, для транспортировки параметров при удаленном вызове методов (RMI - Remote Methods Invocation), для сохранения на диске компонентов JavaBeans.

4.1. Интерфейс Serializable

Чтобы обладать способностью к сериализации, класс должен реализовать интерфейс-метку Serializable.

Для сохранения объекта в потоке необходимо надстроить выходной поток OutputStream потоком ObjectOutputStream, а после этого вызвать метод потока writeObject(), передав ему объект.

Пример. Сохранить объект в файле при помощи сериализации.

Для эксперимента определим класс с целым и строковым полями, реализующий интерфейс Serializable.

⇓

```
class A implements Serializable {
  int i;
  String s;
  public A(int ai, String as) {
       i = ai;
       s = as;
  }
}
```

Создадим объект и сохраним его в файле "aaa.obj".

⇓

```
// Создаем объект
A a = new A(1, "Hello");

// Создаем поток сериализации
ObjectOutputStream out = new ObjectOutputStream (
  new FileOutputStream("aaa.obj"));

// Сохраняем объект в потоке
out.writeObject(a);

out.close();
```

Для восстановления объекта необходимо надстроить входной поток InputStream потоком ObjectInputStream, а после этого вызвать метод потока readObject(). Этот метод создает объект в памяти и возвращает ссылку типа Object, поэтому следует выполнить нисходящее преобразование, чтобы получить объект нужного типа.

Пример. Восстановить объект, сохраненный в файле "aaa.obj".

⇓

```
// Создаем входной поток сериализации
ObjectInputStream in = new ObjectInputStream (
```

```
    new FileInputStream("111.obj"));

    // Читаем объект из потока и выполняем восходящее преобразование
    A a = (A)in.readObject();
```

В процессе восстановления ни один конструктор объекта не участвует, десериализация выполняется внеязыковыми средствами Java.

Если восстанавливающая программа не имеет доступа к описанию класса объекта, восстановление произойдет, но восходящее преобразование не выполнится и будет выброшено исключение ClassNotFoundException.

Сериализуемый объект может ссылаться на другие объекты. Если эти объекты реализуют интерфейс Serializable, то они сохраняются и восстанавливаются автоматически.

Пример. Сериализовать объект, ссылающийся на другой объект.

⇓

```
// Определяем класс A
class A implements Serializable { }

// Определяем класс B с полем-ссылкой на объект A
class B implements Serializable {
  A a;
  public B(A a) {this.a = a; }
}

// Создаем экземпляр класса B
B b = new B(new A());

// Сериализуем его, как в предыдущем примере
```

Размер и сложность сети объектов при сериализации не ограничена.

Если какой-то из подчиненных объектов не реализует интерфейса Serializable, будет выброшено исключение NotSerializableException.

Если есть несколько ссылок на один объект, то при сериализации он сохранится лишь однажды и восстановится в одном экземпляре. Циклические ссылки также не приводят к многократному сохранению объектов.

Бывает, что необходимо отказаться от сохранения в потоке некоторых полей объекта, например, пароля. Такие поля надо объявить со спецификатором transient.

⇓

```
class A implements Serializable {
  transient String s = "qwerty";
}
```

После восстановления объекта transient-поля будут равны null. Это надо учитывать при дальнейших операциях с объектом.

4.2. Интерфейс Externalizable

Процессом сериализации объекта можно управлять, если реализовать не интерфейс Serializable, а интерфейс Externalizable. Последний расширяет Serializable и включает два метода:

- writeExternal(ObjectOutput out) - сохранение объекта в потоке;
- readExternal(ObjectInput in) - восстановление объекта из потока.

Программируя эти методы, разработчик класса может реализовать тот способ сохранения-восстановления объекта, который сочтет нужным.

Пример. Реализовать в классе A интерфейс Externalizable.

⇓

```
class A implements Externalizable {
  String s;
  public A() {};
  public A(String s) {this.s = s;}

  public void writeExternal(ObjectOutput out) throws IOException {
  out.writeObject(s);
  }

  public void readExternal(ObjectInput in)
  throws IOException, ClassNotFoundException{
      s = (String)in.readObject();
  }
}
```

Прежде, чем сохраненный объект будет прочитан, он будет неявно создан, поэтому класс обязан иметь открытый конструктор без параметров.

Интерфейсы ObjectInput и ObjectOutput наследуют интерфейсы DataInput и DataOutput.

4.3. Управление сериализацией без Externalizable

Иной способ управлять сериализацией состоит в том, чтобы определить в классе, реализующем Serializable, закрытые методы:

- private void writeObject(ObjectOutputStream);
- private void readObject(ObjectInputStream).

В процессе сохранения-восстановления объекта эти методы будут вызываться вместо выполнения стандартной сериализации. При этом стандартная сериализация тоже остается доступной через вызов методов потоков:

- out.defaultWriteObject() - из метода writeObject(ObjectOutputStream out);
- in.defaultReadObject() - из метода readObject(ObjectInputStream in).

Пример выполнения стандартной сериализации из закрытого метода writeObject().

⇓

```
private void writeObject(ObjectOutputStream out) {
  out.defaultWriteObject();
   ...
}
```

4.4. Задание

1. Создать класс, представляющий собой связанный список точек (точка - два целых числа x и y). Чтобы класс имел возможность сериализации и восстановления, реализовать в нем интерфейс Serializable.
2. Сделать то же, реализовав интерфейс Externalizable.
3. Сериализовать циклический список строк и восстановить его.
4. Объявить подходящий класс и организовать в памяти дерево. Сериализовать и десериализовать объект дерева.

5. РЕФЛЕКСИЯ

Рефлексия - это механизм, позволяющий исследовать строение класса по его двоичному представлению. Механизм основан на классах из пакета java.lang.reflect и классах Class и Package из пакета java.lang.

Рефлексия позволяет создать объект, задав имя его класса в виде строки, и далее вызывать методы объекта и обращаться к его полям, но делать это надо только когда нельзя обойтись обычными средствами работы с классами и объектами.

5.1. Класс Class

Каждому типу java (простым типам, классам, интерфейсам и массивам) соответствует один объект класса Class. Есть особый объект и для слова void.

Объекты Class могут служить для получения информации о типах и для создания объектов и загрузки классов, если тип ссылочный.

Есть 4 способа получения объекта Class:

- методом объекта getClass();
- при помощи имени класса с суффиксом ".class";
- статическим методом Class.forName();
- при помощи методов рефлексии, которые возвращают объекты Class.

Пример. Три из четырех способов создания объектов Class.

⇓

```
// Методом объекта
Class c1 = "".getClass();

// При помощи суффикса class
Class c2 = String.class;

// Статическим методом класса Class
Class c3 = Class.forName("java.lang.String");
```

Ссылка на объект Class хранится также в поле TYPE оболочечных типов.

⇓

```
Class c4 = Integer.TYPE;
```

Некоторые методы класса Class приведены в таблице.

Метод	Описание
boolean isInterface()	Возвращает true, если объект Class представляет интерфейс
boolean isArray()	Возвращает true, если объект Class представляет массив
boolean isPrimitive()	Возвращает true, если объект Class представляет простой тип или void
Class[] getInterfaces	Возвращает массив объектов Class для реализуемых интерфейсов. Если таковых нет,

	массив будет нулевой длины
Class getSuperclass()	Возвращает объект Class для базового класса. Для Object, void и простых типов возвращает null. Для массива возвращает тот же объект Class, что и для Object
int getModifiers()	Возвращает числовой код модификатора: public, protected, private, final и т.п. Константы кодов объявлены в классе java.lang.reflect.Modifier
Class getComponentType()	Возвращает объект Class для типа элемента массива. Если текущий объект не массив, возвращает null
Class getDeclaringClass()	Возвращает объект Class для типа, внутри которого объявлен данный тип

Примеры использования методов класса Class.
⇓

```
// Получить информацию о классе X, создав объект Class
Class cx = Class.forName("untitled1.X");

// Создать экземпляр класса X
X x = (X) cx.newInstance();

// Получить и распечатать информацию об интерфейсах класса X
Class[] faces = cx.getInterfaces();
for (int i = 0; i < faces.length; i++)
  System.out.println(faces[i]);

// Получить и распечатать информацию о методах класса X
Method[] methods = cx.getMethods();
for (int i = 0; i < methods.length; i++)
  System.out.println(methods[i]);
```

5.2. Структура имен классов

Параметром метода Class.forName() должно быть полное (с указанием пакета) имя класса. Если класс является массивом, имя составляется из символа "[" и имени класса элемента с ";" в конце, например для массива из Object это будет "[java.lang.Object;". Имена двухмерных массивов начинаются с двух символов "[[" и т.д.

Простые типы кодируются буквами:
B – byte;

C – char;
D – double;
F – float;
...
Z – boolean.

Имена вложенных типов строятся так же, как имена файлов байт-кода внутренних классов: "Outer$Inner".

Локальные и анонимные классы не могут быть распознаны методами рефлексии.

5.3. Получение информации о членах класса

Информация о членах класса передается в объектах типа Field, Method, Constructor из пакета java.lang.reflect. Достают эту информацию разные варианты get-методов класса Class:

- для всех членов класса, относящихся к определенной категории;
- только для одного конкретного члена;
- для членов с определенным уровнем доступа;
- для членов только этого класса (не учитывая предков).

Пример применения рефлексии.

⇓

```
// Получить объект класса Class
Class c1 = Class.forName("java.lang.String");

// Напечатать имена всех публичных методов
Method[] methods = c1.getMethods();
for (int i = 0; i < methods.length; i++) {
  System.out.println(methods[i].getName());
}

// Напечатать имена всех частных методов
Method[] dmethods = c1.getDeclaredMethods();
for (int i = 0; i < dmethods.length; i++) {
int mod = dmethods[i].getModifiers();
if (Modifier.isPrivate(mod))
  System.out.println(dmethods[i].getName());
}

// Получить и вызвать конкретный метод
Method method = c1.getMethod("indexOf", new Class[] {String.class});
Integer i = (Integer)(method.invoke(
```

```
                "12345", new String[] {"34"}));
System.out.println(i);
```

Класс Modifier служит для кодирования всех модификаторов.

5.4. Проверка полномочий доступа

Классы Field, Method, Constructor являются производными от класса AccessibleObject, который может отключать проверку полномочий доступа. Доступ к членам класса посредством рефлексии подчиняется тем же ограничениям, что и доступ в обычном стиле – если поле private, то оно недоступно и для рефлексии.

Ситуацию можно изменить методом AccessibleObject.setAccessible(true) и сделать объект всегда доступным, например, для отладчика. Разумеется, для изменения уровня доступа нужны соответствующие полномочия.

5.5. Задание

1. По имени класса распечатать всех его предков и все интерфейсы, реализуемые этим классом и каждым его предком.

2. Имеется класс "X", который имеет метод без параметров "y", возвращающий фамилию автора класса. По заданным именам x и y распечатать фамилию автора класса. Сделать то же при условии, что метод статический.

6. ЗАГРУЗКА КЛАССОВ

6.1. Подготовка класса к работе

Каждый раз при компиляции нового класса для него создается объект типа Class и сохраняется в двоичном файле с расширением class.

Если в программе необходимо использовать некоторый класс X, то:

1) из файла X.class загружается объект типа Class;
2) выполняется связывание - проверка синтаксиса байт-кода, выделение памяти для статических членов, разрешение ссылок на другие классы с загрузкой этих классов при необходимости;
3) выполняется инициализация: инициализируется базовый класс (с предварительной загрузкой и связыванием) и выполняются все выражения и блоки статической инициализации.

6.2. Загрузчик классов ClassLoader

Загрузка класса из файла .class выполняется при помощи загрузчика классов - объекта, наследующего класс java.lang.reflect.ClassLoader.

Класс ClassLoader имеет два конструктора:

protected ClassLoader();
protected ClassLoader(ClassLoader parent).

Оба они создают новый загрузчик на базе "родительского", но первый конструктор использует в качестве родительского системный загрузчик классов.

Системный загрузчик - это тот объект, который использовался для загрузки основного класса приложения - того, чье имя было передано виртуальной машине для выполнения метода main(). Ссылка на системный загрузчик может быть получена статическим методом класса ClassLoader **getSystemClassLoader()**.

Ссылку на родительский загрузчик можно получить методом **getParent()**.

Основной метод загрузчика -

public Class **loadClass**(String name) throws ClassNotFoundException.

Он просматривает все места, где может находиться байт-код класса name, находит и загружает код класса, создавая тем самым объект Class, который и возвращает.

Порядок его работы таков:

1) вызывает **findLoadedClass()**, чтобы проверить по внутренней таблице, не загружался ли данный класс этим загрузчиком ранее;
2) если класс не загружался, вызывает **loadClass()** родительского загрузчика;
3) если класс все еще не загружен, вызывается метод **findClass()**.

Защищенный метод findClass() перекрывают, если хотят изменить стратегию поиска.

Метод findClass() должен выполнить две функции:

1) обнаружить байт-код заданного класса и сохранить его в массиве байтов;
2) выполнить фактическую загрузку класса.

Второе выполняется путем вызова метода defineClass():

protected final Class defineClass(String name, byte[] data, int offset, int length)

throws ClassFormatError

который получает байт-код класса в массиве data или в его части.

Этот же метод сохраняет ссылку на класс во внутренней таблице уже загруженных классов.

Если загруженный класс больше не нужен, способов его явной выгрузки не существует.

Каждый загрузчик создает собственное пространство имен, поэтому классы, загруженные разными загрузчиками, всегда будут считаться разными, даже если они считаны из одного файла .class.

Пример. Сделать загрузчик, который, вдобавок к обычной стратегии поиска, ищет классы еще и в корневом каталоге диска C.

⇓

```
class MyClassLoader extends ClassLoader {

  protected Class findClass(String name)  throws ClassNotFoundException
{
      try {
          byte[] buf = bytesForClass(name);
          return defineClass(name, buf, 0, buf.length);
      } catch (IOException ex) {
          throw new ClassNotFoundException(ex.toString());
      }
  }
}

// Читает файл <name>.class в массив байтов и возвращает массив
protected byte[] bytesForClass(String name)
throws IOException, ClassNotFoundException {
    FileInputStream in = null;
    try {
        String fname = "C:/" + name + ".class";
        in = new FileInputStream(fname);
        int len = in.available();
        if (len == 0)
            throw new ClassNotFoundException(name);
        byte[] buf = new byte[len];
        in.read(buf);
        return buf;
    } finally {
        if (in != null)
            in.close();
    }
  }
}
```

Методам loadClass() и findClass() имя класса передается в виде пути mypackage/MyClass или в стиле mypackage.MyClass.

⇓

```
MyClassLoader loader = new MyClassLoader();
Class c = loader.loadClass("mypackage/MyClass ");
// или Class c = loader.loadClass("mypackage.MyClass ");
```

6.3. Загрузка дополнительных ресурсов

Классы - это основной ресурс, в котором нуждается программа, но некоторым классам требуются дополнительные ресурсы, такие как фрагменты текста, изображения или данные мультимедиа.

В классе ClassLoader есть метод

InputStream getResourseAsStream(String name),

который может найти ресурс и открыть поток для его ввода.

Прототип этого метода возвращает null, поэтому класс загрузчика должен переопределить этот метод, реализовав в нем нужный алгоритм поиска ресурсов.

Системные ресурсы связаны с системными классами, а таковые могут не обладать собственными загрузчиками. Статический метод класса ClassLoader

InputStream getSystemResourceAsStream(String name)

возвращает объект потока для ресурса с заданным именем.

Пример. Получение потока для загрузки текстового файла.

⇓

```
MyClassLoader loader = new MyClassLoader();
InputStream in = loader.getResourceAsStream("myResource.txt");
...
```

6.4. Задание

1. Сделать загрузчик, который, в дополнение к CLASSPATH, разыскивает файлы и ресурсы в корневом каталоге диска A: (перекрыть методы findClass() и findResource()).
2. Создать 2 разных класса из одного и того же байт-кода, воспользовавшись для этого различными загрузчиками. Показать, что классы действительно разные.

7. СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Взаимодействие прикладной программы с виртуальной машиной и операционной системой осуществляется при помощи трех классов:

- класс System, который отвечает за системные свойства, стандартные потоки ввода-вывода и некоторые прикладные функции;
- класс Runtime, который обеспечивает взаимодействие с JVM;
- класс Process, представляющий сторонний процесс, запущенный из приложения.

7.1. Класс System

Стандартные потоки ввода-вывода

Стандартные потоки ввода-вывода представлены неизменяемыми статическими (public static final) полями класса System:

- InputStream in – стандартный поток ввода;
- PrintStream out – стандартный поток вывода;
- PrintStream err – стандартный поток ошибок.

Поток ошибок нужен, чтобы, при желании, можно было отделить выходную информацию программы от сообщений об ошибках.

Стандартные потоки можно перенаправить в любой поток подходящего типа. Для этого в классе System есть статические методы:

- setIn (InputStream) – изменить назначение для входного потока;
- setOut (PrintStream) – изменить назначение для выходного потока;
- setErr (PrintStream) – изменить назначение для потока ошибок.

Замечание. Хотя поля in, out и err объявлены как final, методы set…, реализованные native-кодом, способны обходить ограничения языка и изменять значение полей.

Пример. Направить сообщения об ошибках в файл "Errors.txt".

⇓

```
PrintStream e = new PrintStream(new FileOutputStream("Errors.txt"));
System.setErr(e);
System.err.println("Тестовое сообщение.");
e.close();
```

Чтобы вернуть стандартное назначение, нужно снова воспользоваться методом setErr().

⇓

```
System.setErr(System.out);
```

Системные свойства

Системные свойства определяют характеристики системного окружения и сохраняются классом System в виде коллекции типа Properties. Класс Properties расширяет старый контейнерный класс Hashtable.

Примеры системных свойств и их значений. Слева от знака равенства находится имя свойства, справа - его значение.

⇓

```
        java.version = 1.3.0_02
        java.home  = e:\jbuilder5\jdk1.3\jre
        java.class.path – путь к классам Java
        os.name = Windows NT
os.version = 4.0
        file.separator = /
        path.separator = ;
        user.home = личный каталог пользователя;
        user.dir = рабочий (текущий) каталог;
```

Класс System имеет открытые статические методы, позволяющие управлять системными свойствами, они показаны в таблице.

Метод	Описание
String getProperty(String key)	Возвращает значение свойства key
String setProperty(String key, String value)	Устанавливает новое значение value свойства key и возвращает старое значение
Properties getProperties()	Возвращает коллекцию Properties
void setProperties(Properties props)	Устанавливает системные свойства, сохраняемые в коллекции props

Пример. Изменить рабочий каталог.

⇓

```
System.setProperty("user.dir", "C:/111");
```

Прикладные методы

Для измерения отрезков времени можно использовать метод

```
public static long currentTimeMillis();
```

который возвращает количество миллисекунд, прошедших с начала эпохи, т.е. с 00:00:00 по Гринвичу, 1 января 1970 г.

Пример. Измерить время выполнения фрагмента программы.

⇓

```
long t = System.currentTimeMillis();
 // Здесь расположен измеряемый фрагмент программы
System.out.println(System.currentTimeMillis() - t);
```

Замечание. Точность таких измерений не высока

Для работы с массивами может пригодиться метод

```
public static void arraycopy(
    Object src, int src_position, Object dst, int dst_position, int count);
```

который копирует содержимое массива-источника в массив-приемник. Источником и приемником может служить один и тот же массив. С помощью метода можно безопасно выполнять сдвиг элементов массива как вправо, так и влево.

Пример. Циклически сдвинуть все элементы массива m вправо на 1 элемент.

⇓

```
int r = m[m.length - 1];
System.arraycopy(m, 0, m, 1, m.length - 1);
m[0] = r;
```

7.2. Класс Process

Из Java-программы можно запустить другой процесс с помощью метода объекта Runtime

```
public Process exec(String command) throws IOException;
```

который активизирует команду операционной системы, представляемую строкой command. Метод возвращает объект Process, соответствующий созданному процессу.

Методы класса Process относятся к двум категориям: методы, которые позволяют получить ссылки на стандартные потоки данных и методы для управления процессом.

Стандартные потоки данных процесса

Метод

```
public OutputStream getOutputStream();
```

возвращает поток, который служит для процесса стандартным потоком ввода. Если в этот поток что-то записать, то процесс сможет это прочесть из своего потока System.in.

Метод

public InputStream getInputStream();

возвращает поток, который служит для процесса стандартным потоком вывода. Из этого потока можно прочитать все, что процесс направил в стандартный вывод System.out.

Метод

public InputStream getErorrStream();

возвращает поток, который процесс воспринимает как поток ошибок. Из этого потока можно прочитать сообщения об ошибках, которые процесс направил в System.err.

Благодаря этим методам, приложение, запустившее внешний процесс, может кормить его данными и получать его выходную информацию для анализа и обработки.

Пример. Из текущего приложения запустить процесс, который выполнит программу "extern.exe" в рабочем каталоге, и подсоединиться с его потоками ввода-вывода.

⇓

```
Process proc = Runtime.getRuntime().exec("extern.exe");
// Потоки соединяет метод plugTogether(), код которого приведен ниже
plugTogether(System.in, proc.getOutputStream());
plugTogether(proc.getInputStream(), System.out);
plugTogether(proc.getErrorStream(), System.err);
```

Метод plugTogether() запускает отдельный поток выполнения (не путать с потоками данных), который периодически проверяет наличие данных во входном потоке и, если данные есть, переписывает их во входной поток.

⇓

```
static Thread plugTogether( final InputStream in, final OutputStream out)
{
 // Внутренний класс потока выполнения для переноса данных
 Thread thread = new Thread() {
     public void run() {
         while (!interrupted()) {
             try {
                 if (in.available() > 0) {
                     // Если в потоке есть данные, переписать их
```

```
                out.write(in.read());
                // Обязательно выгрузить в поток буфер
                out.flush();
            } else {
                // Если данных нет, сделать паузу
                sleep(100);
            }
        }
        catch (InterruptedException ex) {
            interrupt();
        }
                }
                catch (Exception ex) {
                }
            }
        }
    };

    // Создать поток выполнения
    thread.start();

    // Вернуть объект потока, чтобы его можно было прервать извне
    return thread;
}
```

Управление состоянием процесса

Метод

public int waitFor() throws InterruptedException;

приостанавливает поток, запустивший внешний процесс, до момента завершения последнего. Метод возвращает значение кода завершения. Если к моменту вызова метода процесс уже закончен, метод сразу же возвращает код завершения.

Метод

public int exitValue();

возвращает код завершения процесса. Если процесс еще не завершен, метод выбрасывает исключение IllegalThreadStateException.

Метод

public void destroy();

уничтожает текущий процесс. Если процесс уже завершен, метод ничего не делает.

Другие варианты метода exec()

Другая версия метода Runtime.exec() позволяет задавать переменные окружения процесса.

```
public Process exec(String command, String[] env)
throws IOException.
```

Каждый элемент массива env содержит строку вида "имя=значение". В имени не должно быть пробельных символов, а в значении они допускаются.

Существует вариант метода exec(), который позволяет устанавливать рабочий каталог запускаемого процесса.

```
public Process exec(String command, String[] env, File dir)
throws IOException.
```

Если параметр dir не задан или равен null, дочерний процесс имеет тот же рабочий каталог, что и родительский.

Другие методы класса Runtime

public void gc();
форсирует процесс сборки мусора.

public void exit(int status);
завершает приложение и запускает процессы - ловушки закрытия.

public void halt(int status);
завершает приложение, но ловушки закрытия не запускает.

public void loadLibrary(String libname);
загружает динамическую библиотеку native-кода из каталога, отведенного операционной системой под хранение библиотек, например Winnt/System32.

public void load(String filename);
загружает динамическую библиотеку из произвольного каталога.

public void traceInstructions(boolean on);
предлагает виртуальной машине выдавать отладочную информацию после каждой инструкции кода.

public void traceMethodCalls(boolean on);

разрешает или запрещает трассировку вызовов методов.

7.3. Задание

1. Выполнить в виде внешнего процесса команду операционной системы, например, DIR. Команду и параметры передать в командной строке процесса. Результат выполнения получить в виде отдельной строки.
2. Организовать контроль длительности запущенного процесса. Сделать так, чтобы процесс принудительно завершался, если в течение заданного отрезка времени он не выдает информации в стандартный поток вывода.

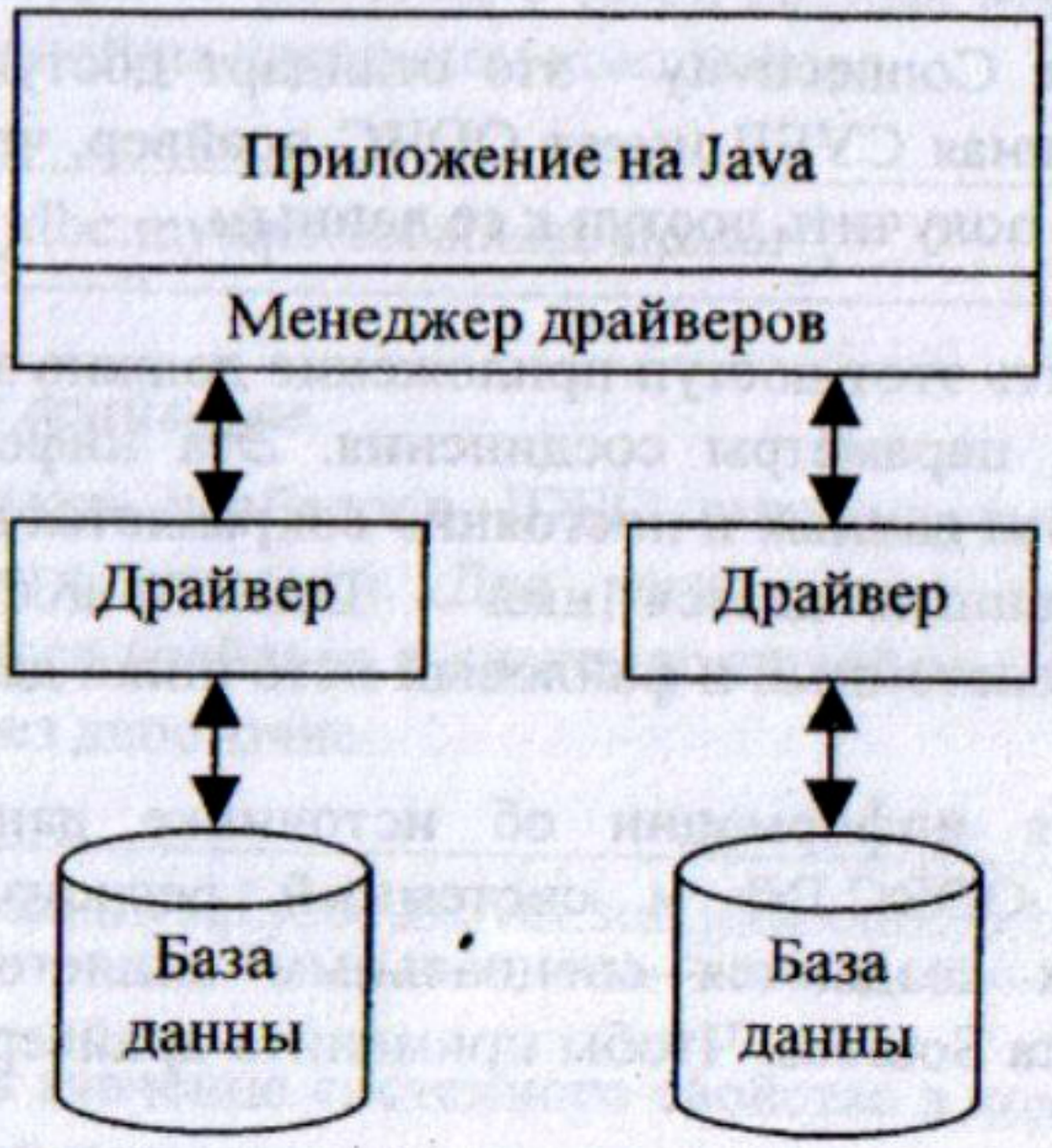
8. ПРОГРАММНЫЙ ИНТЕРФЕЙС JDBC

8.1. Структура JDBC

JDBC (Java DataBase Connectivity) - это платформно-независимая технология, позволяющая из программы на Java получить доступ к любой SQL-совместимой базе данных. Чтение этого раздела требует от читателя знания принципов организации баз данных и основ структурированного языка запросов - SQL.

Основными составляющими JDBC являются драйверы баз данных и класс для управления драйверами - DriverManager. Множество драйверов открыто, в появлении и обновлении заинтересованы как программисты на Java, так и производители СУБД (систем управления базами данных).

Схема связи приложения, использующего JDBC, с базой данных выглядит так.



В задачу менеджера драйверов входит присоединение Java-приложений к требуемому драйверу JDBC.

Драйвер поддерживает обмен данными между приложением и базой данных.

Типы драйверов

Различают четыре типа драйверов JDBC.

1. **Мост JDBC-ODBC.** Этот тип служит связующим звеном между приложением и ODBC-драйвером и требует наличия на клиентской машине драйвера ODBC.
2. **Полу-Java драйверы.** Этот тип преобразует вызовы JDBC в вызовы клиентского API конкретных СУБД и требует наличия на компьютере исполняемого не-Java кода клиентской части СУБД.
3. **Сетевые JDBC-драйверы.** Эти драйверы транслируют программные вызовы JDBC в независимый от СУБД сетевой протокол, который затем транслируется в протокол СУБД специальным промежуточным сервером.
4. **Драйверы СУБД.** Этот тип драйверов преобразует программные вызовы JDBC в сетевой протокол, родной для данной СУБД.

Драйверы 3 и 4 типов являются предпочтительным способом доступа к базам данных. Третий тип более распространен в Интернет, 4-й - во внутренних сетях. Драйверы 1 и 2 типов следует использовать, когда нет драйверов 3 и 4 типа.

Что такое ODBC

Open Database Connectivity - это стандарт доступа к базам данных. Каждая промышленная СУБД имеет ODBC-драйвер, через который любое приложение может получить доступ к ее данным.

Чтобы получить этот доступ приложение должно знать: имя драйвера, имя базы данных, параметры соединения. Эта информация называется описанием источника данных и постоянно сохраняется на диске.

Источнику данных дается имя - Data Source Name. Различают пользовательский, системный и файловый источники данных.

Для хранения информации об источнике данных ОС Windows использует файл ODBC.INI и системный регистр. В ОС Windows источники данных создаются специальным апплетом из контрольной панели - ODBC Data Sources. Чтобы применить драйвер-мост JDBC-ODBC,

предварительно надо создать системный источник данных при помощи этого апплета.

8.2. Установление соединения с базой данных

Инсталляция драйверов

Драйверы 3 и 4 типов написаны на Java, поэтому соответствующие классы должны быть доступны и путь к ним прописан в переменной среды CLASSPATH.

При использовании моста JDBC-ODBC должен быть создан источник данных DSN.

URL-адрес базы данных

В JDBC описанием базы данных служит ее URL-адрес, который строится по образцу описателя ресурсов Интернет и имеет вид:

протокол:подпротокол:информация_для_соединения

В качестве протокола всегда указывается "jdbc".

Подпротокол идентифицирует драйвер JDBC. Это может быть db2 - для коммерческой СУБД DB-2, mysql - для бесплатной СУБД MySQL, odbc - для драйвера-моста и т.п.

Информация_для_соединения зависит от выбранного драйвера и может включать имя хоста, номер порта, имя источника данных ODBC, параметры соединения и другие элементы.

Пример. URL для соединения с базой данных MySQL под названием mentor, расположенной на локальном компьютере.

⇓

```
String url = "jdbc:mysql://localhost/mentor";
```

Регистрация драйверов

Чтобы менеджер драйверов JDBC смог управлять драйвером, он должен быть зарегистрирован. Для регистрации драйвера достаточно упомянуть имя класса драйвера в системном свойстве jdbc.drivers, там они перечисляются через двоеточие.

⇓

```
jdbc.drivers=com.mysql.jdbc.Driver:sun.jdbc.odbc.JdbcOdbcDriver
```

Можно задать значение системного свойства в командной строке при запуске виртуальной машины.

⇓

```
java -Djdbc.drivers=com.mysql.jdbc.Driver
```

Но часто драйвер регистрируют, просто загружая нужный класс по его имени.

⇓

```
Class.forName("com.mysql.jdbc.Driver");
```

Соединение с базой

Чтобы получить доступ из программы к данным базы необходимо:

зарегистрировать драйвер, задать URL и создать объект соединения при помощи одного из статических методов getConnection() класса DriverManager.

Класс DriverManage, класс Connection, и другие классы, необходимые для JDBC находятся в пакете java.sql.

Пример. Установить соединение с базой данных MySQL "mentor".

⇓

```
try {
  Class.forName("com.mysql.jdbc.Driver");
  String url = "jdbc:mysql://localhost/mentor";
  Connection con = DriverManager.getConnection(url, "root", "1");
catch (ClassNotFoundException ex) { … }
```

Второй и третий параметры метода getConnection() означают имя пользователя и пароль соответственно.

8.3. Выполнение SQL-команд

После того как соединение установлено, можно переходить к выполнению SQL-операторов и вообще любых команд, которые понимает СУБД, т.к. синтаксис передаваемых команд не ограничен. Тем не менее, JDBC драйверы гарантированно воспринимают некое подмножество языка SQL, которое можно охарактеризовать как вводный уровень (entry level) ANSI SQL-2 и далее мы будем говорить о командах SQL.

Для выполнения команд в программе используют три интерфейса:

- Statement - для операторов SQL без параметров;
- PreparedStatement - для операторов SQL с параметрами и часто выполняемых операторов;
- CallableStatement - для исполнения хранимых в базе процедур.

Интерфейсы PreparedStatement и CallableStatement расширяют интерфейс Statement, поэтому имеют все его методы.

Объекты-носители интерфейсов создаются при помощи методов объекта Connection:

- Метод createStatement() возвращает объект Statement;
- метод prepareStatement() возвращает объект PreparedStatement;
- метод prepareCall() возвращает объект CallableStatement.

8.4. Интерфейс Statement

Интерфейс Statement предоставляет три различных метода выполнения SQL-выражений:

- executeQuery() - для запросов, результатом которых является один единственный набор значений, таких как запросы SELECT. Метод возвращает набор данных, полученный из базы.
- executeUpdate - для выполнения операторов INSERT, UPDATE или DELETE, а также для операторов DDL (Data Definition Language). Метод возвращает целое число, показывающее, сколько строк данных было модифицировано.
- execute() - используется, когда операторы SQL возвращают более одного набора данных, здесь эта возможность не рассматривается.

Пример "Запрос данных". Создадим объект Statement и выполним запросы с его помощью.

⇓

```
try {
  Class.forName("com.mysql.jdbc.Driver");
  String url = "jdbc:mysql://localhost/mentor";
  Connection con = DriverManager.getConnection(url, "root", "1");

  Statement st = con.createStatement();

  String sql;
  // Выполнить SQL-оператор изменения данных
  sql = "INSERT INTO tutor (id_tutor, name) VALUES (123, 'Иванов')";
  int n = st.executeUpdate(sql);

  // Выполнить SQL-оператор выборки данных
  sql = "SELECT * FROM student";
  ResultSet rs = st.executeQuery(sql);

  // Закрытие оператора высвободит ресурсы
```

```
    st.close();
  }
  catch (ClassNotFoundException ex)
  {
  }
  catch (SQLException ex)
  {
  }
```

8.5. Интерфейс ResultSet

Метод executeQuery() возвращает объект с интерфейсом ResultSet, который хранит в себе результат запроса к базе данных. Набор данных представляет собой множество записей, разделенных на поля. Подходящей метафорой для этого набора является таблица, строки которой - это записи, а колонки - поля.

В наборе данных есть курсор, который может указывать на одну из строк таблицы, эта строка называется текущей. Курсор перемещается по строкам при помощи методов next(), previous(), first() и last(). Методы возвращают true, если курсор действительно указывает на строку и false, если это не так.

Сразу после получения набора данных его курсор находится перед первой строкой, чтобы сделать первую строку текущей надо вызвать метод next().

Поля текущей записи (колонки таблицы) доступны программе при помощи методов интерфейса ResultSet: getInt() getFloat(), getString(), getDate() и им подобных. Аргументом методов служит имя колонки или ее номер, нумерация колонок ведется с 1. Эти методы пытаются преобразовать данные набора к типам языка Java и, как правило, это можно сделать несколькими способами.

Пример "Получение данных". В базе данных, с которой уже установлено соединение, находится таблица Student, ранее созданная оператором

⇓

```
CREATE TABLE Student (
  id_student INT PRIMARY KEY,
  name VARCHAR(30),
  average FLOAT
)
```

Послать запрос к базе данных и распечатать строки полученного набора. Обработка исключений опущена.

⇓

```
// Получить набор данных
Statement st = con.createStatement();
ResultSet rs = st.executeQuery("SELECT * FROM Student");

// Распечатать набор данных
while (rs.next()) {
  // Напечатать значения в текущей строке.
  int id_student = rs.getInt("id_student");
  String name = rs.getString("name");
  float average = rs.getFloat("average");
  System.out.println("Строка = " + id_student + " " + name + " " +
average);
  }
  // Закрыть объект оператора
  st.close();
```

Хотя интерфейс ResultSet имеет метод close(), вызывать его не обязательно, т.к. набор данных будет закрыт вместе с объектом Statement, который его получил, или после получения этим оператором другого набора данных.

Значения NULL

Некоторые поля таблиц в базах данных могут не иметь значения. Те методы ResultSet.getXxx(), которые преобразуют данные в объектный тип, возвращают в этом случае значение null. Те методы, которые преобразуют данные в простые типы, возвращают 0. Чтобы отличить не инициализированные поля от полей с нулевым значением, можно воспользоваться методом ResultSet.wasNull().

Большие значения

Некоторые поля таблиц способны хранить очень большие значения. Это могут быть строки, двоичные объекты, изображения и т.п. Если программа не готова принять большое значение стразу, например, методом getString() или getObject(), она может сделать это частями. Для этого есть такие методы как getBinaryStream(), getAsciiStream(), getCharacterStream(), которые возвращают поток байтов, однобайтовых ASCII-символов и двухбайтных символов Unicode соответственно.

Пример. Читать большое значение частями и направлять его в другой поток.

⇓

```
Statement st = con.createStatement();
ResultSet rs = st.executeQuery(
    "SELECT biography FROM Student WHERE id_student=123");
// Теперь считываем колонку результатов частями по 4 K:
byte[] buff = new byte[4096];
while (rs.next()) {
    InputStream fin = rs.getAsciiStream("biography");
    for (;;) {
        int size = fin.read(buff);
        if (size == -1) {
            // в конце потока
            break;
        }
        // Отослать заполненный буфер в поток:
        System.out.write(buff, 0, size);
    }
}
st.close();
```

8.6. Интерфейс PreparedStatement

Экземпляры PreparedStatement сохраняют скомпилированные SQL-выражения. Это приводит к более быстрому выполнию запросов с их помощью.

Другой особенностью SQL-выражений в PreparedStatement является то, что они могут иметь параметры. Параметризованное выражение содержит знаки вопроса в своем тексте. Перед выполнением запроса значение каждого вопросительного знака устанавливается методами setXxx().

Пример. Выполнить запрос с параметрами.

⇓

```
// Создать и откомпилировать запрос с параметрами
PreparedStatement st = con.prepareStatement(
"SELECT * FROM student WHERE id_student > ? AND name = ? ");

// Установить значение параметров
st.setInt(1, 100);
```

```
st.setString(2, "ОЛЕГ");

// Выполнить перекомпилированный запрос
ResultSet rs = st.executeQuery();
```

Однажды подготовив оператор запроса, можно многократно повторять установку новых значений параметров и получать новые наборы данных.

8.7. Пример приложения для работы с MS Access

Конечно, для серьезных приложений рекомендуется использовать большие СУБД и JDBC-драйверы 3 и 4 типа. Тем не менее, если на вашем компьютере установлены ОС Windows и Microsoft Office, вы имеете возможность изучать технологию Java для работы с данными без установки дополнительного программного обеспечения. В качестве СУБД можно использовать MS Access, а в качестве средства доступа к данным – драйвер-мост jdbc-odbc.

Рассмотрим законченный пример маленького приложения для ввода запросов и получения ответов от базы данных под управлением MS Access.

Для построения пользовательского интерфейса в примере задействованы классы библиотеки Swing.

Приложение имеет единственную форму с текстовым полем JTextField для SQL-оператора, таблицей JTable для отображения результата запроса и кнопкой JButton для выполнения запроса. Класс формы за исключением двух методов приведен ниже.

⇓

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

public class MyFrame extends JFrame {
  // Поля формы для организации пользовательского интерфейса
  private JTextField txtSql = new JTextField();  // для запроса
  private JTable grdResult = new JTable();    // для ответа
  private JButton btnRun = new JButton();    // кнопка
  private JPanel pane = new JPanel();
  private JScrollPane scrollPane = new JScrollPane();
  private JButton btnRun = new JButton();
```

```
// Оператор Statement для выполнения запросов
private Statement statement;

// Конструктор формы
public MyFrame() {
    try {
        // Соединиться с базой данных
        connectDb("jdbc:odbc:mentor");
        // Инициализировать форму
        jbInit();
    }
    catch(Exception e) {
        e.printStackTrace();
    }
}

// Метод для инициализации формы
private void jbInit() throws Exception {
    txtSql.setText("");
    btnRun.setText("Run");
    btnRun.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(ActionEvent e) {
            showResult();
        }
    });
    this.getContentPane().add(pane, BorderLayout.NORTH);
    pane.add(txtSql, null);
    pane.add(btnRun, null);
    this.getContentPane().add(scrollPane, BorderLayout.CENTER);
    scrollPane.getViewport().add(grdResult, null);
    this.setSize(new Dimension(400, 300));
}

// Метод для соединения с базой
void connectDb(String dsname) { код метода показан ниже }

// Метод для отображения результата  запроса
void showResult() { код метода показан ниже }

// Точка входа в приложение
public static void main(String[] args) {
```

```
        MyFrame frame = new MyFrame();
        frame.setVisible(true);
    }
}
```

Метод для соединения с базой получает URL-адрес базы и выполняет соединение через мост jdbc-odbc.

⇓

```
void connectDb(String dbUrl)
throws ClassNotFoundException, SQLException {
  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
  Connection con = DriverManager.getConnection( dbUrl, "", "");
  statement = con.createStatement();
}
```

Метод для демонстрации результатов собирает в одномерный вектор наименования колонок, а в двухмерный - содержимое полей. По этим двум векторам он строит таблицу.

⇓

```
void showResult() {
  ResultSet rs;
  Vector rowData = new Vector();
  Vector columnNames = new Vector();

  try {
    // Получить набор данных
    String sql = txtSql.getText();
    rs = statement.executeQuery(sql);

    // Получить метаданные
    ResultSetMetaData meta = rs.getMetaData();

    // Определить количество колонок и их имена
    int colCount = meta.getColumnCount();
    for (int i = 0; i < colCount; i++) {
      columnNames.add(meta.getColumnLabel(i+1));
    }

    // Собрать данные в двухмерный вектор
    for (; rs.next();) {
      Vector v = new Vector();
```

```
            for (int i = 0; i < colCount; i++) {
                v.add(rs.getObject(i + 1));
            }
            rowData.add(v);
        }

        // Создать новую таблицу с данными и вставить в панель
        grdResult = new JTable(rowData, columnNames);
        scrollPane.getViewport().add(grdResult, null);

        } catch (SQLException ex) {
            scrollPane.getViewport().remove(grdResult);
        }
    }
```

8.8. Задание

1. Создать базу данных с таблицей Books и полями: id_book, author, title, content. Написать приложение, которое сможет пополнять эту таблицу. Содержание книги должно извлекаться из соответствующего текстового файла.
2. Дополнить таблицу books полем picture, которое будет хранить изображение обложки книги. Написать приложение с графическим интерфейсом, которое пополняет таблицу books и позволяет просматривать ее содержимое.

9. СЖАТИЕ И РАЗБОР ДАННЫХ

9.1. Сжатие файлов в формате ZIP

Алгоритм Лемпела-Зива

В основе процедуры упаковки в формат ZIP лежит следующая идея, известная как алгоритм Лемпела-Зива.

За один проход по тексту строится и словарь кодирования, и кодируется текст. Вначале словарь содержит только одно пустое слово, оно имеет код 0. Далее в тексте последовательно выделяются слова. Выделяемое слово - это максимально длинное слово из тех, что уже накоплены в словаре плюс еще один символ. В сжатый текст записывается код имеющегося в словаре слова и дополнительный символ, а словарь пополняется новой комбинацией. Словарь кодирования не хранится, так как он динамически восстанавливается по сжатому тексту.

Упаковка данных

Классы, необходимые для упаковки и распаковки данных расположены в пакете java.util.zip.

Упаковка данных состоит в их выводе в специальный байтовый поток ZipOutputStream, содержимое которого обычно сохраняется в файле. В упакованных данных можно создать несколько секций, которые можно извлекать, удалять и добавлять независимо друг от друга. Далее секции будем называть входами.

Пример. Упаковать файлы “111.txt” и “111.doc” в файл “111.zip”.

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

  // Создать поток сжатия
  String fout = "111.zip";
  ZipOutputStream out = new ZipOutputStream(new
FileOutputStream(fout));

  // Упаковать 2 файла, каждый в отдельную секцию
  pack("111.txt", out);
  pack("111.doc", out);
  out.close();
}
```

Упаковку одного файла выполняет метод pack().

⇓

```
// Упаковывает файл по имени fin
static void pack(String fin, ZipOutputStream out) throws IOException {

  // Открыть вводной файл
  FileInputStream in = new FileInputStream(fin);

  // Создать вход
  out.putNextEntry(new ZipEntry(fin));

  // Выполнить сжатие
  int c;
  while((c = in.read()) != -1)
      out.write(c);

  in.close();
```

```
}
```

Задача без использования буферизации, поэтому программа работает долго.

Надстроим буфер на входе.

⇓

```
BufferedInputStream in = new BufferedInputStream(new FileInputStream(fin));
```

и на выходе.

⇓

```
ZipOutputStream out = new ZipOutputStream(
                        new BufferedOutputStream(
                          new FileOutputStream(fout)));
```

Распаковка данных

Распаковка данных состоит в их чтении из специального байтового потока ZipInputStream, содержимое которого хранится в архивном файле.

Пример. Извлечь все упакованные файлы из архива “111.zip”.

⇓

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

  // Создать входной поток
  ZipInputStream in = new ZipInputStream(
                        new BufferedInputStream(
                          new FileInputStream("111.zip")));

  // Распаковать все имеющиеся входы
  ZipEntry entry;
  while ((entry = in.getNextEntry()) != null) {
      unpack(in, entry.getName());
    }
  }
```

Пока метод in.getNextEntry() возвращает непустую ссылку, входы в архиве есть.

Распаковку одного входа выполняет метод unpack().

⇓

```
// Распаковывает очередной вход в файл по имени fout.
static void unpack(ZipInputStream in, String fout) throws IOException {

  // Создать выходной поток
  BufferedOutputStream out = new BufferedOutputStream(
                             new FileOutputStream(fout));
  // Выполнить распаковку
  int c;
  while((c = in.read()) != -1) {
      out.write(c);
  }

  out.close();
}
```

9.2. Разбор данных

Разбор - это переработка потока символов в последовательность лексем.

Для разбора данных в библиотеке Java существуют два класса: java.io.StreamTokenizer и java.util.StringTokenizer.

Английский глагол tokenize означает "размечать", поэтому объекты данных классов будем называть разметчиками.

9.3. Класс StreamTokenizer

Класс StreamTokenizer расширяет класс Object. При создании объект StreamTokenizer получает входной поток символов java.util.Reader, из которого он будет черпать материал для разбора. В результате разбора получается последовательность лексем, которые становятся доступны клиенту разметчика одна за другой.

Работа разметчика регулируется таблицами и флагами.

Каждый символ потока имеет один или более из следующих атрибутов: пробельный символ, буква, цифра, строковая кавычка, символ комментария. Информация о том, какие из атрибутов имеет тот или иной символ, содержится в синтаксических таблицах разборщика.

Имеется четыре флага, которые устанавливают режим разбора:

- конец строки рассматривать как лексему;
- распознавать и пропускать комментарии в стиле C;
- распознавать и пропускать комментарии в стиле C++;
- делать все символы лексем строчными.

Чтобы использовать разметчик, необходимо:

1) сконструировать объект;
2) если необходимо, настроить его синтаксические таблицы и флаги;
3) повторять вызовы метода nextToken() до тех пор, пока он не вернет лексему с типом TT_EOF.

После каждого вызова nextToken() поле type содержит тип извлеченной лексемы: TT_WORD, TT_NUMBER, TT_EOL, TT_EOF.

Если тип лексемы TT_WORD, очередная лексема находится в поле String sval.

Если тип лексемы TT_ NUMBER, очередная лексема находится в поле double nval.

Задача.

Открыть файл с текстом и распечатать все его слова в столбик.

⇓

```
public static void main(String[] args)  throws Exception{
  // Создать исходный поток символов
  Reader in = new BufferedReader(new InputStreamReader(
                              new FileInputStream("111.txt")));
  // Создать объект StreamTokenizer
  StreamTokenizer st = new StreamTokenizer (in);

  // Сделать разбор
  while (st.nextToken() != st.TT_EOF) {
      if (st.ttype == st.TT_WORD)
          System.out.println(st.sval);
  }
}
```

Если поместить в файл "111.txt" оператор

Reader in = new BufferedReader(new InputStreamReader(
new FileInputStream("111.txt")));

получим набор слов

⇓

```
Reader
in
new
BufferedReader
new
InputStreamReader
new
```

```
FileInputStream
```

Включим в стандартный перечень разделителей букву 'R'

⇓

```
// Это надо сделать после создания разметчика, но до начала разбора
st.whitespaceChars('R', 'R');
```

получим другой результат

⇓

```
eader
in
new
Buffered
eader
...
```

9.4. Класс StringTokenizer

Класс java.util.StringTokenizer имеет назначение сходное с классом StreamTokenizer, но разбирает строку, а не поток и устроен намного проще.

Исходные данные для разбора содержатся в строке. Есть множество символов, выполняющих роль разделителей, остальные используются для построения лексем.

Настройка разметчика происходит во время выполнения конструктора.

В конструкторе

StringTokenizer(String str, String delim, boolean returnDelims)

параметр str означает разбираемую строку;

параметр delim задает множество символов-разделителей (в форме строки);

параметр returnDelims определяет, будут ли разделители рассматриваться как лексемы (значение - true) или нет (значение - false).

Помимо этого есть еще два конструктора, которые сводятся к уже рассмотренному:

```
public StringTokenizer(String str, String delim) { this(str, delim, false); }
public StringTokenizer(String str)        { this(str, " \t\n\r\f", false); }
```

Пример. Разобрать строку на слова.

⇓

```
StringTokenizer st = new StringTokenizer("This is a test.");
```

```
while (st.hasMoreTokens()) {
  System.out.println(st.nextToken());
}
```

9.5. Задания

1. Реализуйте простейшую форму алгоритма Лемпела-Зива, изложенную в начале раздела. Сравните его по результатам работы с каким-нибудь промышленным вариантом того же алгоритма.
2. Напишите программу для упаковки содержимого всех файлов заданного каталога и распаковки того, что упаковано.
3. Сделайте рекурсивный вариант предыдущей программы. Упаковываться должно все дерево каталогов, "растущее" из заданного каталога.
4. Выполните разбор текста программы на Java, чтобы подсчитать количество операторов цикла, условных и присваивания, которые в ней содержатся. Используйте класс StreamTokenizer.
5. Выполните разбор содержимого текстового файла, чтобы подсчитать количество слов и предложений, которые в нем содержатся. Используйте класс StringTokenizer.

10. ДОКУМЕНТАЦИЯ

10.1. Комментарии и тэги

Помимо комментариев в стиле C и C++, в языке Java имеются комментарии документатора, по которым автоматически строятся программные документы в формате HTML.

Комментарий документатора бывает многострочный

⇓

```
/**
 * Многострочный
 * комментарий
 * документатора.
 */
```

и однострочный

⇓

```
/** Однострочный комментарий. */
```

И те, и другие могут содержать тэги HTML.

Первое предложение текста комментария должно быть самым информативным и оканчиваться точкой.

После текста комментария следуют тэги документатора. Исключение составляют тэги @link - перекрестные ссылки, которые могут находиться внутри текста.

Общий формат тэга @link:

{@link package.class#member label}

Перекрестная ссылка позволяет быстро перейти на описание члена member класса class из пакета package. Если имеется в виду член текущего класса, элементы class и package не нужны. Ссылка будет представлена в документе текстом label, а если таковой отсутствует, то именем члена package.class.member.

Пример тэга @link, в котором отсутствуют элементы package, class и label.

⇓

```
/**
 * The current set of system properties for use by the
 * {@link #getProperty(String)} method is returned as a
 * <code>Properties</code> object.
 */
```

10.2. Комментарий класса

Этот комментарий располагается непосредственно перед заголовком класса. После текста располагаются тэги документатора. В комментарии класса допускаются тэги:

{@link} - гиперссылка;

@see - разновидность гиперссылки;

@since - версия программного обеспечения, в которой впервые появился данный класс;

@deprecated - пометка, что класс не рекомендован к использованию;

@author - разработчик класса;

@version - версия класса.

Пример комментария класса.

⇓

```
/**
 * A class representing a window on the screen.
 * For example:
```

```
 * <pre>
 *   Window win = new Window(parent);
 *   win.show();
 * </pre>
 *
 * @author  Sami Shaio
 * @version 1.0
 * @see    java.awt.BaseWindow
 * @see    java.awt.Button
 */
class Window extends BaseWindow {
  ...
}
```

10.3. Комментарий поля

Допустимые тэги комментария: @see, {@link}, @since, @deprecated.

Пример комментария поля.

⇓

```
    /**
     * The X-coordinate of the component.
     *
     * @see #getLocation()
     */
    int x = 1263732;
```

В тэге @see точка между именем класса и именем члена класса заменяется знаком #.

10.4. Комментарий метода

Допустимые тэги: @see,{@link}, @since, @deprecated;
@param - описание одного параметра;
@return - описание возвращаемого значения;
@throws - перечень исключений, возможных при выполнении метода;
@exception - синоним тэга @throws.

Пример комментария метода.

⇓

```
    /**
```

```
* Returns the character at the specified index. An index
* ranges from <code>0</code> to <code>length() - 1</code>.
*
* @param     index  the index of the desired character.
* @return    the desired character.
* @exception StringIndexOutOfRangeException
*            if the index is not in the range <code>0</code>
*            to <code>length()-1</code>.
* @see       java.lang.Character#charValue()
*/
public char charAt(int index) {
  ...
}
```

Для простых методов, таких как getName(), тэг @return может быть опущен как избыточный. Аналогично можно опустить тэги @param в описании конструкторов, если смысл параметров ясен из комментария к конструктору.

10.5. Комментарий пакета

Комментарий пакета записывается в виде HTML-текста в файл **package.html,** где package - имя пакета. Сам файл помещается в каталог исходных текстов пакета, где его найдет документатор. Ограничители комментария /** */ писать не обязательно.

Допустимые тэги: @see,{@link}, @since, @deprecated.

Пример комментария пакета.

⇓

```
<html>
 <body>
  <p>Пакет оболочки для тестов.</p>
 </body>
</html>
```

10.6. Параметры и опции документатора

Документация составляется утилитой **javadoc.** Эта утилита поставляется компанией Sun Microsystems, но возможно создание пользовательских документаторов с целью изменить вид выходного документа, обработать пользовательские тэги, осуществить побочные операции и т.п.

При помощи утилиты javadoc можно документировать как отдельные классы, так и пакеты и группы пакетов.

Документирование одного или более пакетов

Файлы пакета должны находиться в одноименном каталоге. Если имя пакета составное, каталоги должны быть вложенными.

Пример. Документировать два пакета: java.awt и java.awt.event .
Оба пакета находятся в каталоге C:\home\src\ .
Файлы документов необходимо разместить в каталоге C:\home\doc\.

Вариант 1. Сначала сделать исходный каталог текущим.
⇓

```
C:> cd C:\home\src\
C:> javadoc -d C:\home\doc java.awt java.awt.event
```

Расположение будущих документов задает опция **–d**.

Вариант 2. Воспользоваться опцией **–sourcepath** для указания пути к пакетам.
⇓

```
C:> javadoc -d C:\home\doc -sourcepath C:\home\src java.awt
java.awt.event
```

Опция **-sourcepath** применяется только при документировании пакетов.

Документирование одного или более классов

Пример. Документировать класс и классы, имена которых начинаются со слова Graphics.
Все классы находятся в каталоге C:\home\src\java\awt .
Файлы документов необходимо разместить в каталоге C:\home\doc.

Вариант 1. Сначала сделать исходный каталог текущим.
⇓

```
C:> cd C:\home\src\java\awt
C:> javadoc -d C:\home\doc Button.java Graphics* .java
```

Вариант 2. Если указывать полные имена классов, текущим может быть любой каталог.

⇓

```
C:> javadoc -d C:\home\doc C:\home\src\java\awt\Button.java
   (продолжение строки)  C:\home\src\java\awt\Graphics*.java
```

10.7. Задание

1. Снабдите комментариями документатора задания предыдущего раздела.

2. Сгенерируйте документацию при помощи утилиты javadoc.

Часть 3. Графический интерфейс пользователя

1. ПРИЛОЖЕНИЯ С ОКОННЫМ ИНТЕРФЕЙСОМ

Так называемый графический интерфейс пользователя давно стал стандартом общения человека с компьютерной программой. Java предлагает разработчикам независимые от платформы средства построения такого интерфейса.

В настоящее время в составе ядра JDK имеются два пакета, которые можно использовать для этой цели. Первый пакет - jawa.awt (Abstract Windowing Toolkit) появился еще в версии JDK 1.0. Он содержит минимально необходимый набор компонентов, сравнительно просто организован и довольно неказист. Второй пакет – javax.swing вошел в Java2 и представляет собой обширную библиотеку компонентов, пригодную для построения пользовательских интерфейсов любой сложности.

Мы начнем с краткого знакомства с jawa.awt, а потом перейдем к javax.swing, тем более, что многие классы библиотеки Swing базируются на классах awt.

1.1. Пакет jawa.awt

Пакет jawa.awt содержит набор классов для создания окон и вывода графики. Классы, входящие в пакет, можно разделить на 3 категории.

- *Компоненты* – кнопки, меню, списки, диалоговые окна и др. элементы GUI.
- *Менеджеры расположения* компонент в контейнерах.
- *Графика* – цвета, шрифты, изображения, фигуры.

В отдельных пакетах содержатся классы для работы с буфером обмена (java.awt.datatransfer), для обработки событий (java.awt.event) и работы с изображениями (java.awt.image).

Компоненты

Чтобы дать представление о структуре пакета, приведем фрагмент дерева наследования его классов.

```
Component – общий предок всех компонентов.
  |
  +--Button – кнопка.
```

```
|
+--Checkbox – переключатель.
|
+--TextComponent – текстовый элемент.
|       |
|       +--TextField – однострочное текстовое поле.
|       |
|       +--TextArea – многострочная текстовая область.
|
        +--Container – компонент, который может вмещать другие компоненты.
                |
                +--Panel – простейший контейнер компонент.
                |       |
                |       +--Applet – маленькое приложение, работающее
                |                   внутри web-браузера.
                +--ScrollPane - контейнер с возможностью прокрутки.
                |
                +--Window - общий предок окон.
                        |
                        +--Frame – полнофункциональное окно.
                        |
                        +--Dialog – диалоговое окно.
```

```
MenuComponent - предок элементов, из которых строятся меню.
 |
 +--MenuBar - главное меню окна.
 |
 +--MenuItem - пункт меню.
     |
     +--Menu – подменю.
```

Вершинами иерархии компонентов служат классы Component и MenuComponent, эти классы происходят непосредственно от Object.

Компоненты содержатся в контейнере (класс Container), который размещает их при помощи менеджера расположения. Программист добавляет компоненты в контейнер методом **add()** и задает менеджер расположения посредством метода **setLayout()**.

Среди подклассов класса Container широко используются три: Frame, Dialog и Panel.

Менеджеры расположения

Поскольку приложения, написанные на Java, могут выполняться на разных платформах и в самых различных условиях (размер окна, шрифты, разрешение экрана), большое значение придается способности окна автоматически подстраиваться под эти условия и сохранять, насколько это возможно, приличный вид.

Автоматическую настройку не только окон, но и других контейнеров должны выполнять объекты классов, производных от класса LayoutManager. Разные подклассы реализуют различные способы расположения компонентов в контейнере:

- FlowLayout – последовательное, компоненты занимают места слева направо и сверху вниз.
- GridLayout – табличное, компоненты находятся в ячейках таблицы одинакового размера.
- BorderLayout – полярное, все компоненты прижимаются к сторонам контейнера, а один расположен в центре.
- CardLayout – карточное, компоненты расположены на карточках, и в каждый момент времени видна только одна карточка.
- GridBagLayout - компоненты находятся в ячейках таблицы, различного размера и формы.

1.2. Порядок разработки оконного приложения

Определяем класс главного окна – наследника класса java.awt.Frame. При этом:

1) ссылки на элементы управления и менеджер расположения делаем полями этого класса;
2) в выражениях инициализации создаем элементы управления и объект избранного менеджера расположения;
3) в конструкторе класса или в отдельном методе, вызываемом из конструктора, устанавливаем в окно менеджер расположения и добавляем в окно элементы управления;
4) там же устанавливаем свойства управляющих элементов и самого окна.

В методе main или другом методе, вызываемом из main:

1) создаем объект окна;
2) задаем размеры и местоположение окна на экране;
3) делаем окно видимым.

Пример "Окно с кнопкой". Разработать приложение с одним окном и кнопкой на нем. При нажатии на кнопку ничего не происходит.

В файле MyFrame.java определим класс главного окна.
⇓

```
import java.awt.*;

public class MyFrame extends Frame {

  // Объявляем и создаем менеджер расположения
  BorderLayout manager = new BorderLayout();
  // Объявляем и создаем кнопку
  Button button = new Button();

public MyFrame () {
    try {
        // Настраиваем окно и элементы управления
        init();
    }
    catch(Exception e) {
        e.printStackTrace();
    }
}

// Отдельный метод для настройки окна и элементов

private void init() throws Exception {
    // Делаем надпись на кнопке
    button.setLabel("Ok");
    // Задаем размеры окна в пикселях
    this.setSize(new Dimension(400, 300));
    // Устанавливаем строку в заголовке окна
    this.setTitle("Frame Title");
    // Устанавливаем менеджер расположения в окне
    this.setLayout(manager);
    // Добавляем кнопку в окно, задавая при этом
    // ограничения, специфичные для данного менеджера
    this.add(button, BorderLayout.SOUTH);
}

// Точка входа в программу

public static void main(String[] args) {
    // Создаем объект окна
    MyFrame frame = new MyFrame();
```

```
            // Делаем окно видимым
            frame.setVisible(true);
        }
    }
```

1.3. Графика

Важным абстрактным классом awt является Graphics - графический контекст. Он представляет собой унифицированный интерфейс графической системы. У Graphics есть свойства, которые отражают текущее состояние графического контекста и различные методы рисования и закраски.

К свойствам контекста относятся: цвет фона, текущий цвет, прямоугольник рисования, положение точки с координатами (0, 0), режим рисования.

Текущий цвет служит для рисования линий, написания текста и закраски.

Некоторые методы класса Graphics приведены в таблице.

⇓

Метод	Описание
Color getColor()	Возвращает текущий цвет
setColor(Color color)	Устанавливает текущий цвет color
void setPaintMode()	Устанавливает режим обычного рисования
void setXORMode(Color color)	Устанавливает режим рисования XOR. Пиксели текущего цвета изменяют свой цвет на цвет color и обратно. Остальные цвета меняются непредсказуемо, но обратимо
Font getFont()	Возвращает установленный в контексте шрифт
void setFont(Font font)	Устанавливает шрифт в контексте
void setClip(int x, int y, int w, int h)	Устанавливает область рисования
Shape getClip()	Возвращает область рисования
void copyArea(int x, y, w, h, dx, dy)	Копирует прямоугольник x, y, w, h
void drawLine(int x1, y1, x2, y2)	Рисует линию между точками 1 и 2
void fillRect(int x, int y, int w, int h)	Закрашивает прямоугольник цветом переднего плана
void clearRect(int x, int y, int w, int h)	Закрашивает прямоугольник цветом

	фона
void drawRect(int x, int y, int w, int h)	Рисует прямоугольник цветом переднего плана
void drawOval(int x, int y, int w, int h)	Рисует овал
void drawArc(int x, y, w, h, start, fin)	Рисует дугу овала
void drawPolygon(int[] xPoints, int[] yPoints, int nPoints)	Рисует замкнутый многоугольник
drawString(String str, int x, int y)	Рисует строку текста
void dispose()	Удаляет графический контекст*

* Вызывается только для контекстов, созданных программистом методом Component.getGrphics() или Grphics.create(), а не полученных в виде параметра методов paint() или update().

Помимо класса Graphics для рисования используются вспомогательные классы: Color, Shape, Rectangle, Polygon, Font, Image.

Получить ссылку на объект Graphics программист может двумя способами:

1) как параметр метода рисования, такого как **Component.paint(Graphics g)** или **update()**;
2) вызвав метод **Component.getGraphics()**.

Как правило, все действия по рисованию сосредотачивают в особом методе компонента - paint(), который будет вызываться исполняющей системой всякий раз, когда окно потребует перерисовки, т.е. при первом появлении окна на экране, при изменении размеров окна, при повреждении окна в результате перекрытия другим окном и т.п. Принудительно вызвать перерисовку окна можно методом repaint().

Пример "Красный квадрат". Нарисовать красный квадрат на белом фоне.

Объявим класс формы, который расширяет java.awt.Frame и перекроем его метод paint().

⇓

```
import java.awt.*;

public class MyFrame extends Frame {

  // Метод, в котором сосредоточено рисование
```

```
public void paint(Graphics g) {
    // Цвет переднего плана - красный.
    g.setColor(Color.red);
    // Определяем размеры окна
    int w = this.getWidth();
    int h = this.getHeight();
    // Определяем сторону квадрата
    int q = w < h ? w / 2 : h / 2;
    // Рисуем красный квадрат
    g.fillRect(w / 4, h / 4, q, q);
}

// Точка входа в программу

public static void main(String[] args) {
    // Создаем объект окна
    MyFrame frame = new MyFrame();
    // Устанавливаем первоначальный размер окна
    frame.setSize(new Dimension(400, 400));
    // Делаем окно видимым
    frame.setVisible(true);
  }
}
```

1.4. Задание

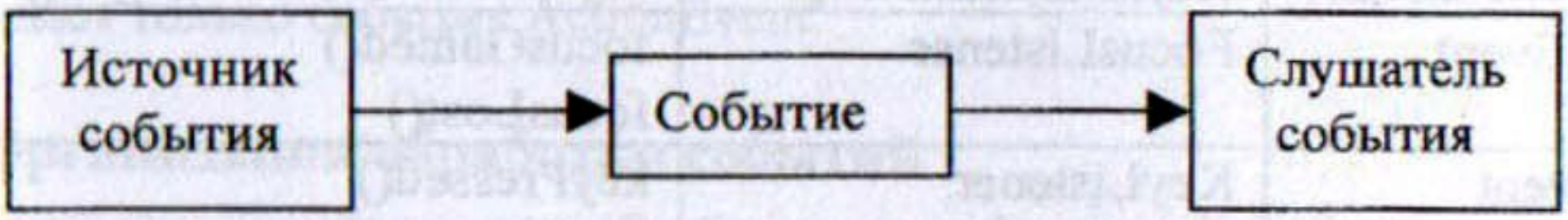
1. Создать приложение с окном, которое моделирует функциональную часть клавиатуры, т.е. три группы по четыре кнопки с надписями F1, F2, …, F12. Видимая группировка должна сохраняться при изменении размеров окна. Никаких действий при нажатии кнопок не выполнять.
2. Создать приложение с окном, в котором постараться разместить по одному экземпляру каждого компонента библиотеки AWT. Выбрать наиболее подходящий для этой цели менеджер расположения.
3. Нарисовать в окне лист школьной тетради в клеточку, шахматную доску, лист тетради в косую линию.
4. Нарисовать 20 концентрических окружностей разного цвета с центром, расположенным в центре окна. Количество окружностей в окне не должно зависеть от размеров окна.
5. Нарисовать спираль с центром, расположенным в центре окна.

2. ОБРАБОТКА СОБЫТИЙ В AWT

В процессе работы с программой пользователь перемещает по экрану курсор мыши, нажимает на кнопки мыши и клавиатуры, открывает и закрывает окна и т.п. Все действия пользователя, и не только они, расцениваются программой как последовательность событий, на которые она должна как-то реагировать. Обработка событий в программе распределена между компонентами, т.е., если событие относится к какому-то компоненту, например, пользователь нажал кнопку в окне, то именно кнопка и должна начать его обработку.

Метод, выполняющий обработку события, может располагаться в любом месте программы, он может принадлежать как объекту кнопки, так и любому другому объекту. В этом случае говорят, что кнопка делегирует обработку события другому объекту.

В общем случае компонент-источник направляет созданный им объект события другому объекту, который называется слушатель. Именно слушатель и должен отреагировать на событие, иными словами, обработать событие.



2.1. Событие

Любое событие в библиотеке AWT описывается подклассом абстрактного класса **java.awt.AWTEvent**.

С каждым событием связан объект-источник. Узнать его можно при помощи метода

Object getSource().

Каждому событию присвоен идентификатор (закрытое поле id), узнать который можно методом

int getID().

Уже готовые подклассы для описания конкретных событий находятся в пакете **java.awt.event**. Это MouseEvent, ActionEvent, FocusEvent и др.

2.2. Интерфейсы слушателей

Объект, которому делегируют обработку некоторого события, называется слушателем события, а передача ссылки на него объекту-источнику события называется регистрацией слушателя в источнике.

Чтобы объект имел право быть слушателем, его класс должен реализовать интерфейс слушателя. Например, любые слушатели события ActionEvent должны реализовать интерфейс ActionListener. В пакете **java.awt.event** объявлен целый ряд интерфейсов слушателей. Все интерфейсы слушателей расширяют интерфейс-метку **java.util.EventListener.**

Один интерфейс слушателя может включать несколько методов. Каждый метод обрабатывает какой-то один вид события, например, нажатие кнопки мыши, отпускание кнопки мыши, перемещение мыши. По установленному соглашению методу передается один единственный аргумент-объект события, адресованного данному слушателю.

Некоторые классы событий и соответствующие интерфейсы слушателей приведены в таблице.

Класс события	Интерфейс слушателя	Методы слушателя
ActionEvent	ActionListener	actionPerformed()
AdjustmentEvent	AdjustmentListener	adjustmentValueChanged()
FocusEvent	FocusListener	focusGained() focusLost()
KeyEvent	KeyListener	keyPressed() keyReleased() keyTyped()
MouseEvent	MouseListener	mouseClicked() mouseEntered() mouseExited() mousePressed() mouseReleased()
	MouseMotionListener	mouseDragged() mouseMoved()
WindowEvent	WindowListener	windowActivated() windowDeactivated() windowOpened() windowClosing() windowClosed() windowIconifed() windowDeiconifed()

2.3. Адаптеры слушателей

Для каждого интерфейса слушателя, в котором есть более одного метода, пакет java.awt.event имеет класс-адаптер, который этот интерфейс тривиально реализует пустотелыми методами. Расширяя класс-адаптер, программист реализует только те методы интерфейса слушателя, которые необходимы в данной программе.

Если бы каждый интерфейс слушателя имел ровно один метод, то адаптеры были бы не нужны.

2.4. Регистрация слушателей

В пакете AWT источником событий всегда является какой-нибудь класс пакета. Для регистрации слушателя в источнике используются специальные методы источника, имена которых подчиняются следующему правилу. Если событие называется **XxxEvent**, то регистрируется слушатель методом **addXxxListener()**, а удаляется методом **removeXxxListener()**.

Благодаря этому соглашению по методам компонента легко определить, источником каких событий он может являться. Например, класс Button имеет лишь один метод addActionListener, значит Button порождает только события ActionEvent.

2.5. Организация обработки событий

Если программа должна обрабатывать события, то необходимо:

1) выбрать событие, которое надо обработать и создать для него объект слушателя;
2) в метод слушателя поместить собственно код обработки;
3) зарегистрировать слушателя в компоненте-источнике события.

Пример "Окно с кнопкой - 2". Создать окно и кнопку в центре окна. При нажатии на кнопку в заголовке окна должна появиться надпись "Кнопка была нажата".

Воспользуемся программой примера "Окно с кнопкой" из предыдущего раздела и добавим туда обработку события нажатия на кнопку - ActionEvent. Для этого в конце метода init() достаточно приписать:

⇓

```
// Объявить внутренний класс слушателя
class MyListener implements ActionListener {
public void actionPerformed(ActionEvent e) {
        Frame1.this.setTitle("Кнопка была нажата");
    }
```

```
    }

    // Создать объект слушателя
    MyListener listener = new MyListener();

    // Зарегистрировать слушателя в объекте-источнике
    button.addActionListener(listener);
```

Напомним, что синтаксис Frame1.this используется для обращения к внешнему объекту из кода внутреннего класса.

Поскольку все три действия составляют одно целое, их часто программируют при помощи анонимного класса.

⇓

```
    // Экономпакет - три в одном.
    button.addActionListener( new ActionListener() {
      public void actionPerformed(ActionEvent e) {
          Frame1.this.setTitle("Кнопка была нажата");
      }
    });
```

Пример "Завершить приложение". Сделать так, чтобы при нажатии на кнопку приложение завершалось.

Здесь все то же самое, кроме одной строчки.

⇓

```
    button.addActionListener( new ActionListener() {
      public void actionPerformed(ActionEvent e) {
          System.exit(0);
      }
    });
```

2.6. Рисование при помощи мыши

Напишем программу, которая будет делать видимым след от перемещения курсора мыши по поверхности формы, если перемещать мышь с нажатой левой кнопкой.

Движение мыши воспринимается программой как последовательность событий мыши MouseEvent. Это позволяет изображать траекторию курсора в виде серии коротких отрезков прямой линии, по одному отрезку на каждое элементарное перемещение курсора. Чтобы нарисовать один

отрезок, обработаем один из видов события MouseEvent - перемещение мыши с нажатой левой кнопкой.

Концом очередного отрезка будем считать точку расположения курсора мыши в событии перемещения. Началом отрезка будем считать конец предыдущего отрезка.

Это событие обрабатывает слушатель MouseMotionListener при помощи метода mouseDragged().

⇓

```
addMouseMotionListener(new MouseMotionAdapter() {
  public void mouseDragged(MouseEvent e) {
    Graphics g = getGraphics();
    int x = e.getX(),  y = e.getY();
    g.drawLine(xLast, yLast, x, y);
    xLast = x;
    yLast = y;
  }
});
```

Для сохранения координат конца предыдущего отрезка между двумя событиями перемещения, введем два целых поля: xLast и yLast.

⇓

```
int xLast, yLast;
```

Чтобы определить начало самого первого отрезка серии, обработаем еще один вид события MouseEvent - нажатие на левую кнопку мыши.

Нажатие на левую кнопку мыши обрабатывает слушатель MouseListener при помощи метода mousePressed().

⇓

```
// Координаты курсора - начало первого отрезка

addMouseListener(new MouseAdapter() {
  public void mousePressed(MouseEvent e) {
    xLast = e.getX();
    yLast = e.getY();
  }
});
```

Теперь посмотрим на программу в целом.

⇓

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

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

public class MyFrame extends Frame {
  int xLast, yLast;

  /** Конструктор формы */

  MyFrame () {
      // Создаем и регистрируем слушателя
      addMouseListener(new MouseAdapter() {
        public void mousePressed(MouseEvent e) {
            xLast = e.getX();
            yLast = e.getY();
        }
});

// Создаем и регистрируем слушателя
addMouseMotionListener(new MouseMotionAdapter() {
    public void mouseDragged(MouseEvent e) {
        Graphics g = getGraphics();
        int x = e.getX(),  y = e.getY();
        g.drawLine(xLast, yLast, x, y);
        xLast = x;
        yLast = y;
      }
    });
}

/** Точка входа в программу */

public static void main(String[] args) {
    MyFrame frame = new MyFrame();
    frame.setSize(new Dimension(400, 400));
    frame.setVisible(true);
  }
}
```

2.7. Задание

1. Создать приложение, которое может ввести 2 числа и выполнить над ними одно из 4-х арифметических действий: сложение, вычитание,

умножение, деление. Числа размещаются в полях ввода, для каждого арифметического действия имеется отдельная кнопка.

2. Создать приложение, которое рисует на форме символы при нажатии на соответствующие клавиши. Символы сменяют друг друга так, что в каждый момент времени на форме изображен лишь один символ. Изображение не должно теряться при изменении размеров окна.
3. Создать форму с меткой (класс java.awt.Label), которую можно передвигать по форме при помощи мыши. **Совет:** обработайте событие "таскания" мышью для метки, а не для формы. Если поведение метки в начале движения вам не понравится, обработайте еще и событие нажатия кнопки мыши.
4. Создать форму, на которой можно было бы рисовать мышью методом "резиновой линии". В результате такого рисования получается ломаная линия. Когда мы перемещаем мышь с нажатой кнопкой, от последнего узла ломаной за курсором мыши тянется линия. При отпускании кнопки образуется новый отрезок линии и т.д. Начало ломаной установить в центре окна. Сохранять рисунок при повреждении окна не обязательно. **Совет:** обработайте события "таскания" мышью и отпускание кнопки мыши. Стирание линии выполняйте при помощи рисования в режиме XOR.

3. БИБЛИОТЕКА SWING

3.1. Что такое Swing

Пакет Swing является частью библиотеки JFC (Java Foundation Classes), которая предназначена для разработки пользовательского интерфейса. В версиях JDK 1.0 и 1.1 эту роль выполняла библиотека AWT.

Swing - это кодовое наименование проекта, направленного на разработку новых компонентов и вспомогательного API. Оно вошло в состав наименований пакетов, например, javax.swing и закрепилось за библиотекой. Swing полностью написана на Java. В отличие от компонентов AWT имена компонентов Swing начинаются с буквы “J”.

3.2. Приложение HelloSwing

Рассмотрим приложение, которое создает окно и метку на нем с надписью “Hello World!”

⇓

```
import javax.swing.*;
```

```
public class HelloWorldSwing {

  public static void main(String[] args) {
      JFrame frame = new JFrame("HelloSwing");
      JLabel label = new JLabel("Hello World!");
      frame.getContentPane().add(label);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.pack();
      frame.setVisible(true);
  }
}
```

Каждое Swing-приложение должно иметь, как минимум, один компонентный контейнер верхнего уровня. В примере таким контейнером служит компонент JFrame – окно, имеющее заголовок, рамку и кнопки на заколовке.

Эта часть кода создает главное окно и делает его видимым. Метод pack() приводит окно к предпочтительному размеру и размещает его компоненты.

⇓

```
JFrame frame = new JFrame("HelloSwing");
...
frame.pack();
frame.setVisible(true);
```

Приложение имеет еще один компонент – метку. Эта часть кода создает метку и помещает ее на форму.

⇓

```
JLabel label = new JLabel("Hello World");
frame.getContentPane().add(label);
```

Если мы хотим, чтобы при закрытии окна заканчивалась работа приложения, достаточно написать следующее.

⇓

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

Метод setDefaultCloseOperation() задает действие, которое будет выполняться при закрытии окна. Константа EXIT_ON_CLOSE означает,

что после закрытия окна приложение будет завершено командой System.exit(0). Кроме этой, определены следующие константы:

- HIDE_ON_CLOSE - окно делается невидимым;
- DISPOSE_ON_CLOSE - окно удаляется из памяти;
- DO_NOTHING_ON_CLOSE - никаких действий не производится, окно при этом закрыть не удастся.

Заметим, что действия, выполняемые при обработке события WindowEvent, возникающего при закрытии окна, имеют приоритет над действиями, заданными методом setDefaultCloseOperation().

3.3. Приложение SwingApplication

Задача. Создать окно с кнопкой и меткой, на которой отображается количество нажатий на кнопку.

При создании реальных приложений разместить весь программный код в одном классе невозможно. Во-первых, чтобы адаптировать форму для конкретной цели, нужно унаследовать класс JForm, поэтому каждая форма нуждается в отдельном классе. Во-вторых, в приложении может быть много форм, поэтому главную точку входа и другие вещи, относящиеся к программе в целом, целесообразно вынести в отдельный класс.

Класс MyApplication

Этот класс создается программистом и происходит непосредственно от Object.

В конструкторе MyApplication создается объект пользовательской формы и делается видимым. В методе main создается объект класса MyApplication. Сам метод main разместим в этом же классе.

⇓

```
public class MyApplication {

  public MyApplication() {
      MyFrame frame = new MyFrame();
      frame.setVisible(true);
  }

  public static void main(String[] args) {
      new MyApplication();
  }
}
```

Класс MyForm

Пользовательский класс формы наследуется от JForm. Все компоненты, расположенные на форме, являются полями класса формы.

В операторах инициализации класса создаются управляющие элементы. В конструкторе формы они настраиваются и помещаются на рабочую область. Для настройки и вставки компонентов можно отвести специальный метод, скажем init(), и вызывать его из конструктора.

Приведем содержимое файла MyFrame.java

⇓

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

public class MyFrame extends JFrame {
  private JPanel contentPane;
  private BorderLayout borderLayout1 = new BorderLayout();
  private JButton button = new JButton();
  private JLabel label = new JLabel();

  /** Конструктор формы */
  public MyFrame() {
    try {
      init();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }

  /** Инициализация компонентов */
  private void init() throws Exception  {
    contentPane = (JPanel) this.getContentPane();
    button.setText("push me");
    button.addActionListener(
      new java.awt.event.ActionListener() {
        public void actionPerformed(ActionEvent e) {
          int n = Integer.parseInt(label.getText());
          label.setText(String.valueOf(++n));
        }
      }
    );
    contentPane.setLayout(borderLayout1);
```

```
        this.setSize(new Dimension(400, 300));
        this.setTitle("Frame Title");
        label.setHorizontalAlignment(SwingConstants.CENTER);
        label.setText("0");
        contentPane.add(button, BorderLayout.SOUTH);
        contentPane.add(label, BorderLayout.CENTER);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

3.4. Обработка событий

Обработка событий в библиотеке Swing основана на модели источник – событие – слушатель и целиком заимствована у библиотеки AWT. К одному источнику можно присоединить много слушателей, а один слушатель может обслуживать несколько источников.

Обработка всех событий происходит в одном потоке выполнения, это гарантирует, что обработка следующего события не начнется, пока не закончится обработка предыдущего события.

Поскольку вывод графики происходит в том же потоке, что и обработка событий, никакой обработчик не должен выполнять длительных операций.

3.5. Порядок отрисовки

После каждого изменения состояния компонента он обязан себя перерисовать. Если перерисовка не происходит, это ошибка. В таком случае надо вызывать метод компонента **repaint()**.

Если компонент не хочет автоматически менять свои размеры, надо вызвать метод **revalidate()**, причем до вызова repaint().

Контейнер рисуется прежде, чем содержащиеся в нем компоненты. Если компоненты можно сделать непрозрачными **setOpaque(true)**, это немного уменьшит время рисования, т.к. фон под ними изображаться не будет. При отрисовке используется буферизация, т.е. изображение вначале создается в памяти, а затем быстро выводится на экран.

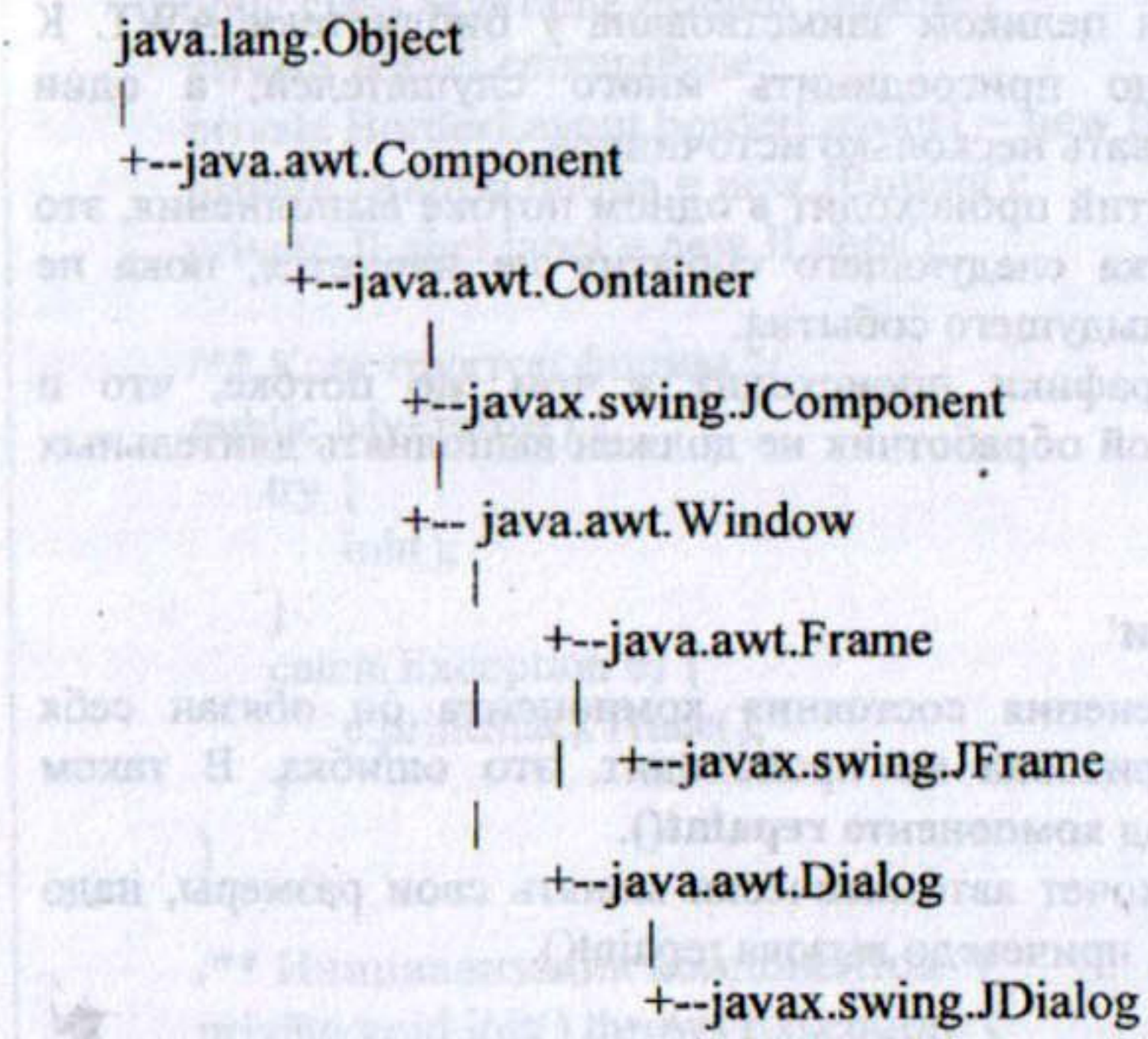
3.6. Задание

1. Создать окно с тремя кнопками. Каждая кнопка должна окрашивать поверхность окна в один из трех цветов: красный, зеленый, синий.
2. Довести число кнопок до семи, чтобы окрашивать окно во все цвета радуги. Обработчик нажатия на кнопку сделать общим для всех кнопок.

3. Сделать то же, используя не отдельные кнопки, а массив кнопок.
4. Создать приложение с тремя одинаковыми формами. Формы должны быть расположены каскадом. Воспользоваться массивом форм.
5. Создать приложение с четырьмя одинаковыми формами. Формы должны быть расположены черепицей. Сделать так, чтобы при изменении размеров одной формы, все остальные также меняли свои размеры.

4. КОМПОНЕНТЫ И КОНТЕЙНЕРЫ

Связь между основными классами библиотек Swing и AWT показывает следующая диаграмма.



4.1. Иерархия контейнеров в Swing

Компоненты в приложении организованы при помощи иерархии контейнеров.

Контейнерами верхнего уровня являются:

- JFrame - полнофункциональное окно с рамкой и заголовком;
- JDialog - диалоговое окно;
- JWindow - окно без заголовка и рамки;
- JApplet - маленькое несамостоятельное приложение.

Контейнерами среднего уровня являются:

- JPanel - простейший контейнер компонент;
- JScrollPane - контейнер с возможностью прокрутки;
- JTabbedPane - группа панелей снабженных язычками.

Каждый контейнер верхнего уровня неявно содержит особый контейнер среднего уровня, называемый панелью содержания (content pane).

Атомарные компоненты, такие как JButton и JLabel, не могут вмещать других компонентов и применяются для взаимодействия с пользователем.

4.2. Класс Container

Все контейнеры библиотеки Swing расширяют класс **java.awt.Container.**

Важнейшие методы этого класса сведены в таблицу.

⇓

Метод	Описание метода
Component add (Component comp)	Добавляет компонент в контейнер
Component add (Component comp, int i)	Добавляет компонент в i-ю позицию контейнера
void remove (Component comp)	Удаляет один компонент comp из контейнера
void removeAll ( )	Удаляет все компоненты из контейнера
int getComponentCount ()	Возвращает количество компонентов, находящихся в контейнере
Component getComponent (int i)	Возвращает компонент, находящийся в i-й позиции контейнера
Component getComponentAt (int x, int y)	Находит компонент, содержащий точку (x, y), среди компонентов непосредственно содержащихся в контейнере
Component findComponentAt (int x, int y)	Находит компонент, содержащий точку (x, y), среди всего содержимого контейнера
void setLayout (LayoutManager mgr)	Устанавливает в контейнере менеджер расположения

Чтобы добавить компонент в контейнер, используют одну из форм метода контейнера add(). Первый аргумент метода – добавляемый объект. Добавочные аргументы метода могут потребоваться для менеджеров расположения.

Пример. Расположить на форме две панели и на одной из панелей поместить кнопку. При нажатии на кнопку она должна перемещаться с одной панели на другую.

Структура программы в целом не будет отличаться от приложения "HelloSwing", включающего класс приложения MyApplication и класс формы MyFrame. В классе MyFrame определим следующие поля.

⇓

```
// Ссылка на внутренний контейнер формы
private JPanel contentPane;

// Массив из двух панелей
JPanel[] panels = {new JPanel(), new JPanel()};

// Кнопка под названием "кнопка"
JButton button = new JButton("Button");

// Номер панели, владеющей кнопкой
int home;

// Табличный менеджер, чтоб разместить две панели
private GridLayout gridLayout1 = new GridLayout();
```

Метод init(), предназначенный для инициализации компонент, будет содержать следующий код.

⇓

```
// Сохраняем ссылку на панель содержания формы,
// чтоб не вызывать метод getContentPane() много раз
contentPane = (JPanel) getContentPane();

// Устанавливаем табличный менеджер расположения
contentPane.setLayout(gridLayout1);
gridLayout1.setColumns(2);

// Ставим панели на форму
contentPane.add(panels[0]);
contentPane.add(panels[1]);

// Ставим кнопку на 0-ю панель
home = 0;
panels[home].add(button);
```

```
// Обработчик нажатия на кнопку снимает ее
// с одной панели и ставит на другую
button.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(ActionEvent e) {
        panels[home].remove(button);
        panels[home].repaint();
        panels[1 - home].add(button);
        home = 1 - home;
    }
});

// Эти операции нам уже знакомы
setSize(new Dimension(400, 300));
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

4.3. Легковесные и тяжеловесные компоненты

В каждой операционной системе графический интерфейс создается при помощи объектов этой операционной системы; в MS Windows такие объекты называются окнами.

Java предлагает свои собственные объекты для построения GUI, что делает ее программы платформно-независимыми, но на стадии вывода графики на экран объекты Java все же должны взаимодействовать с родными объектами операционной системы. Взаимодействие достигается тем, что для некоторых компонентов библиотек AWT и Swing изначально создается парный графический объект операционной системы. Такие компоненты имеют повышенную потребность в ресурсах и называются тяжеловесными.

В библиотеке Swing тяжеловесными компонентами являются JDialog, JFrame, JWindow, JApplet.

Все прочие компоненты Swing, не имеют парных объектов операционной системы, потребляют меньше ресурсов и называются легковесными. Чтобы получить возможность отрисовки, они должны иметь, по крайней мере, один тяжеловесный компонент среди тех контейнеров, в которых они содержатся.

Чем меньше тяжеловесных компонентов задействовано в пользовательском интерфейсе, тем меньше памяти требуется программе.

4.4. Управление расположением компонентов

Контейнеры используют менеджер расположения, чтобы определить размер и местоположение вмещаемых компонентов.

Менеджер расположения – это класс, который реализует интерфейс **java.awt.LayoutManagar**. В пакете java.awt определены наиболее употребительные менеджеры расположения.

Как правило, менеджеры расположения устанавливают в два типа контейнеров: в панель содержания (content pane) и JPanel. По умолчанию в панели содержания установлен менеджер расположения BorderLayout, а в JPanel - менеджер расположения FlowLayout.

Менеджер расположения устанавливается в контейнер методом **setLayout()**.

⇓

```
// Создаем панель
JPanel pane = new JPanel();
// Устанавливаем в нее менеджер BorderLayout
pane.setLayout(new BorderLayout());
```

Если в метод setLayout передать null, то контейнер лишится менеджера расположения. Положение вставляемых компонентов будет определяться их собственными границами, поэтому границы компонентов должны быть установлены методом setBounds() или reshape().

⇓

```
// Удаляем из панели содержания менеджер расположения
getContentPane().setLayout(null);
// Устанавливаем форму кнопки
button.setBounds(new Rectangle(33, 26, 152, 61));
// Ставим кнопку на форму
getContentPane().add(button);
```

Большим недостатком абсолютного позиционирования является негибкость приложений по отношению к размерам шрифта, разрешению экрана, размерам окон.

4.5. Классы менеджеров расположения

BorderLayout – полярное расположение

По сторонам панели выделяются 4 области и 5-я в центре. Метод add() перегружен, и второй параметр указывает область при помощи констант:

- BorderLayout.NORTH – север;
- BorderLayout.SOUTH – юг;
- BorderLayout.EAST – восток;
- BorderLayout.WEST – запад;
- BorderLayout.CENTER - центр.

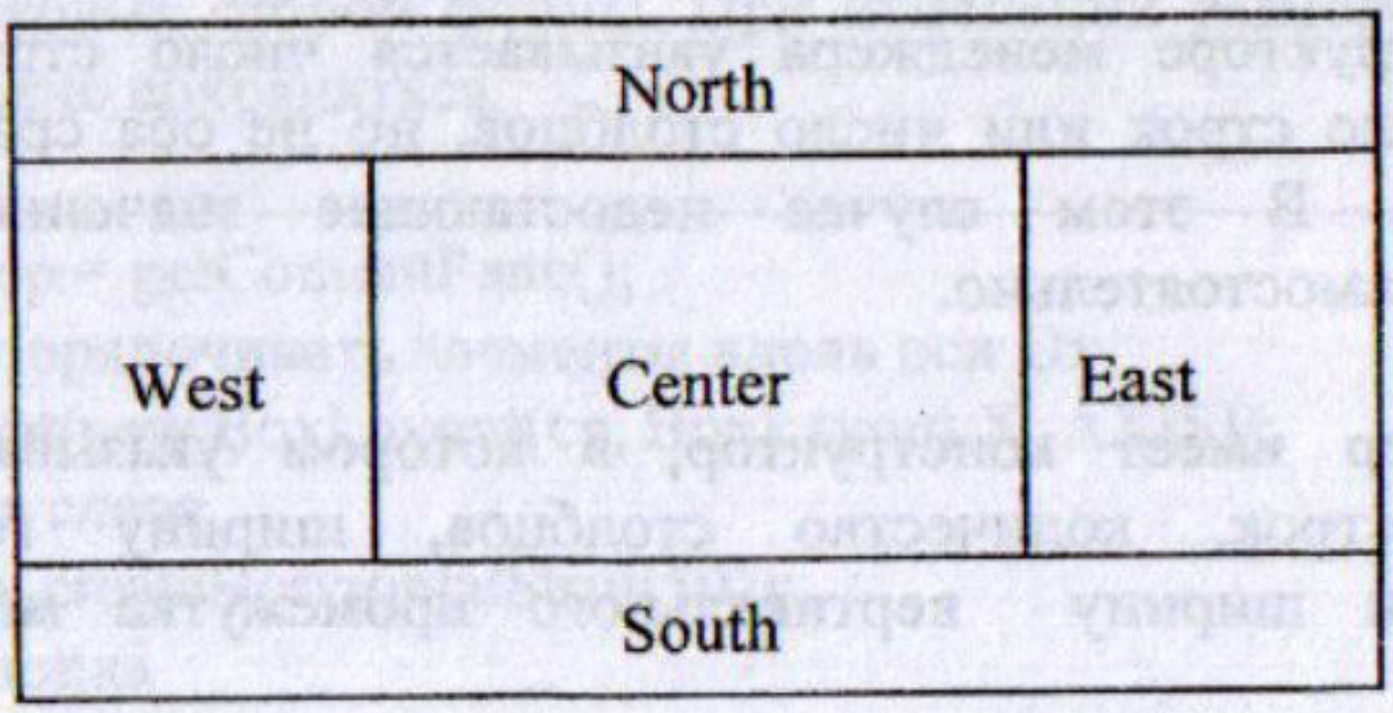
Метод add() с одним параметром добавляет компонент в центральную область.

Пример. Расположить 5 кнопок в контейнере, по одной в каждой области.

⇓

```
Container cp = getContentPane();
cp.setLayout(new BorderLayout());
cp.add(new JButton("North"), BorderLayout.NORTH) ;
cp.add(new JButton("South"), BorderLayout.SOUTH) ;
cp.add(new JButton("East" ), BorderLayout.EAST)  ;
cp.add(new JButton("West" ), BorderLayout.WEST)  ;
cp.add(new JButton("Center"),BorderLayout.CENTER);
```

Информация, предназначенная менеджерам расположения, передается в метод Container.add в дополнительных параметрах.



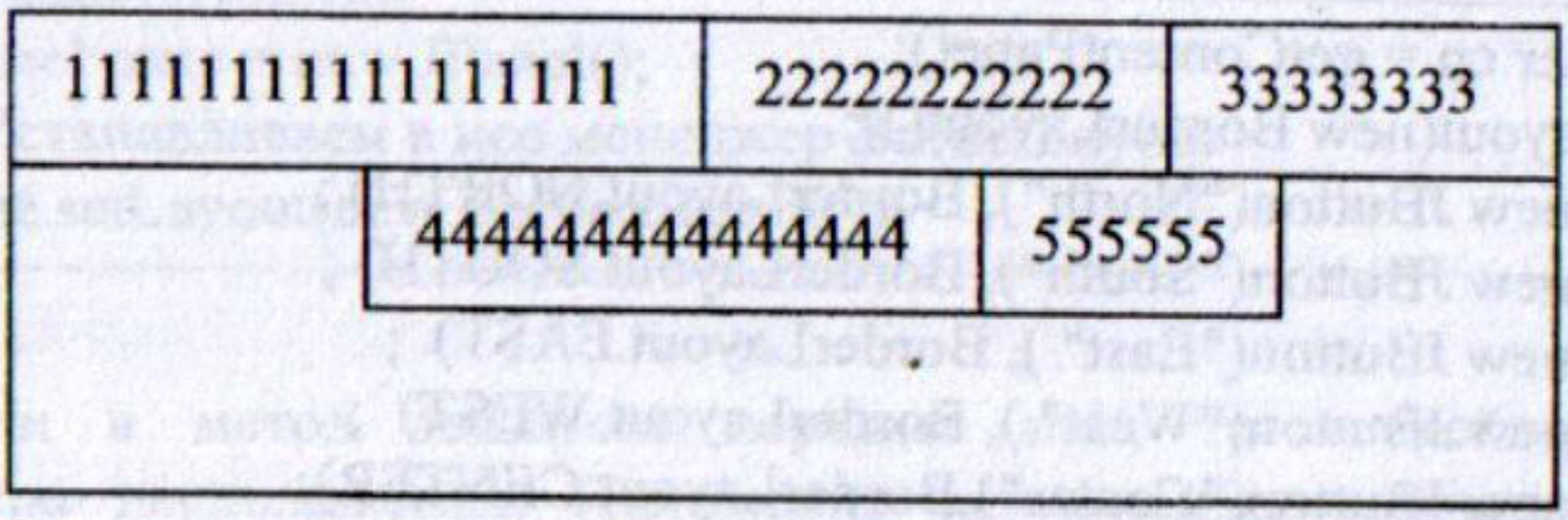
FlowLayout – последовательное расположение

Этот менеджер последовательно выкладывает компоненты в прямоугольник контейнера слева направо, пока не заполнится пространство одной строки, потом переходит на следующую строку и т.д.

Пример. Расположить кнопки в контейнере с менеджером FlowLayout.

⇓

```
Container cp = getContentPane();
cp.setLayout(new FlowLayout());
cp.add(new Button("1111111111111"));
cp.add(new Button("22222222"));
cp.add(new Button("333"));
cp.add(new Button("4444444444444444"));
cp.add(new Button("555555"));
```



GridLayout – табличное расположение

Менеджер GridLayout строит невидимую таблицу, затем заполняет ее компонентами построчно. Все компоненты будут одинакового размера.

В конструкторе менеджера указывается число строк и столбцов таблицы. Число строк или число столбцов, но не оба сразу, могут быть равны нулю. В этом случае недостающее значение вычисляется менеджером самостоятельно.

Менеджер имеет конструктор, в котором указывается 4 числа: количество строк, количество столбцов, ширину горизонтального промежутка и ширину вертикального промежутка между ячейками таблицы.

Пример. Расположить 6 кнопок в 2 ряда в контейнере с менеджером GridLayout. Промежутки между кнопками сделать по 10 пикселей.

⇓

```
Container cp = getContentPane();
```

```
cp.setLayout(new GridLayout(2, 3, 10, 10));
cp.add(new Button("1111111111111"));
cp.add(new Button("22222222"));
cp.add(new Button("333"));
cp.add(new Button("4444444444444444"));
cp.add(new Button("55555"));
```

11111111111	22222222	3333
44444444444	55555	

BoxLayout - блочное расположение

Менеджер BoxLayout позволяет располагать компоненты вертикально и горизонтально, а также позволяет управлять пространством между компонентами при помощи распорок и заполнителей. Распорка - жесткая невидимая палочка, заполнитель - невидимая пружина, способная к бесконечному расширению. Компоненты укладываются вдоль одной оси - X или Y. Для создания сложных конфигураций компонентов используют вложенные контейнеры с блочным расположением.

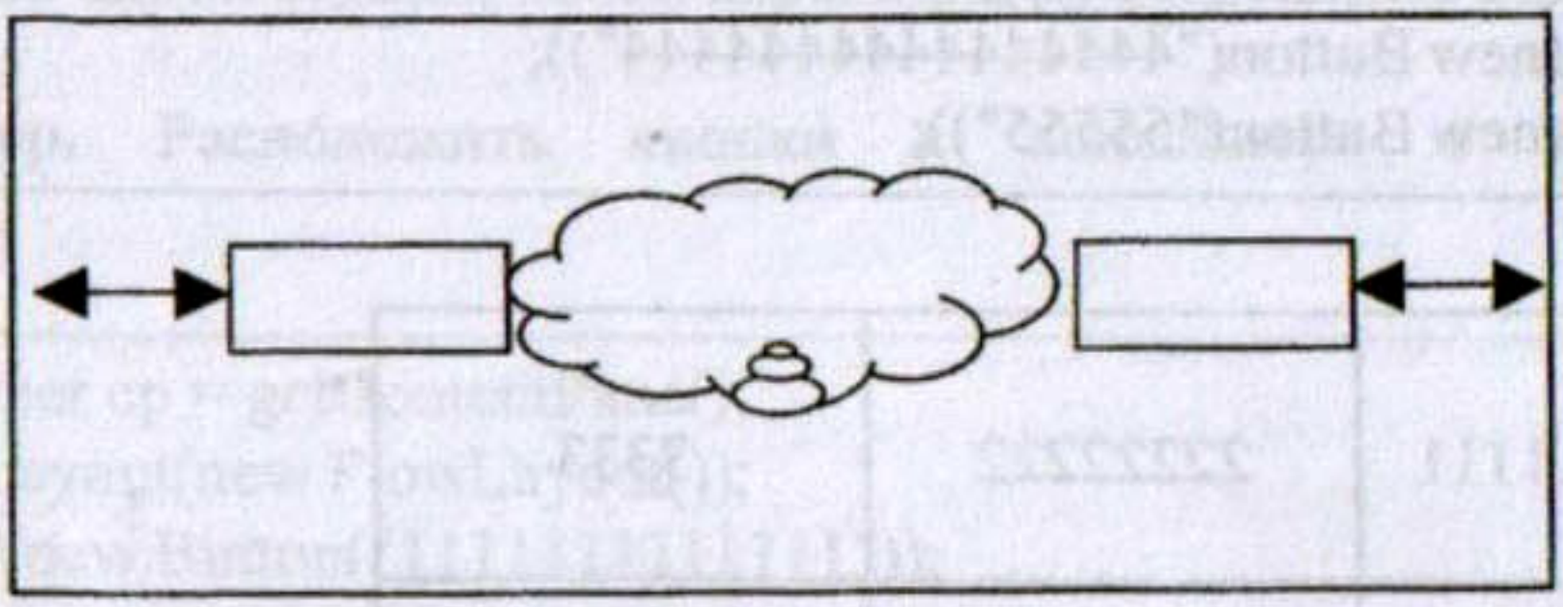
В конструкторе менеджера два параметра: контейнер, для которого будет работать менеджер, и направление оси расположения (Ox или Oy).

Пример. Расположить 2 кнопки на фиксированном расстоянии в 50 пикселей от боковых сторон формы. При изменении размеров формы это расстояние должно сохраняться.

⇓

```
Container cp = getContentPane();
// Будем упорядочивать элементы вдоль оси Ox
cp.setLayout(new BoxLayout(cp, BoxLayout.X_AXIS));
// Распорка слева
cp.add(Box.createHorizontalStrut(50));
// Левая кнопка
cp.add(new JButton("Left"));
// Бесконечно расширяющийся заполнитель
cp.add(Box.createGlue());
// Правая кнопка
cp.add(new JButton("Right"));
```

```
// Распорка справа
cp.add(Box.createHorizontalStrut(50));
```



GridBagLayout – RAD расположение

Этот менеджер используется в системах быстрого построения пользовательского интерфейса (Rapid Application Design - RAD). Он напоминает менеджер GridLayout, но компоненты могут занимать несколько смежных ячеек таблицы. В ручном размещении используется редко из-за сложных ограничений, используемых контейнером.

4.6. Реализация менеджера расположения

Если вам понадобится создать свой собственный менеджер расположения, то вы должны определить класс, который реализует интерфейс java.awt.LayoutManager. Этот интерфейс имеет пять методов, которые приведены в таблице.

Метод	Описание метода
void layoutContainer (Container parent)	Устанавливает размеры и координаты всех компонентов в контейнере parent
Dimension preferredLayoutSize (Container parent)	Определяет предпочтительный размер контейнера parent
Dimension minimumLayoutSize (Container parent)	Определяет минимально допустимый размер контейнера parent
void addLayoutComponent (String name, Component comp)	Добавляет объект ограничения с именем name
void removeLayoutComponent (Component comp)	Удаляет объект ограничения

Метод layoutContainer() является основным. Он вызывается всякий раз, когда изменяется размер контейнера.

Методы addLayoutComponent() и removeLayoutComponent() применяются, когда менеджер использует ограничения при расчете расположения компонентов. Ограничения могут быть простыми, как NORTH для BorderLayout, или сложными, как ограничения менеджера GridBagLayout.

Пример "Менеджер расположения". Разработаем простой менеджер, который располагает все компоненты строго по центру контейнера.

Класс менеджера назовем MyLayout.
⇓

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

public class MyLayout implements LayoutManager {

  /** Определение предпочтительных размеров контейнера:
   *  Из всех размеров компонентов выбираем наибольший
   *  горизонтальный и наибольший вертикальный размеры.
   */

public Dimension preferredLayoutSize(Container parent) {
    Component[] components = parent.getComponents();
    Dimension contSize = parent.getSize();
    int prefferedWidth = 0,  prefferedHeight = 0;
    for (int i = 0; i < components.length; i++) {
        Dimension d = components[i].getPreferredSize();
        if (prefferedWidth < d.getWidth())
            prefferedWidth = (int)d.getWidth();
        if (prefferedHeight < d.getHeight() )
            prefferedHeight = (int)d.getHeight();
    }
    return new Dimension(prefferedWidth, prefferedHeight);
}

/** Расчет координат компонентов */

public void layoutContainer(Container parent) {
    Component[] components = parent.getComponents();
```

```
        Dimension contSize = parent.getSize();
        for (int i = 0; i < components.length; i++) {
            Component c = components[i];
            int x = (int)(contSize.getWidth() - c.getWidth()) / 2;
            int y = (int)(contSize.getHeight() - c.getHeight()) / 2;
            c.setBounds(x, y, c.getWidth(), c.getHeight() );
        }
    }

    /** Определение минимальных размеров контейнера */

    public Dimension minimumLayoutSize(Container parent) {
        return new Dimension(50, 50);
    }

    /** Остальные методы не делают ничего. */

    public void addLayoutComponent(String name, Component comp) { }
    public void removeLayoutComponent(Component comp) { }
}
```

Для проверки создадим объект MyLayout и установим его в панель содержания формы.

⇓

```
getContentPane().setLayout(new MyLayout());
```

Теперь можно добавлять компоненты.

4.7. Что еще влияет на расположение компонентов

Рекомендации менеджерам расположения

Иногда компоненты могут нести в себе информацию рекомендательного характера, предназначенную для менеджеров расположения. Так вы можете установить размеры компонентов методами setMinimumSize, setPreferredSize и setMaximumSize или создать подкласс компонента, в котором перекроете соответствующие get-методы.

Информацию о выравнивании можно передавать аналогично при помощи методов setAlignmentX() и setAlignmentY().

Впрочем, на рекомендуемые размеры и выравнивание обращают внимание далеко не все менеджеры расположения.

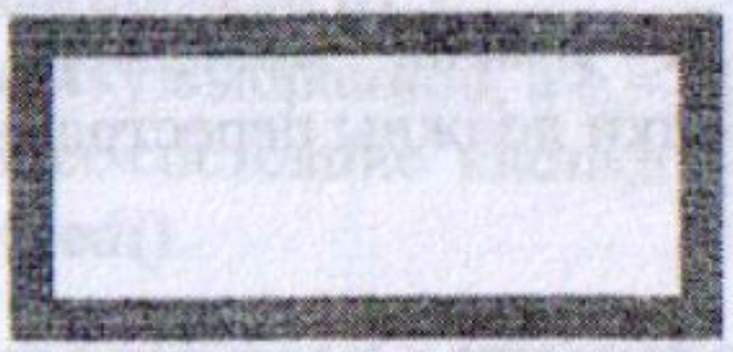
Промежутки между компонентами

Некоторые менеджеры автоматически делают промежутки между компонентами, например GridLayout. Для других менеджеров, например, BoxLayout, используют невидимые компоненты.

Всегда можно использовать пустые границы компонентов.

Границы компонентов

Границы компонента с геометрической точки зрения представляют собой рамку, очерченную внутри прямоугольника компонента.



Эта рамка может иметь различную ширину сторон, определенную окраску, создавать эффект приподнятости или опущенности, может быть очерчена линией по внешнему краю. Если компонент является контейнером, то все его содержимое будет располагаться в пределах внутреннего прямоугольника, создаваемого границами компонента.

С программной точки зрения, границы компонента - это объект, обладающий интерфейсом javax.swing.border.Border и зарегистрированный в компоненте методом компонента setBorder(). Характер границ целиком определяется свойствами этого объекта.

Пример. Обеспечить зазор между управляющими элементами и краями формы.

Для этого снабдим панель содержания границами шириной в 20 пикселей каждая.

⇓

```
Container cp = getContentPane();
cp.setBorder(BorderFactory.createEmptyBorder(20, 20, 20, 20);
```

4.8. Задание

1. Разработать калькулятор с одним окном, выполняющий четыре арифметических действия и работающий с двоичными числами. Калькулятор должен допускать изменение своих размеров в широких пределах.
2. Создать окно, разделенное на 4 области двумя линиями - вертикальной и горизонтальной. В каждой области разместить по цифре: 1, 2, 3, 4. Сделать так, чтобы линии можно было перемещать мышью, а

изображения цифр масштабировались бы в соответствии с размером областей.

3. Сделать форму, на которой можно динамически создавать новые кнопки, просто щелкая по ней мышью. Нажатие на созданную таким образом кнопку должно ее уничтожать.
4. Разработать менеджер расположения, который размещает все кнопки JButton вдоль диагонали контейнера, а положение прочих компонентов оставляет неизменным.
5. Создать форму с пятью кнопками и менеджером расположения BorderLayout. Сделать так, чтобы при щелчке мышью по поверхности окна в нем происходила замена менеджера расположения на FloowLayout и обратно. Кнопки должны перестраиваться тоже.

5. НЕКОТОРЫЕ КОМПОНЕНТЫ БИБЛИОТЕКИ SWING

Все рассматриваемые ниже компоненты наследуют класс javax.swing.JComponent.

В целях экономии бумаги далее мы будем опускать префикс javax.swing в названиях классов из этого пакета.

5.1. Кнопки

Рассмотрим диаграмму наследования основных "кнопочных" классов.

```
JComponent
   |
   +-AbstractButton
         |
         +-JMenuItem
         |    |
         |    +-JButton
         |    |    |
         |    |    +-JToggleButton
         |    |         |
         |    |         +-JRadioButton
         |    |         |
         |    |         +-JCheckBox
```

Все кнопки, флажки, переключатели и даже пункты меню унаследованы от класса AbstractButton. В этом классе сосредоточена основная масса методов, определяющих поведение кнопки.

Класс JButton представляет кнопку, которую можно нажимать при помощи мыши или клавиатуры, и мы уже не раз встречали их в примерах. Типичное использование таких кнопок состоит в обработке события ActionEvent, которое наступает при нажатии на кнопку.

Класс JToggleButton - это кнопка с двумя состояниями: выбранным и невыбранным. От нее происходят радиокнопки (класс JRadioButton) и переключатели (класс JCheckBox).

Программно установить кнопку JToggleButton в некоторое состояние можно методом

void setSelected(boolean b),

b = true делает кнопку выбранной, а b = false - невыбранной.

Возвращает текущее состояние кнопки метод

boolean isSelected().

Пример "Двоичная раскраска". Известно, что цветовая модель RGB представляет каждый цвет в виде смеси красного, зеленого и синего лучей различной интенсивности. Поставим на форму три кнопки с двумя состояниями с тем, чтобы каждая кнопка управляла своим лучом: нажата - максимум интенсивности, отжата - минимум, т.е. 0.

В операторах инициализации формы MyForm создадим необходимые компоненты.

⇓

```
private JPanel cp;
private FlowLayout flowLayout1 = new FlowLayout();
private JToggleButton btnR = new JToggleButton("R");
private JToggleButton btnG = new JToggleButton("G");
private JToggleButton btnB = new JToggleButton("B");
```

В конструкторе формы включим кнопки в панель содержания.

⇓

```
cp = (JPanel) this.getContentPane();
cp.setLayout(flowLayout1);
cp.add(btnR);
cp.add(btnG);
cp.add(btnB);
```

Там же создадим общий для всех слушатель события ActionEvent и зарегистрируем его во всех трех кнопках.

⇓

```
ActionListener listener = new java.awt.event.ActionListener() {
```

```
    public void actionPerformed(ActionEvent e) {
        int r = btnR.isSelected() ? 255 : 0;
        int g = btnG.isSelected() ? 255 : 0;
        int b = btnB.isSelected() ? 255 : 0;
        Color color = new Color(r, g, b);
        MyFrame.this.cp.setBackground(color) ;
    }
};

btnR.addActionListener(listener);
btnG.addActionListener(listener);
btnB.addActionListener(listener);
```

Радиокнопки часто объединяют в группы, только одна из кнопок группы может находиться в выбранном состоянии. В роли объединителя выступает объект класса ButtonGroup.

Пример "Группа радиокнопок". Создать группу радиокнопок. Номер нажатой радиокнопки показывать в метке, расположенной на этой же форме.

В операторах инициализации формы MyForm создадим необходимые компоненты.

⇓

```
private JPanel cp;
private JRadioButton button1 = new JRadioButton("1");
private JRadioButton button2 = new JRadioButton("2");
private JRadioButton button3 = new JRadioButton("3");
private ButtonGroup group = new ButtonGroup();
private JLabel label = new JLabel();
```

В конструкторе формы сгруппируем кнопки и вставим их в панель содержания.

⇓

```
// Собираем кнопки в группу
group.add(button1);
group.add(button2);
group.add(button3);

// Вставляем кнопки в панель содержания
cp = (JPanel) this.getContentPane();
cp.setLayout(flowLayout1);
```

```
cp.add(button1);
cp.add(button2);
cp.add(button3);

// Переводим первую кнопку в выбранное состояние
button1.setSelected(true);
```

Создадим один объект слушателя события ChangeEvent и зарегистрируем его во всех трех кнопках. Событие ChangeEvent происходит, когда кнопка изменяет свое состояние. Слушатель определяет, какая кнопка послужила источником события и делает сопутствующий ей текст текстом метки.

⇓

```
ChangeListener listener = new ChangeListener() {
  public void stateChanged(ChangeEvent e) {
     // Определить кнопку - источник события
     JRadioButton btn = (JRadioButton)e.getSource();
     // Если кнопка стала выбранной, выводим ее текст
     if (btn.isSelected())
         label.setText(btn.getText());
  }
};

// Регистрируем слушателя во всех кнопках
button1.addChangeListener(listener);
button2.addChangeListener(listener);
button3.addChangeListener(listener);
```

5.2. Иконки

Иконки - это небольшие изображения, которые могут находиться в метках, на кнопках и других компонентах.

С программной точки зрения иконка - это класс, реализующий интерфейс javax.swing.Icon. В этом интерфейсе есть следующие методы:

- void paintIcon(Component c, Graphics g, int x, int y) - рисует иконку на поверхности, предоставляемой объектом g, левый верхний угол иконки имеет координаты (x, y). Параметр c служит для получения полезной для рисования информации о компоненте, такой как цвет фона или цвет переднего плана компонента;
- int getIconWidth() - возвращает ширину иконки;

- int getIconHeight()- возвращает высоту иконки.

В том же пакете есть класс **ImageIcon**, который реализует интерфейс Icon на базе растрового изображения Image.

Чтобы получить иконку, которой затем можно пользоваться в программе, надо создать объект класса ImageIcon, например,

⇓

```
Icon sun = new ImageIcon("Sun.gif");
```

Необходимое для иконки изображение можно загружать как из локальных файлов, так и из сети.

Пример «Кнопка с изображениями». Снабдить кнопку изображениями, первое из которых будет видно в отжатом состоянии, второе - в нажатом состоянии, третье - при прохождении курсора мыши над кнопкой.

Подготовим три файла в формате gif и создадим три изображения:

⇓

```
Icon icon0 = new ImageIcon("icon0.gif") ;
Icon icon1 = new ImageIcon("icon1.gif");
Icon icon2 = new ImageIcon("icon2.gif");
```

Способность кнопок нести на себе изображения заложена на уровне класса AbstractButton и реализуется методами этого класса: **setIcon(), setPressedIcon(), setRolloverIcon()**. Воспользуемся ими, чтобы решить задачу.

⇓

```
jButton1.setIcon(icon0);
jButton1.setPressedIcon(icon1);
jButton1.setRolloverIcon(icon2;
```

5.3. Меню

Для построения меню используются классы JMenuBar, JPopupMenu, JMenuItem, JMenu, JCheckBoxMenuItem, JRadioButtonMenuItem. Отношения между ними напоминают те, что существуют в библиотеке AWT.

```
javax.swing.JComponent
```

```
|
-- JMenuBar - полоса в верхней части окна
|
-- JPopupMenu - всплывающее меню
|
-- AbstractButton
    |
    -- JMenuItem - универсальный пункт меню
    |
    -- JMenu - подменю
    |
    -- JCheckBoxMenuItem - пункт-флажок
    |
    -- JRadioButtonMenuItem - пункт-радиокнопка
```

Выпадающее меню

Класс JMenuBar является контейнером для выпадающих меню.

Меню может быть вставлено в любой контейнер верхнего уровня методом **setMenuBar()**.

⇓

```
JMenuBar mb = new JMenuBar();
...
this.setJMenuBar(mb);
```

Класс JMenu можно охарактеризовать как подменю. Подменю составлено из отдельных пунктов (класс JMenuItem), других подменю и разделителей (класс JSeparator).

Меню создается и собирается в одно целое при помощи методов add() контейнерных компонентов меню.

Пример "Стандартное меню". Снабдить форму главным меню с подменю File и Edit.

В блоке инициализации формы создаем необходимые детали меню.

⇓

```
// Создаем контейнер для подменю
private JMenuBar menubar = new JMenuBar();
// Создаем два подменю (контейнеры для пунктов)
private JMenu mnuFile = new JMenu("File");
private JMenu mnuEdit = new JMenu("Edit");
// Создаем отдельные пункты меню
```

```
private JMenuItem mnuNew = new JMenuItem("New");
private JMenuItem mnuOpen = new JMenuItem("Open");
private JMenuItem mnuClose = new JMenuItem("Close");
private JMenuItem mnuExit = new JMenuItem("Exit");
private JMenuItem mnuCopy = new JMenuItem("Copy");
private JMenuItem mnuPaste = new JMenuItem("Paste");
```

В методе инициализации собираем меню из деталей.

⇓

```
// Ставим меню на форму
setJMenuBar(menubar);
// Добавляем подменю в меню
menubar.add(mnuFile);
menubar.add(mnuEdit);
// Добавляем пункты к подменю File
mnuFile.add(mnuNew);
mnuFile.add(mnuOpen);
mnuFile.add(mnuClose);
// Это не пункт, а разделитель
mnuFile.addSeparator();
mnuFile.add(mnuExit);
// Добавляем пункты к подменю Edit
mnuEdit.add(mnuCopy);
mnuEdit.add(mnuPaste);
```

С каждым пунктом меню JMenuItem и даже с JMenu можно связать выполнение действий точно так же, как мы связывали их с нажатием кнопки JButton, т.е. через слушателя события ActionEvent.

В нашем примере сделаем так, чтобы приложение завершалось при выборе пункта File / Exit.

⇓

```
mnuExit.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    System.exit(0);
  }
});
```

Всплывающее меню

Всплывающее контекстно-зависимое меню является обязательной частью пользовательского интерфейса.

Класс JPopupMenu играет для всплывающего меню ту же роль, что класс JMenuBar для выпадающего, т.е. контейнера пунктов. Прочие детали во всплывающем меню те же, что и в выпадающем: пункты JMenuItem, подменю JMenu, разделители JSeparator.

Для привязки меню к компоненту создается слушатель событий от мыши, который регистрируется в компоненте. На разных платформах принято вызывать локальное меню по-разному. Чтобы упростить создание кроссплатформенных приложений, в классе MouseEvent есть метод

```
boolean isPopupTrigger();
```

который проверяет, является ли данное событие на данной платформе сигналом для появления контекстно-зависимого меню.

Пример "Привязка всплывающего меню". Связать всплывающее меню непосредственно с формой.

Поскольку интерфейс MouseListener имеет много методов, при создании слушателя воспользуемся классом-адаптером MouseAdapter.

⇓

```
addMouseListener(new java.awt.event.MouseAdapter() {
  public void mousePressed(MouseEvent e) {
      if (e.isPopupTrigger()) {
          pm.show(e.getComponent(), e.getX(), e.getY());
      }
  }

  public void mouseReleased(MouseEvent e) {
      if (e.isPopupTrigger()) {
          pm.show(e.getComponent(), e.getX(), e.getY());
  }
  }
});
```

На платформе Windows вызов всплывающего меню происходит при отпускании правой кнопки мыши. Событие нажатия кнопки мыши обрабатывается на случай переноса программы на другие платформы.

Таким образом, локальное меню можно связать с любым компонентом, чувствительным к мыши.

Интерфейс Action

Согласно стандарту пользовательского интерфейса человек имеет право выполнять одно и то же действие по-разному: при помощи мыши, горячей клавиши, кнопки, пункта меню и т.п.

С целью создания универсального слушателя для всего этого разнообразия в javax.swing объявлен интерфейс Action, который расширяет интерфейс ActionListener. Помимо метода actionPerformed, он сосредотачивает в одном месте поясняющий текст, иконку, возможность активировать или деактивировать слушателя, мнемоническую клавишу.

В некоторые контейнеры, такие, как JMenu или инструментальная панель, объект, реализующий интерфейс Action, может быть добавлен наряду с обычными элементами, и контейнер сам создаст соответствующий элемент - пункт меню или кнопку.

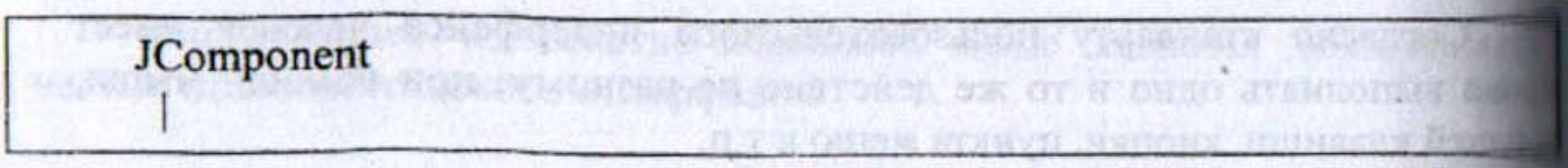
5.4. Задание

1. Расположить горизонтально несколько кнопок с двумя состояниями. Нажатое состояние кнопки символизирует 1, отжатое – 0. При каждом изменении состояния кнопок отображать образованное ими двоичное число в десятичном представлении в заголовке формы.
2. Разработать приложение, которое позволяло бы при помощи пунктов меню управлять положением иконки на форме: слева, справа, вверху или внизу формы. В меню должно быть 4 пункта с такими же названиеми. **Совет:** иконку поместите на метку JLabel.
3. Добавьте к предыдущему заданию возможность перемещать иконку при помощи всплывающего меню. Всплывающее меню привязать к метке, а не к форме. **Совет:** Если в предыдущей задаче вы не воспользовались интерфейсом Action, сделайте это сейчас.
4. Усовершенствовать предыдущее решение, сделав не активным неактуальный пункт меню. Например, если иконка находится на форме слева, в меню должны быть активны три пункта: "Вправо", "Вверх" и "Вниз".
5. Разработать приложение, которое бы строило свое меню по образу файлового дерева на диске a:.

6. ТЕКСТОВЫЕ ПОЛЯ И МОДЕЛЬ ДОКУМЕНТА

6.1. Текстовые поля

Ввод и редактирование текста обеспечивается в библиотеке Swing целой группой компонентов. Отношения наследования меду ними показаны на следующей диаграмме.



```
-- javax.swing.text.JTextComponent
|
-- JTextField – однострочное текстовое поле
|       |
|       -- JPasswordField – поле для ввода пароля
|
-- JTextArea – многострочное текстовое поле
|
-- JEditorPane - панель редактирования
   |
    -- JTextPane – текстовая панель
```

Класс JTextComponent

Этот класс является базовым для всех текстовых полей. Он поддерживает следующие возможности.

1. Генерирует события CaretEvent при перемещении каретки, которые получает CaretListener. Это позволяет отслеживать положение каретки и положение выделенного участка текста.
2. Имеет список выполняемых команд в виде массива объектов Action. Эти команды можно связать с событиями от клавиатуры, кнопок и т.п.
3. Для ввода текста взаимодействует с набором программных компонент, называемых input method framework, что позволяет вводить большое количество символов при помощи малого числа клавиш.
4. Поддерживает модель документа, что позволяет отделить содержание текста от его представления и ввода.
5. Осуществляет двустороннюю связь между позицией внутри документа и двухмерными координатами видимого отображения текста.
6. Поддерживает историю изменений, что позволяет реализовать Undo/Redo операции, если это нужно.
7. Имеет методы, которые являются потокобезопасными.

Класс JTextField

Класс JTextField представляет собой однострочное текстовое поле. Фактически это редактор текста, позволяющий добавлять и удалять символы, выделять непрерывный участок текста, использовать буфер обмена.

Поместить текст в поле можно методом

void setText(String text).

Чтобы получить весь текст или его часть, применяют методы

String getText()

String getText(int offs, int len),

где offs - смещение, а len - длина участка текста.

Ширина поля устанавливается менеджером расположения, сообразуясь с размером содержащего в нем текста, но если воспользоваться методом

void setColumns(int col);

то можно задать постоянный размер поля. Параметр col означает количество символов моноширинного шрифта, если шрифт иной, то число видимых в поле символов будет больше.

Пример "Редактор заголовка". Поместить на форму поле JTextField и использовать его для редактирования заголовка формы.

Создадим текстовое поле в операторах инициализации формы.

⇓

```
private JTextField text = new JTextField();
```

В методе инициализации, вызываемом из конструктора формы, поместим следующий код.

⇓

```
// Добавляем текстовое поле в панель содержания.
getContentPane().add(text);
// Заголовок формы переносим в текстовое поле
text.setText(getTitle());
// Устанавливаем ширину поля в 20 символов
text.setColumns(20);
// В ответ на двойной щелчок мыши обновляем заголовок
text.addMouseListener(
  new java.awt.event.MouseAdapter() {
  public void mouseClicked(MouseEvent e) {
      if (e.getClickCount() > 1)
          setTitle(text.getText());
      }
  }
);
```

Чтобы определить был ли щелчок мыши двойным, мы воспользовались методом MouseEvent.getClickCount(), который показывает, сколько именно щелчков связывается с данным "мышиным" событием.

Пример "Позиция каретки". Показывать позицию каретки при перемещении ее по тексту.

Создадим слушателя событий каретки и зарегистрируем его в текстовом компоненте.

⇓

```
text.addCaretListener(new
  javax.swing.event.CaretListener() {
      public void caretUpdate(CaretEvent e) {
          // Выводим значение позиции в метку на форме
          label.setText(String.valueOf(e.getDot()));
      }
  }
);
```

6.2. Шаблон MVC и модель Document

Отделение собственно данных от формы их представления является фундаментальной идеей, которая проявляется в самых разных отраслях программирования. Смысл ее в том, что, если некоторый объект несет в себе информацию, то в нем есть отношения, существенные и несущественные для этой самой информации. Например, слова "Курение опасно для вашего здоровья" не меняют своего грозного смысла от того, где они написаны, каким шрифтом и каким цветом.

В библиотеке Swing эта идея реализуется в виде шаблона проектирования MVC - Model-View-Control (Модель-Представление-Управление), и в таком виде она даже более развита. Существенные для содержания отношения составляют модель, форма передачи содержания - это представление, а способы изменить содержание - это управление.

Все атомарные компоненты библиотеки Swing имеют модель, но реальную ценность она имеет только в сложных компонентах, таких как текстовые.

Модель компонента JTextComponent представлена интерфейсом Document и реализуется отдельным объектом. Это, кстати, дает возможность представления одного и того же содержания при помощи нескольких текстовых компонентов одновременно.

Когда текстовый компонент создается, объект стандартной модели автоматически создается тоже. При желании программист может создать текстовый компонент со своей собственной моделью, может заменить имеющуюся модель своей.

Объект модели текстового компонента доступен при помощи методов компонента

Document getDocument();

и

void setDocument(Document doc).

Интерфейс Document

Основные методы интерфейса Document показаны в таблице.

Метод	Содержание
int getLength()	Возвращает количество символов в тексте документа
String getText (int offset, int length)	Возвращает весь текст документа или его часть в виде строки
void insertString(int offset, String str, AttributeSet a)	Вставляет в документ строку str, в позиции offset
void remove(int offs, int len)	Удаляет из текста документа len символов, начиная с позиции offset
void addDocumentListener(DocumentListener listener)	Регистрирует слушателя изменений, происходящих в документе
void removeDocumentListener(DocumentListener listener)	Удаляет слушателя изменений, происходящих в документе

Пример "Все буквы прописные". Сделать так, чтобы все буквы, попадающие в текстовое поле, становились прописными.

Поскольку речь идет об изменении содержания, а не отображения, имеет смысл использовать модель.

Унаследуем класс javax.swing.text.PlainDocument, реализующий интерфейс Document, и перекроем его метод insertString(), который вызывается при добавлении текста в документ.

⇓

```
javax.swing.text.*;

class UpperCaseDocument extends PlainDocument {

public void insertString(int offs, String str, AttributeSet a)
throws BadLocationException
{
    if (str != null)
        super.insertString(offs, str.toUpperCase(), a);
}
```

```
}
```

Подправленную таким образом модель дадим текстовому полю вместо модели по умолчанию.

⇓

```
text.setDocument(new UpperCaseDocument());
```

Теперь текст будет состоять только из прописных букв, независимо от того, как эти буквы туда попадут: с клавиатуры, из буфера обмена, при помощи особых кнопок, как в калькуляторе и т.п.

Событие DocumentEvent

Интерфейс Document позволяет зарегистрировать слушателя типа DocumentListener, который будет получать извещение (событие DocumentEvent) о любом изменении документа, каким бы образом, это изменение не было сделано. Это избавляет программиста от необходимости отслеживать многочисленные события от управляющих элементов, в результате которых изменяется содержание документа.

Пример "Подсчет слов". Показывать текущее количество слов в тексте.

К модели текстового поля добавим слушателя события DocumentEvent. Этот слушатель имеет три метода, вызываемые в ответ на вставку, удаление и изменение текста в документе. В каждом из них будем подсчитывать количество слов в документе и показывать это количество в заголовке формы.

⇓

```
text.getDocument().addDocumentListener(
  new DocumentListener() {

     // Все три метода вызывают showWordNo()

     public void insertUpdate(DocumentEvent e)
     { showWordNo(e); };
     public void removeUpdate(DocumentEvent e)
     { showWordNo(e); };
     public void changedUpdate(DocumentEvent e)
     { showWordNo(e); };

     // Подсчет слов и вывод в заголовок формы
```

```
        void showWordNo(DocumentEvent e) {
            Document doc =  e.getDocument();
            try {
                String t = doc.getText(0, doc.getLength());
                // Подсчет числа слов
                int n = (new StringTokenizer(t)).countTokens();
                // Вывод числа в заголовок формы
                setTitle(String.valueOf(n));
            } catch (Exception ex) {};
        }
    }
);
```

6.3. Другие текстовые компоненты

TextArea - текстовая область

TextArea - это поле ввода, в котором располагается не одна, как в TextField, а много строк текста. Многострочностью объясняется присутствие в классе дополнительных методов, некоторые из которых приведены в таблице.

Метод	Описание
void setLineWrap(boolean wrap)	Задает режим переноса строк. Если wrap = true, длинные строки сворачиваются
void setWrapStyleWord(boolean word)	Задает режим переноса слов. Если word = true, разрывы при свертке строк происходят по границам слов
void setRows(int rows)	Устанавливает высоту текстовой области в rows строк
void setColumns(int columns)	Устанавливает ширину текстовой области в columns символов

Пример "Текстовая область". Разработать класс, который наследует панель и несет на себе компонент JTextArea, вставленный в JScrollPane для обеспечения прокрутки.

⇓

```
class MyTextArea extends JPanel {

  // В конструкторе выполняются все настройки
```

```
public MyTextArea() {
    String str = "TextArea это поле ввода, \n" +
    "в котором располагается не одна, \n" +
    "как в TextField, а много строк текста.";

    JTextArea textArea = new JTextArea(str, 4, 10);
    textArea.setLineWrap(false);
    textArea.setWrapStyleWord(true);
    textArea.setFont(new Font("Serif", Font.PLAIN, 18));
    JScrollPane scrollPane = new JScrollPane(textArea);
    add(scrollPane);
}

// Метод main для испытания разработанного класса

public static void main(String [] args) {
    JFrame frame = new JFrame("My Text Field");
    frame.getContentPane().add(new MyTextArea());
    frame.pack();
    frame.setVisible(true);
}
}
```

JEditorPane - панель редактирования

Этот компонент обязан своим названием тому, что с ним связан некий редакторский инструментарий (в виде класса - наследника javax.swing.text.EditorKit), причем этот инструментарий сменный, что позволяет компоненту работать с различными форматами текста. По умолчанию панель редактирования способна отображать текст в формате HTML, в формате Microsoft RTF и простой текст.

Замечание. Инструменты редактирования связаны и с другими текстовыми компонентами, но здесь они доступны программисту.

Пример "Панель редактирования". Разработать класс, который наследует панель и несет на себе компонент JEditorPane, вставленный в JScrollPane для обеспечения прокрутки.

Пример аналогичен предыдущему, поэтому дадим его в сокращенном виде.

⇓

```
class MyEditorPane extends JPanel {
```

```
public MyEditorPane() {
    JEditorPane editorPane = new JEditorPane();
    JScrollPane scrollPane = new JScrollPane(editorPane);
    scrollPane.setPreferredSize(new Dimension(800, 600));
    // Запрет на изменение содержимого панели пользователем
    editorPane.setEditable(false);
    try {
        URL url = new URL("file:/C:/111.html");
        editorPane.setPage(url);
    } catch (Exception e) { }
    add(scrollPane);
  }
}
```

URL - это класс, который позволяет единым образом идентифицировать как сетевые, так и локальные ресурсы.

JTextPane - текстовая панель

Текстовая панель является подклассом панели редактирования, в котором моделью документа служит стилизованный текст. В такой модели любому участку текста (от параграфа до отдельного символа) сопоставлен стиль.

Стиль - это объект, являющийся коллекцией атрибутов, таких как цвет_символов=синий, вид=наклонный, цвет_фона=голубой и т.п.

Код, который призван наполнить текстовую панель содержанием, может иметь следующую структуру.

1. Создать различные объекты стиля.
2. Добавить эти объекты в объект текстовой панели.
3. Вставить в документ участки текста, указывая стиль для каждого из них.

Пример "Текстовая панель". Разработать класс, который наследует панель и несет на себе компонент JTextPane.

⇓

```
class MyTextPane extends JPanel {

  // Типовой каркас обеспечивает конструктор

  public MyTextPane() {
```

```
        JTextPane textPane = new JTextPane();
        // Этот метод создает стили и добавляет их в панель
        initializeStyles(textPane);
        // Этот метод вставляет в документ стилизованный текст
        setPaneText(textPane);

        JScrollPane scrollPane = new JScrollPane(textPane);
        scrollPane.setPreferredSize(new Dimension(450, 225));
        add(scrollPane);
    }

    // Здесь реализация методов initializeStyles() и setPaneText()
    ...
}
```

Отдельно рассмотрим метод initializeStyles().

⇓

```
public void initializeStyles(JTextPane pane) {

    // context - контейнер стилей, имеющихся в панели
    StyleContext context = StyleContext.getDefaultStyleContext();
    // defaultStyle - базовый стиль текста
    Style defaultStyle = context.getStyle(StyleContext.DEFAULT_STYLE);

    // Создаем новый стиль и помещаем его в контейнер
    Style newStyle = pane.addStyle("normal", defaultStyle);
    // Изменяем атрибуты нового стиля: семейство и размер шрифта
    StyleConstants.setFontFamily(newStyle, "SansSerif");
    StyleConstants.setFontSize(newStyle, 16);

    // Создаем новый стиль и помещаем его в контейнер
    newStyle = pane.addStyle("italic-14", defaultStyle);
    StyleConstants.setItalic(newStyle, true);

    // Создаем новый стиль и помещаем его в контейнер
    newStyle = pane.addStyle("italic-16", defaultStyle);
    StyleConstants.setItalic(newStyle, true);
    StyleConstants.setFontFamily(newStyle, "Serif");
    StyleConstants.setFontSize(newStyle, 16);
    ...
```

```
}
```

Отдельно рассмотрим метод setPaneText().

⇓

```
public void setPaneText(JTextPane pane) {
  Document document = pane.getDocument();
  try {
      document.insertString(document.getLength(), "In addition to the ",
          pane.getStyle("normal"));
      document.insertString(document.getLength(), "In addition to the ",
          pane.getStyle("italic-14"));
      document.insertString(document.getLength(), "In addition to the ",
          pane.getStyle("italic-16"));
      ...
  }
  catch (BadLocationException e) {  }
}
```

6.4. Задание

1. Создать форму и текстовое поле на ней. В заголовке формы должно непрерывно отображаться текущее число символов, находящихся в текстовом поле. Пробелы тоже считаются символами.
2. Разработать текстовое поле с функцией автоматической замены русского текста, ошибочно набранного английскими буквами, на правильный русский текст. Для выполнения замены пользователь должен установить каретку в любом месте ошибочно набранного участка и нажать на кнопку "Замена". **Совет:** создайте две постоянных согласованных строки: одна из латинских, а другая из заменяющих их русских букв.
3. Создать текстовое поле для ввода пароля. Существенными в пароле должны признаваться только согласные буквы, например, пароли "table" и "to blue" должны считаться идентичными. Гласными считаются буквы "e","u","o","a","y".
4. Создать текстовую панель, в которую можно было бы вводить двуцветный текст: черными и красными буквами. Переключение цветов выполнять при помощи кнопки с двумя состояниями.
5. Создать текстовую панель, в которой можно было бы изменять высоту символов выделенного фрагмента текста.

7. ФОРМЫ И ДИАЛОГИ

Приложения с графическим интерфейсом обычно используют контейнер JFrame в качестве главного окна. Если необходимо иметь окно, зависящее от другого окна, сворачивающееся и разворачивающееся вместе с ним, используют контейнер JDialog.

7.1. Формы

Хотя мы давно применяем формы в своих программах, нам необходимо знать о них больше.

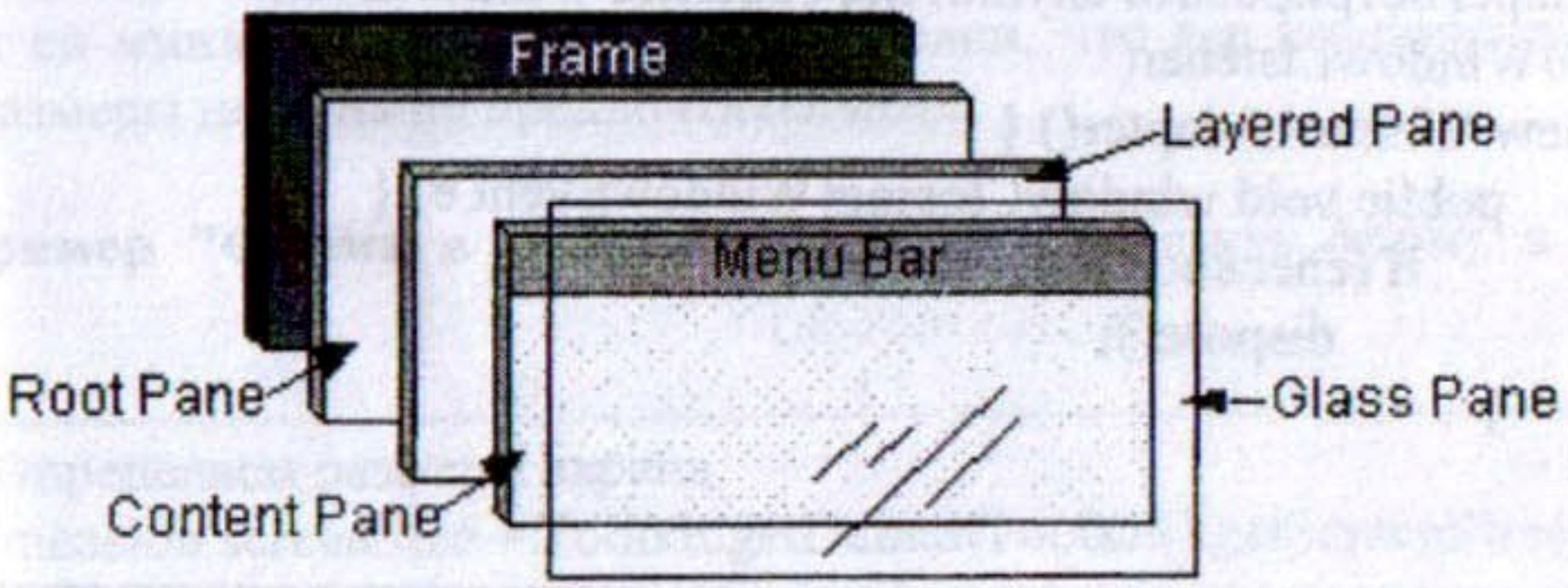
Корневая панель

Разработчики awt-приложений всегда помещали на форму панель, а уже на нее остальные компоненты. В библиотеке Swing такой необходимости нет, т.к. все контейнеры верхнего уровня изначально содержат в себе корневую панель (объект JRootPane). Эта панель имеет четыре части: панель расположения (поле layeredPane), панель содержания (поле contentPane), полоса меню (поле menuBar) (факультативно) и стеклянная панель (поле glassPane).

Панель содержания и полоса меню нам уже знакомы.

Панель расположения отвечает за расположение полосы меню и панели содержания. В нее можно помещать и другие компоненты, причем кроме двух координат на плоскости они будут иметь третью, z-координату или глубину. Это позволяет размещать компоненты с перекрытием.

Стеклянная панель располагается поверх панели содержания и обычно невидима. Она используется, когда надо перехватить события мыши до того, как они пройдут обычную обработку, или что-нибудь временно нарисовать поверх постоянных компонентов.



Оконные события

Обработку "оконных" событий выполняет слушатель WindowListener. Он реагирует на:

- открытие окна - метод windowOpened();
- попытку закрытия окна из системного меню - метод windowClosing();
- свершившийся факт закрытия окна - метод windowClosed();
- активацию окна - метод windowActivated();
- выход окна из активного состояния - метод windowDeactivated();
- сворачивание окна в иконку - метод windowIconified();
- разворачивание окна из иконки - метод windowDeiconified().

Заметим, что попытка закрытия окна будет успешной, только если в обработчике события windowClosing() сделать окно невидимым методом setVisible(false) или удалить методом dispose().

Пример "Разрешение закрыть". Создать форму с переключателем JCheckBox. Сделать так, чтобы форму можно было закрыть из системного меню, только если закрытие разрешено, т.е. переключатель на форме выбран.

⇓

```
// Создать переключатель как поле формы
private JCheckBox checkbox = new JCheckBox("CLOSE");
```

В конструкторе формы

⇓

```
// Установить переключатель на форму
getContentPane().add(checkbox, BorderLayout.WEST);

// Зарегистрировать слушателя события WindowEvent
addWindowListener(
  new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
          if (checkbox.isSelected())
              dispose();
      }
  }
);
```

Для контейнеров верхнего уровня есть возможность задать операцию, которая будет выполняться при закрытии окна из системного меню. В частности, для JFrame возможны следующие операции:

1) сделать окно невидимым;
2) уничтожить окно;
3) закрыть приложение;
4) не делать ничего.

Нужный вариант выбирается методом

void setDefaultCloseOperation(int operaton).

Метод получает в качестве аргумента одну из констант, названия которых говорят сами за себя:

1) HIDE_ON_CLOSE;
2) DISPOSE_ON_CLOSE;
3) EXIT_ON_CLOSE;
4) DO_NOTHING_ON_CLOSE.

Приведенный выше пример будет работать, как задумано, только если выбрать последний вариант, поскольку по умолчанию выбирается первый.

Иными словами вместо создания и регистрации слушателя WindowsListener к программе следует добавить одну строку.

⇓

```
setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
```

Если в программе и установлена операция при закрытии, и задан обработчик события закрытия, то приоритет будет у обработчика.

7.2. Размеры формы и ее положение на экране

Размеры формы устанавливаются методом setSize(), положение на экране - методом setLocate(), то и другое сразу - методом setBounds().

Чтобы сделать форму компактной, вызывают метод pack(), который придаст ей минимальный размер при условии, что все компоненты будут иметь размеры не меньше предпочтительных.

Пример "Форма в центре экрана". Поместить форму в центре экрана.

⇓

```
// Определяем размеры экрана
Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();

// Определяем размеры формы
Dimension frameSize = frame.getSize();
```

```
// Если надо, уменьшаем размеры формы
if (frameSize.height > screenSize.height) {
  frameSize.height = screenSize.height;
}
if (frameSize.width > screenSize.width) {
  frameSize.width = screenSize.width;
}

// Устанавливаем местоположение формы
frame.setLocation(
  (screenSize.width  -  frameSize.width) / 2,
  (screenSize.height - frameSize.height) / 2 );
```

Объект Toolkit является связующим звеном между классами AWT и объектами конкретной операционной системы. Большинство методов этого класса не вызываются напрямую, но есть исключения, например, getScreenSize(). Ссылку на действующий объект Toolkit возвращает статический метод getDefaultToolkit().

Класс Dimension инкапсулирует в одном объекте два значения: длину и ширину.

7.3. Диалоги

Диалоговые окна являются объектами класса JDialog.

Каждое диалоговое окно имеет владельца, им может быть форма или другой диалог. Если владелец сворачивается в иконку, диалог сворачивается тоже, если владелец закрывается, диалог закрывается тоже.

Диалог может быть создан модальным или немодальным. Пока модальный диалог остается видимым, он не дает остальным окнам приложения получать события от мыши или клавиатуры, оставаясь, таким образом, в центре внимания пользователя. В остальном диалог похож на форму.

Пример "Координаты формы". Создать модальное диалоговое окно с двумя текстовыми полями: "X-координата" и "Y-координата" и двумя кнопками "Ok" и "Cancel". Диалог использовать для отображения и изменения положения формы-владельца на экране.

По аналогии с формой, управляющие элементы диалога делаем полями класса MyDialog.

⇓

```
private JTextField txtX = new JTextField(6);
private JTextField txtY = new JTextField(6);
private JButton btnOk = new JButton("Ok");
private JButton btnCancel = new JButton("Cancel");
```

Настройку управляющих элементов производим в конструкторе или специальном методе, который из него вызывается.

⇓

```
// Размещаем элементы в окне диалога
getContentPane().add(txtX);
getContentPane().add(txtY);
getContentPane().add(btnOk);
getContentPane().add(btnCancel);

// Определяем владельца и его координаты
Point p = getOwner().getLocation();

// Заносим координаты владельца в текстовые поля
txtX.setText(String.valueOf(p.getX()));
txtY.setText(String.valueOf(p.getY()));
```

В ответ на нажатие кнопки btnOk изменяем положение владельца и делаем диалог невидимым.

⇓

```
int x = (int)Float.parseFloat(txtX.getText());
int y = (int)Float.parseFloat(txtY.getText());
getOwner().setLocation(x, y);

setVisible(false);
```

В ответ на нажатие кнопки btnCancel просто делаем диалог невидимым.

⇓

```
setVisible(false);
```

Чтобы открыть окно диалога, достаточно создать экземпляр класса MyDialog.

⇓

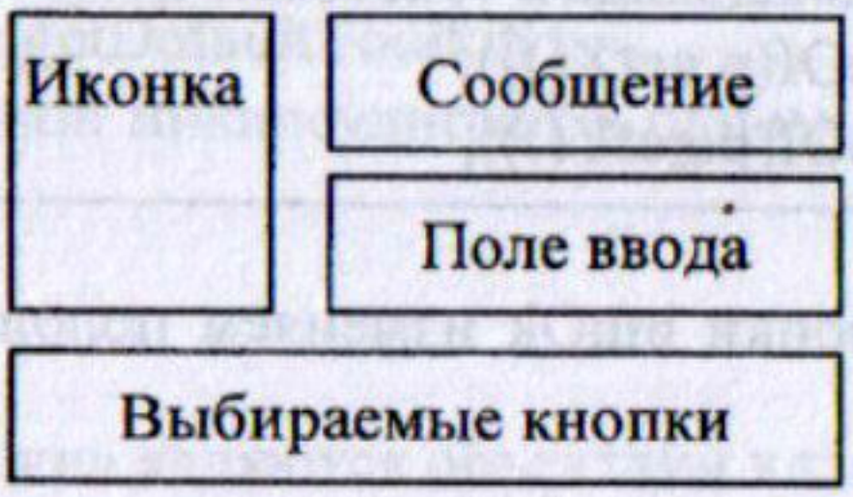
```
new MyDialog(this,                    // ссылка на владельца
```

```
        "Установка расположения",    // надпись в заголовке
        true                         // признак модальности
    );
```

7.4. Класс JOptionPane

В программах с графическим интерфейсом встречается ряд типовых ситуаций, в которых нужны небольшие диалоговые окна. Например, необходимо выдать сообщение пользователю, получить от пользователя строку текста, предоставить пользователю возможность выбора одной из нескольких альтернатив и т.п.

Для создания простых диалоговых окон и предназначен класс JOptionPane. Объект этого класса играет роль посредника между программистом и объектом диалогового окна, при этом создавать класс, наследующий JDialog, не требуется. Программист может задать вид иконки, текст в заголовке, сообщение пользователю, состав кнопок и текст на них. Окна, которые получаются, имеют следующую структуру.



Создаваемые при помощи JOptionPane окна всегда модальные.

Пример "Выбор пути". Создать диалоговое окно с тремя кнопками. В результате выбора пользователя вернуть выбранную опцию.

⇓

```
String mes = "Налево пойдешь - коня потеряешь, направо пойдешь - клад найдешь, прямо пойдешь - получишь одни неприятности.";
JOptionPane pane = new JOptionPane();

// Настроить диалог
Object[] options = {"Налево", "Прямо", "Направо"};
pane.setOptions(options);
pane.setMessage(mes);
pane.setMessageType(JOptionPane.QUESTION_MESSAGE);

// Создать и открыть диалоговое окно
```

```
JDialog dialog = pane.createDialog(this, "title");
dialog.show();

// Получить результат выбора
Object selectedValue = pane.getValue();
```

Основным элементом настройки является массив опций и это может быть массив любых объектов. Если элементом массива является объект JComponent, он будет помещен в окно диалога. В противном случае будет создана кнопка, а элемент массива будет преобразован в строку, которая будет написана на кнопке.

Если пользователь закрыл окно, не сделав выбора, метод getValue() вернет

null.

Статические методы JOptionPane

В классе JOptionPane есть несколько статических методов, которые сводят создание и настройку модального диалога к одному оператору.

Их общее название showXxxDialog(), где вместо Xxx нужно подставить:

- Message - чтобы получить окно с сообщением для пользователя;
- Option - чтобы получить окно с выбором альтернативы;
- Confirm - чтобы получить окно с запросом на подтверждение операции;
- Input - чтобы получить окно с полем ввода, способное принять от пользователя строку текста..

Пример "Выбор пути-2". Сделать то же, что в предыдущем примере, но при помощи одного из статических методов класса JOptionPane.

⇓

```
int res = JOptionPane.showOptionDialog(
  this,        // окно-владелец
  mes,         // сообщениее "Налево пойдешь...
  "title",     // заголовок диалогового окна
  0,           // тип опции, здесь роли не играет
  JOptionPane.QUESTION_MESSAGE,   // тип сообщения
  null,        // иконка, здесь не используется
  options,     // массив опций
  null         // выбор по умолчанию, здесь роли не играет
);
```

Функция возвращает номер выбранной опции или -1, если выбор не сделан.

7.5. Стандартные диалоги

Стандартные диалоги применяются, когда надо выбрать файл на диске (класс JFileChooser) или цвет (класс JColorChooser). Стандартные диалоги не являются простыми объектами, но они очень специализированы, поэтому их использовать довольно просто.

Пример "Просто выбрать файл". Открыть окно выбора файла, выбрать один из имеющихся файлов и сделанный выбор сохранить в объекте File.

⇓

```
// Создать диалог выбора файла
JFileChooser fc = new JFileChooser();
// Открыть диалог выбора файла
int returnVal = fc.showOpenDialog(this);
// Получить выбранный файл
File file = fc.getSelectedFile();
```

Все это нужно проделать в ответ на выбор некоторого пункта меню или нажатие кнопки.

Перед открытием стандартный диалог можно настроить, задав:

- вид диалога - надпись в заголовке окна и на кнопке выбора;
- текущий каталог или файл;
- фильтр для просматриваемых файлов.

Вид диалога

В классе JFileChooser есть три функции, при помощи которых можно открыть модальное окно диалога:

- public int showOpenDialog(Component parent);
- public int showSaveDialog(Component parent);
- public int showDialog(Component parent, String approveButtonText).

Все они получают в качестве аргумента родительский объект, который определяет окно-владельца и компонент, относительно которого диалог будет расположен на экране.

Все функции возвращают одно из трех значений:

- CANCEL_OPTION – пользователь закрыл окно кнопкой Cancel;
- APPROVE_OPTION - пользователь закрыл окно кнопкой выбора;
- ERROR_OPTION – в процессе работы диалога произошла ошибка.

Все значения определены в виде статических констант класса JFileChooser.

Различие между открываемыми окнами незначительно и заключается в надписи, которая появляется в заголовке окна и на кнопке выбора.

Текущий каталог или файл

Методом

```
void setSelectedFile(File selectedFile)
```

можно задать тот файл, который будет выбран сразу после открытия окна.

Текущий каталог, который будет виден в окне после открытия, можно задать методом

```
void setCurrentDirectory(File currentDir);
```

или в конструкторе объекта

```
JFileChooser(File currentDir).
```

Если этого не сделать, в окне будет открыт каталог, заданный в системном свойстве "user.home". Напомним, что рабочему каталогу соответствует системное свойство "user.dir".

Фильтры для файлов

Если нужно, чтобы при просмотре дерева каталогов были видны не все файлы, в объект JFileChooser следует установить фильтр. Фильтром может служить объект класса, который расширяет абстрактный класс

```
javax.swing.filechooser.FileFilter.
```

Класс имеет два абстрактных метода, один из которых

```
public abstract boolean accept(File f)
```

собственно и выполняет фильтрацию, а второй

```
public abstract String getDescription()
```

возвращает строку описания, которая будет включена в выпадающий список выбора фильтра, например, "HTM и HTML – web-документы".

Пример "Фильтр в диалоге". Открыть диалоговое окно, в котором будут видны файлы с расширением java и class. В окне должен быть показан рабочий каталог.

⇓

```
// Создать диалог выбора файла
// с рабочим каталогом в роли текущего
JFileChooser fc = new JFileChooser(new File("."));

// Создать и установить фильтр
fc.setFileFilter(
  new javax.swing.filechooser.FileFilter() {
```

```
        public boolean accept(File f) {
            return   f.isDirectory() ||
                     f.getName().endsWith(".java") ||
                     f.getName().endsWith(".class");
        }

        public String getDescription() {
            return "Файлы Java (*.java, *.class)";
        }
    }
);

// Открыть диалог выбора файла
int returnVal = fc.showOpenDialog(this);
```

Чтобы осуществлять навигацию, в окне выбора файла, фильтр должен пропускать не только нужные файлы, но и каталоги.

Выбор цвета при помощи диалога JColorChooser

Класс JColorChooser имеет три уровня программных средств, перечислим их в порядке возрастания сложности:

- статический метод showDialog(), который открывает модальное окно выбора цвета и возвращает выбранный пользователем цвет;
- статический метод createDialog(), который создает окно JDialog с панелью JColorChooser внутри и задает слушателей для событий нажатия кнопок в окне диалога;
- средства создания панелей выбора цвета внутри любого контейнера с заданием слушателей для событий изменения текущего цвета.

Пример "Выбор цвета". Выберем цвет при помощи статического метода showDialog().

⇓

```
Color color = JColorChooser.showDialog(
  this,                  // родительское окно
  "Выбор цвета",   // надпись в заголовке окна
  Color.red              // исходное значение цвета
);
```

Любой из параметров, и даже все три сразу, могут быть равны null.

7.6. Задание

1. Сделать приложение, которое по щелчку мыши открывало бы новую форму - копию главной. При этом общее число открытых форм не должно превышать семи.
2. Известно, что форму можно перемещать по экрану, таская ее за заголовок. Создайте форму, которую можно было бы перемещать, таская за любую ее часть. **Совет:** если на форме есть компоненты, воспользуйтесь стеклянной панелью (glass pane). Сделайте панель видимой.
3. Создайте форму и к ней диалог, в котором во время выполнения программы можно было бы устанавливать основные свойства формы: цвета, шрифт, иконку и т.п. числом не менее 8.
4. Снабдите текстовый редактор функциями открытия и сохранения файла, и при помощи диалогов JOptionPane обеспечьте напоминание о возможной потере изменений текста.
5. Создайте текстовую панель, в которой любую часть текста можно окрасить в любой цвет. Цвет выбирайте при помощи компонента JColorChooser.

8. РИСОВАНИЕ В SWING

8.1. Общие правила

Под рисованием мы понимаем вывод графики (линий, фигур, закраски) и текста на поверхность компонентов. Рисовать нужно лишь тогда, когда для построения пользовательского интерфейса нельзя обойтись стандартными компонентами Swing. Например, битовое изображение или текст можно поместить в стандартный компонент JLabel.

Чтобы нарисовать что-то на поверхности компонента, надо унаследовать этот компонент и перекрыть один из его методов. Обычный предок пользовательских компонентов, предназначенных для рисования, - JPanel, обычное место для кода рисования - метод paintComponent().

Как и в пакете AWT, рисование выполняется при помощи объекта Graphics, ссылку на который надо получить перед рисованием. Впрочем, в метод paintComponent() уже готовая ссылка передается в качестве параметра.

Код метода paintComponent() должен начинаться с вызова метода предка super.paintComponent() для зарисовки фона. Если не вызывать этот метод, фон надо зарисовать самому или сделать компонент прозрачным (setOpaque(false)), чтобы просвечивал фоновый компонент.

Область рисования

Область рисования - это прямоугольник компонента, из которого исключена площадь, закрываемая рамкой. Чтобы программно определить область рисования, вызывают метод компонента getInsets(), который возвращает объект Insets с 4-мя полями: left, right, top и bottom.

⇓

```
Insets insets = getInsets();
// Ширина области рисования
int width = getWidth() - insets.left - insets.right;
// Высота области рисования
int height = getHeight() - insets.top - insets.bottom;
```

Перерисовка

Метод repaint() сообщает системе, что компонент нуждается в перерисовке. Вариант метода repaint(int left, int top, int width, int height) говорит, что в перерисовке нуждается лишь часть компонента.

Прямоугольник перерисовки передается в метод paintComponent в объекте Graphics и может быть извлечен оттуда методом класса Graphics getClipBounds().

⇓

```
public void paintComponent(Graphics g) {
  Rectangle clipRect = g.getClipBounds();
  if (clipRect != null) {
      // Рисуем лишь то, что внутри clipRect
  } else {
      // Рисуем все
  }
}
```

Архитектура отрисовки

Всего существуют три метода, которые компонент использует для отрисовки себя:

- paintComponent() - зарисовывает прямоугольник компонента;
- paintBorder() - рисует рамку, если она есть (не вызывать и не перекрывать!);
- paintChildren() - рисует дочерние компоненты (не вызывать и не перекрывать!).

Имеется также метод paint() (не вызывать и не перекрывать!), который вызывает эти три метода.

Многие Swing-компоненты поручают look-n-fill специфичное рисование объекту, называемому UI-delegate.

Причина, по которой рекомендуется наследовать для рисования JPanel, а не JComponent, в том, что JComponent не делегирует рисование UI-delegate, это делают только его подклассы.

Объект Graphics

Объект Graphics инкапсулирует графический контекст и имеет методы рисования, такие как

drawImage() - нарисовать битовый образ,
drawString() - нарисовать строку текста,
drawRect() - нарисовать контурный прямоугольник,
fillRect() - нарисовать закрашенный прямоугольник.

Графический контекст имеет атрибуты: цвет, шрифт, область рисования, которые устанавливаются методами setColor(), setFont(), setClip() и могут быть получены методами getColor(), getFont(), getClip().

Область рисования можно игнорировать, ибо все, что рисуется за ее пределами, все равно видно не будет, но это может снизить эффективность кода.

8.2. Изображение фигур

Класс Graphics имеет методы для изображения следующих фигур:

- Линий - drawLine();
- Прямоугольников - drawRect() и fillRect();
- Выпуклых прямоугольников - draw3DRect() и fill3DRect();
- Закругленных прямоугольников - drawRoundRect() и fillRoundRect();
- Овалов - drawOval() и fillOval();
- Дуг - drawArc() и fillArc();
- Многоугольников - drawPolygon(), drawPolyline() и fillPolygon().

Методы с названием drawXxx() рисуют контурные изображения, методы с названием fillXxx() - закрашенные фигуры. При вызове методов drawXxx() необходимо задавать ширину и высоту на 1 пиксель меньше желаемых, тогда как при вызове методов fillXxx() надо задавать точно желаемый размер.

⇓

```
g.drawRect(x, y, rectWidth - 1, rectHeight - 1);
g.fillRect(x, y, rectWidth, rectHeight);
```

8.3. Рисование текста

Для вывода текста используются классы Graphics, Font и FontMetrics. Графический вывод текста выполняется методами класса Graphics drawBytes(), drawChars() и drawString().

Пример "Вывод строки".

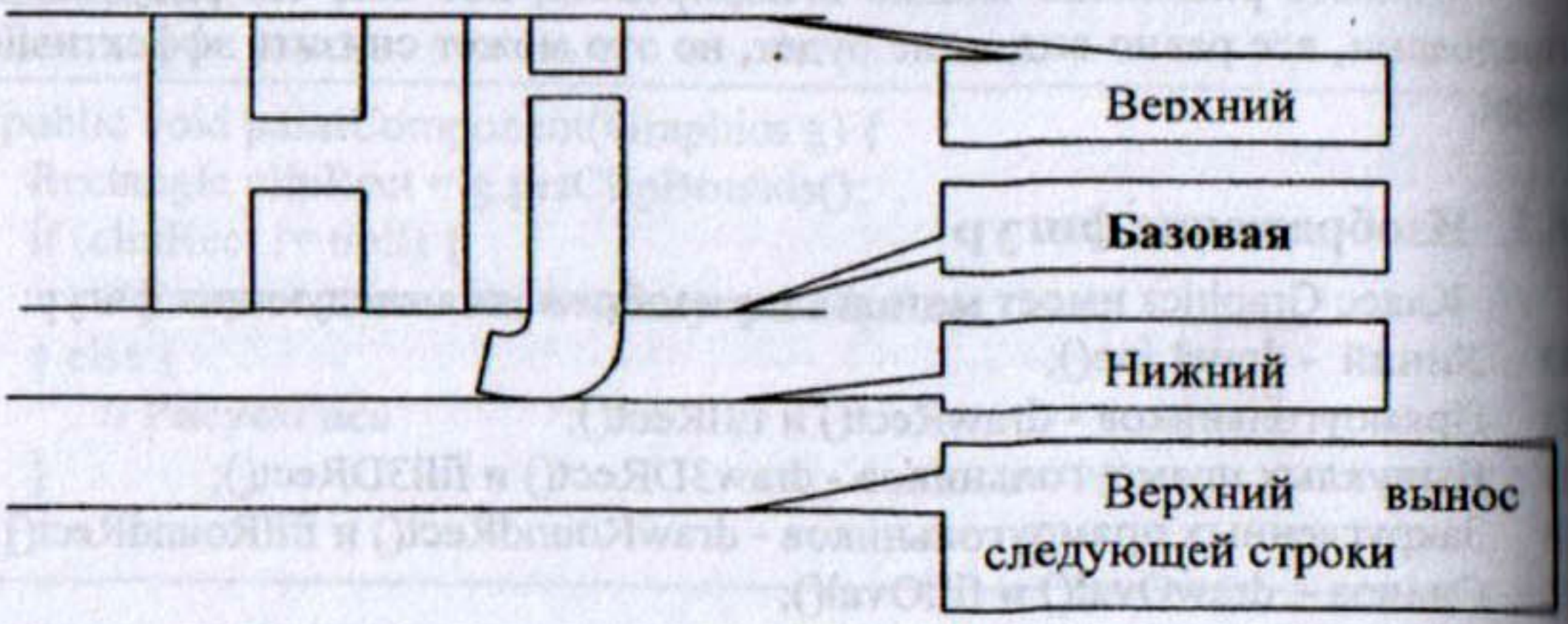
⇓

```
g.drawString(
  "Hello World!",    // выводимое строковое выражение
  10,                // x-координата в пикселях
  50                 // y-координата в пикселях
);
```

Методы drawBytes() и drawChars() мало отличаются от этого.

Вертикальные размеры текста определяются расстояниями:

- от базовой линии до верхнего выноса букв;
- от базовой линии до нижнего выноса букв;
- между базовыми линиями двух соседних строк.



Класс Font

Класс jawa.awt.Font представляет шрифт. Прежде, чем говорить о шрифте составим небольшой словарь.

Символ – число, обозначающее цифру или букву, или знак.

Глиф (glyph) – изображение символа.

Таблица кодировки сопоставляет каждому символу глиф.

Шрифт – это коллекция глифов. Шрифт может иметь много видов: плотный, средний, наклонный, готический, нормальный. Все виды одного шрифта имеют сходный типографский дизайн.

В объекте Font используются три различных имени шрифта:

- логическое имя шрифта – передается в конструктор при создании объекта и идентифицирует объект;
- имя шрифта – например, Helvetica Bold, Courier New;
- имя семейства – название группы шрифтов, имеющих общие черты, например, Helvetica, Courier.

Некоторые методы класса Font приведены в таблице.

Метод	Описание
Font(String name, int style, int size)	Конструктор. Получает логическое имя, стиль и размер шрифта
String getFamily()	Возвращает имя семейства шрифтов
String getName()	Возвращает логическое имя шрифта
String getFontName()	Возвращает имя шрифта
int getStyle()	Стиль шрифта: PLAIN, BOLD, ITALIC или BOLD+ITALIC
int getSize()	Размер в пунктах. 1 пункт=1/72 дюйма
boolean isBold()	Возвращает true, если стиль BOLD
boolean isItalic()	Возвращает true, если стиль ITALIC
Font fontDeriveFont (float size)	Создает новый шрифт на базе старого, но другого размера
Font fontDeriveFont (int style)	Создает новый шрифт на базе старого, но другого стиля
Font fontDeriveFont (AffineTransform trans)	Создает новый шрифт на базе старого, но с аффинным преобразованием

Статические константы для задания стиля определены в классе Font:

- PLAIN - шрифт обычный;
- BOLD - шрифт жирный;
- ITALIC - шрифт наклонный.

Под размером шрифта понимается расстояние между двумя базовыми линиями в тексте с одним межстрочным интервалом.

Аффинное преобразование включает поворот, растяжение и параллельный перенос. Оно представлено объектом AffineTransform, который при создании должен получить 6 чисел - матрицу преобразования.

Пример "Текст наискось". Написать текст "Hello World!" под углом к горизонтали.

Поворот на угол fi без масштабирования и переноса задает матрица из 6 чисел:

| cos(fi) sin(fi) 0 |
| -sin(fi) cos(fi) 0 |

В конструктор объекта AffineTransform она передается по столбцам.

⇓

```
// Трансформация заключается в повороте на 45 градусов
// против часовой стрелки
double fi = Math.PI / 4;
AffineTransform transform = new AffineTransform(
  Math.cos(fi), -Math.sin(fi),
  Math.sin(fi), Math.cos(fi), 0, 0 );

// Из текущего шрифта создаем новый шрифт
Font font = g.getFont().deriveFont(transform);
// Делаем новый шрифт текущим
g.setFont(font);
// Пишем, начав немного снизу, чтоб все поместилось
g.drawString("Hello World!", 0, 100);
```

Класс FontMetrics

Объект класса java.awt.FontMetrics содержит параметры конкретного шрифта Font в конкретном графическом контексте Graphics. Программист должен использовать эти параметры для расчета координат при изображении текста.

Ссылку на объект FontMetrics можно получить методами класса Graphics

FontMetrics getFontMetrics() - для текущего шрифта

и

FontMetrics getFontMetrics(Font font) - для произвольного шрифта.

Некоторые методы класса FontMetric приведены в таблице.

Метод	Описание
int stringWidth(String str)	Возвращает ширину строки str
int charWidth(char ch)	Возвращает ширина символа ср, включая промежуток

int getLeading()	Возвращает промежуток между строками
int getAscent()	Возвращает расстояние от базовой линии текста до верхнего выноса
int getDescent()	Возвращает расстояние от базовой линии текста до нижнего выноса

Пример "Эластичная панель". Создать панель с надписью на ней. Ширина панели должна соответствовать размеру надписи.

Объявим класс MyPanel, который наследует JPanel и имеет поле String text. Метод setText() будет изменять текущее значение поля и перерисовывать панель.

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

public class MyPanel extends JPanel {

  // Изображаемый текст
  private String text = "";

// Изменяет текст и размеры панели
public void setText(String val) {
    text = val;
    Graphics g = this.getGraphics();
    if (g != null) {
        FontMetrics fm = g.getFontMetrics();
        int width = fm.stringWidth(text);
        setSize(width, getHeight());
    }
}

// Выводит надпись
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.drawString(text, 0, 20);
  }
}
```

Чтобы испытать нашу панель, поместим ее на форму и настроим в методе формы init()

⇓

```
private void init() throws Exception  {
  contentPane = (JPanel) this.getContentPane();
  contentPane.setLayout(new FlowLayout());
  setSize(new Dimension(400, 300));

  // Создаем панель
  MyPanel panel = new MyPanel();
  // Изменяем ее цвет, чтоб отличить от фона
  panel.setBackground(new Color(151, 192, 192));
  // Устанавливаем предпочтительный размер, его
  // учитывает менеджер FlowLayout
  panel.setPreferredSize(new Dimension(30, 50));
  // Вставляем нашу панель в панель содержания
  contentPane.add(panel);
}
```

Выберем подходящее событие, например, щелчок мышью по форме, и в его обработчике изменим текст на панели.

⇓

```
panel.setText("1234567890");
```

Заметим, что получение графического контекста компонента возможно, только если этот контекст существует, т.е. компонент уже находится в контейнере, и этот контейнер имеет реальную поверхность рисования. Так вызов метода setText(), когда панель еще не попала на форму или уже попала, но сама форма еще не достроена (конструктор не закончил работу) будет преждевременным.

8.4. Задание

1. Разработать приложение, которое бы позволяло рисовать в окне методом "резиновой линии" (см. задание 2.4). При изменении размеров или повреждении окна рисунок должен восстанавливаться. **Совет**: координаты узлов ломаной надо запоминать в коллекции точек.
2. Разработать метод, который бы писал заданную строку, начав с точки x, y, под углом в а радиан к оси Ox.
3. Создать панель, надпись на которой увеличивалась бы с увеличением ширины панели, постоянно занимая всю ее ширину.
4. Накрыть рабочую область формы полупрозрачной панелью. Прозрачность панели задавать при помощи текстового поля на форме.

5. Создать панель, которая служила бы простым текстовым редактором для рисования прямоугольников.

9. БИТОВЫЕ ИЗОБРАЖЕНИЯ

Битовые изображения являются важным элементом пользовательского интерфейса. Они могут находиться на кнопках, пунктах меню, служить иконками, фоном и т.п. Работа с изображениями в приложениях и апплетах выполняется сходным образом, но здесь мы рассмотрим только приложения, а об особенностях апплетов поговорим позже.

Классы, необходимые для работы с изображениями, находятся в пакетах java.awt, и java.awt.image, основной класс называется java.awt.Image.

9.1. Загрузка изображения

Как правило, изображения хранятся в файлах, которые расположены на локальном компьютере или в сети. Чтобы увидеть такое изображение на экране, его надо загрузить в память и вывести на экран.

В классе java.awt.Toolkit есть метод

Image getImage(String filename),

который создает объект Image, для изображения, хранящегося в файле filename формата GIF или JPG.

Метод getImage() создает объект Image и возвращает его немедленно, даже не проверяя, существует ли файл. Настоящая загрузка начнется не ранее попытки показать изображение на экране.

9.2. Рисование изображения

Попытаться нарисовать изображение может метод grawImage() класса Graphics.

Пример "Рисование изображения". По нажатию кнопки загрузить и нарисовать загруженное изображение.

⇓

```
Toolkit toolkit = Toolkit.getDefaultToolkit();
Image image = toolkit.getImage("c:/some.jpg");
Graphics g = getGraphics();
g.drawImage(image, 0,0, null);
```

Верхний левый угол изображения будет расположен в точке (0, 0) прямоугольника компонента, назначение последнего параметра поясним позже.

Помимо простейшего метода grawImage(), класс Graphics имеет другие его варианты, учитывающие прямоугольник изображения, прямоугольник компонента, фоновый цвет. Например, метод

boolean drawImage(Image img, int x, int y, int width, int height, Color bgcolor, ImageObserver observer)

займет под изображение прямоугольник шириной width и высотой height, сжав или растянув изображение для этого. Фоновый цвет будет виден под прозрачными частями изображения.

Напомним, что цвет в Java складывается из 4-х составляющих: alpha и RGB. Значение alpha изменяется от 0 до 255 (или от 0,0 до 1,0) и определяет прозрачность: 0 – цвет совсем прозрачный, 255 – цвет совсем не прозрачный.

9.3. Контроль загрузки

Посмотрим, что происходит во время выполнения кода из предыдущего примера.

Строка кода

```
Image image = toolkit.getImage("c:/some.jpg");
```

создаст объект Image, но не изображение. Загрузка начнется в результате вызова метода

```
g.drawImage(image, 0,0, null);
```

и будет происходить в отдельном потоке выполнения. Ничего вывести на экран метод drawImage() не сможет, т.к. изображение еще не будет готово. Возвращаемое методом drawImage() значение как раз и говорит, удалось нарисовать изображение (true) или не удалось (false).

Если же нажать кнопку повторно, изображение появится на форме. Причина в том, что при повторном вызове метод drawImage() получит объект image в ином состоянии, изображение будет полностью загружено в память.

Интерфейс ImageObserver

Наблюдать за процессом загрузки и отвечать на запросы о текущем состоянии загружаемого изображения может любой объект, реализующий интерфейс java.awt.image.ImageObserver. Чтобы объект выполнял свои функции наблюдателя, его надо зарегистрировать в этом качестве перед началом загрузки. Это и делает метод drawImage(), получая ссылку на ImageObserver в последнем параметре.

Интерфейс ImageObserver имеет единственный метод

boolean imageUpdate (Image img, int infoflags,
int x, int y, int width, int height),

который будет периодически вызываться из потока загрузки до тех пор, пока метод возвращает true или поток загрузки не закончит свою работу.

Метод получает наблюдаемый объект img, его состояние в виде набора битовых флагов infoflags, координаты x, y, ширину width и высоту height изображения, которые истолковываются в зависимости от значений флагов.

Помимо единственного метода, интерфейс включает определение ряда целочисленных констант, полезных для проверки состояния отдельных флагов.

Пример "Наблюдение за загрузкой". Объявить класс наблюдателя и использовать его для синхронизации загрузки и вывода изображения.

⇓

```
class MyObserver implements ImageObserver {

 public boolean imageUpdate(Image img, int infoflags,
          int x, int y, int width, int height)
 {
    if ((infoflags & ALLBITS) == 0)
        // Флаг ALLBITS сброшен, загрузка не окончена
        return true;
    else {
        // Флаг ALLBITS установлен, загрузка полностью завершена
        Graphics g = getGraphics();
        // Выводим изображение уже без наблюдателя
        g.drawImage(img, 0,0, null);
        // Сигнализируем о том, что вызывать метод больше незачем
        return false;
    }
 }
}
```

Внесем поправку в код примера "Рисование изображения".

⇓

```
Toolkit toolkit = Toolkit.getDefaultToolkit();
Image image = toolkit.getImage("c:/some.jpg");
Graphics g = getGraphics();
g.drawImage(image, 0,0, new MyObserver() );
```

Заметим, что все потомки класса Component реализуют интерфейс ImageObserver, поэтому решение можно упростить - не создавать новый класс и объект, а перекрыть метод imageUpdate(), который есть у формы и передать ссылку this в последнем аргументе метода drawImage().

Класс MediaTracker

Вам, вероятно, показалось, что интерфейс ImageObserver понятен и прост в использовании, но большинство программистов думают иначе, особенно, когда нужно работать с множеством изображений. Поэтому разработчики Java предложили класс java.awt.MediaTracker, с помощью которого можно проверять состояние произвольного числа изображений.

Применение класса MediaTracker состоит в создании объекта, присоединении его к одному или более образам и последующем опросе объекта MediaTracker о состоянии процесса загрузки. Можно также воспользоваться методами ожидания, которые есть у MediaTracker.

Пример "Рисование изображения-2". По нажатию кнопки загрузить и нарисовать изображение. Для ожидания конца загрузки использовать объект MediaTracker.

⇓

```
// Это мы уже делали раньше
Toolkit toolkit = Toolkit.getDefaultToolkit();
Image image = toolkit.getImage("c:/some.jpg");
Graphics g = getGraphics();

// В конструктор передаем компонент с изображениями
MediaTracker mtr = new MediaTracker(this);
// Присоединяем образ и даем ему числовой идентификатор (здесь 0)
mtr.addImage(image, 0);
// Ждем, пока загрузка закончится
mtr.waitForID(0);
// Рисуем уже загруженное изображение
g.drawImage(image, 0,0, this);
```

Метод MediaTracker.waitForID() ждет завершения процесса загрузки. Этот метод может выбрасывать исключение InterruptedException, которое нужно отследить (в примере это опущено).

9.4. Класс ImageIcon

Проблемы загрузки актуальны при работе в сети, особенно когда скорость передачи данных невелика. В настольных приложениях и локальных сетях загрузка происходит незаметно, поэтому лучшим выбором при работе с изображениями в библиотеке Swing является уже знакомый нам компонент

javax.swing.ImageIcon.

Этот компонент содержит MediaTracker внутри себя и скрывает все проблемы синхронизации.

Пример "Рисование изображения-3". По нажатию кнопки загрузить и нарисовать изображение. Воспользоваться компонентом ImageIcon.

⇓

```
ImageIcon icon = new ImageIcon("c:/cat.gif");
Graphics g = getGraphics();
g.drawImage(icon.getImage(), 0,0, null);
```

Когда конструктор компонента заканчивает работу, загрузка изображения уже завершена и можно рисовать.

9.5. Создание изображения в памяти

Изображение не обязательно загружается из файла, оно может быть построено непосредственно в программе. Для программного доступа к пикселям изображения служит класс java.awt.image.BufferedImage, который расширяет уже известный нам java.awt.Image.

Рассмотрим процесс программного создания изображения на примере.

Во-первых, создадим объект BufferedImage, передав в конструкторе ширину, высоту и тип изображения. Тип TYPE_INT_ARGB означает, что каждый пиксель изображения описывается четырьмя числами: для красного, зеленого и синего цвета и альфа-канала, определяющего прозрачность.

⇓

```
int width = 400, height = 400;
BufferedImage img =
   new BufferedImage(width, height, BufferedImage.TYPE_INT_ARGB);
```

Во-вторых, получим объект java.awt.image.WritableRaster, который обеспечит нам доступ к отдельным пикселям.

⇓

```
WritableRaster raster = img.getRaster();
```

Объект WritableRaster имеет методы getPixel() и setPixel(), которые позволяют получить и задать 4 числа, определяющие цвет любого пиксела растра. Следующий код строит изображение, напоминающее шахматную доску из ценных пород дерева.

⇓

```
// Константы, определяющие размеры клеток
final int D = 40, D2 = D * 2;
for (int x = 0; x < width; x++)
for (int y = 0; y < height; y++) {
 if (x % D2 < D && y % D2 > D || x % D2 > D && y % D2 < D)
     // Ставим пиксель в "белую" клетку
     raster.setPixel(x, y, new int[] {255, 255, x * y , 255});
 else
     // Ставим пиксель в "черную" клетку
     raster.setPixel(x, y, new int[] {x * y, 0 , 0 , 255});
}
```

Чтобы поскорее увидеть результат, поместим его в метку, а метку на форму.

⇓

```
JLabel label = new JLabel();
label.setIcon(new ImageIcon(img));
contentPane.add(label,  BorderLayout.CENTER);
```

Создавая изображение в памяти, можно пользоваться всеми методами рисования объекта Graphics. Получить этот объект для создаваемого изображения позволит метод getGraphics(), который BufferedImage унаследовал от класса Image.

Оконтурим нашу доску тонкой красной линией.

⇓

```
Graphics g = img.getGraphics();
g.setColor(Color.red);
g.drawRect(0, 0, width - 1, height - 1);
```

9.6. Сохранение изображения в файле

В JDK 1.3 нет средств для сохранения изображений в файлах, но это легко сделать при помощи пакетов третьих фирм. Наиболее известным классом для сохранения изображений в формате GIF является класс Acme.JPM.Encoders.GifEncoder, разработанный Джефом Посканзером (Jeff

Poscanzer). Исходный текст класса доступен по адресу http://www.acme.com/java.

Для работы вам понадобятся всего два класса пакета Acme.JPM.Encoders: GifEncoder и ImageEncoder. Разместите их в подкаталоге Acme.JPM.Encoders и включите в состав исходных текстов вашей программы.

Чтобы сохранить изображение Image myImage в файле "myImage.gif" достаточно создать экземпляр класса GifEncoder и вызвать его метод encode() (обработка исключения IOException в примере не показана).

⇓

```
OutputStream out = new FileOutputStream("myImage.gif");
(new Acme.JPM.Encoders.GifEncoder(myImage, out)).encode();
out.close();
```

Подобные же классы существуют и для других графических форматов, а в версии JDK 1.4 средства сохранения изображения в файлах включены в состав ядра.

9.7. Анимация

Под анимацией здесь понимается быстрая смена статических изображений, которая создает для наблюдателя эффект движения.

Один из способов анимации состоит в создании массива изображений и запуске отдельного потока вычислений, который будет по очереди выводить их на экран. Если смена изображений покажется слишком быстрой, между ними надо делать небольшую паузу.

Пример "Простая анимация". Загрузить два изображения в формате GIF и поочередно выводить их на форму.

⇓

```
// Создаем массив из двух изображений
Image[] images = {new ImageIcon("9_4_left.gif").getImage(),
                  new ImageIcon("9_4_right.gif").getImage() };

// Создаем и запускаем поток анимации
(new Thread() {
 public void run() {
     // Пауза между соседними изображениями в мсек
     int pause = 200;
     // Номер очередного изображения
     int i = 0;
     while (true) {
```

```
        Graphics g = getContentPane().getGraphics();
        // На тот случай, если графический контекст еще не готов
        if (g == null)
            continue;
        // Вывести i-ое изображение
        g.drawImage(images[i], 50,50, null);
        // Вычислить номер следующего изображения
        i = (++i) % images.length;
        // Сделать паузу, чтоб не частило
        try {sleep(pause);} catch (Exception ex) {}
        }
      }
    }
  }).start();
```

Другой способ периодически чередовать изображения состоит в применении объекта Timer. Существует два класса Timer, один в пакете java.util, а другой в пакете javax.swing. Оба таймера могут создать поток выполнения, являющийся источником периодических событий, но поскольку мы сейчас изучаем библиотеку Swing, рассмотрим класс javax.swing.Timer.

Уже в конструкторе объекта Timer можно задать период повторения событий и зарегистрировать слушателя событий ActionListener. Остается выполнить смену изображений в коде слушателя и включить таймер методом start().

Пример "Простая анимация-2".

⇓

```
// Создать таймер с интервалом повторений в 0.2 сек
Timer timer = new Timer(200, new ActionListener()
  {
      int i; // номер очередного изображения
      public void actionPerformed(ActionEvent evt) {
          Graphics g = getContentPane().getGraphics();
          if (g != null)
              g.drawImage(images[i], 50,50, null);
          i = (++i) % images.length;
      }
  }
);
// Включить таймер
```

```
timer.start();
```

9.8. Задание

1. Разработать приложение, которое бы позволяло загружать картинку из файла и рисовать на ее поверхности.
2. Программно создать изображение, которое напоминало бы раскраску зебры. Для вывода изображения на панель переопределить метод панели paintComponent().
3. Дополнить приложение из предыдущего примера возможностью сохранения созданного изображения в файле формата gif. Для сохранения изображения воспользуйтесь классом Acme.JPM.Encoders.GifEncoder.
4. Сделать анимацию при помощи чередования нескольких связанных изображений, например, человечек с опущенными руками, с поднятыми до уровня плеч, с поднятыми вертикально вверх. Изменять скорость анимации ползунком.
5. Изобразить бильярдный шар, движущийся по клиентской области формы и отражающийся от ее стенок.

10. ОТОБРАЖЕНИЕ ТАБЛИЦ

Платежные ведомости, реестры, перечни, матрицы - словом все, в чем мы способны усмотреть столбцы и строки, можно показать на экране в виде таблиц. В библиотеке Swing за представление таблиц отвечает компонент JTable.

10.1. Создание простой таблицы

Простую таблицу можно создать при помощи конструктора, который получает одномерный массив заголовков и двухмерный массив содержимого таблицы.

Пример "Создание таблицы". Создать таблицу из четырех строк и трех столбцов.

⇓

```
// Массив заголовков таблицы
String[] columnNames =
  {"Язык", "Год рождения", "Популярность"};

// Массив содержимого таблицы
Object[][] data = {
```

```
    {"C++", new Integer(1985), "30%"},
    {"Java", new Integer(1995), "20%"},
    {"Pascal", new Integer(1970),"5%"},
    {"Lisp", new Integer(1960), "0.1%"}
};

JTable table = new JTable(data, columnNames);
```

10.2. Размещение таблицы в контейнере

После создания таблицу можно поместить в контейнер компонентов непосредственно или через посредника - компонент JScrollPane.

⇓

```
// Поместить на форму объект JScrollPane
JScrollPane scroll = new JScrollPane();
contentPane.add(scroll);
// Поместить таблицу в объект JScrollPane
scroll.getViewport().add(table);
```

Если не использовать объект JScrollPane, то заголовки таблицы не будут видны. Это можно исправить, если воспользоваться компонентом JTableHeader, ссылку на который можно получить методом таблицы getTableHeader().

⇓

```
contentPane.add(table.getTableHeader(), BorderLayout.NORTH);
contentPane.add(table, BorderLayout.CENTER);
```

10.3. Установка и изменение ширины столбцов

По умолчанию все столбцы созданной таблицы одинаковой ширины, и именно такой, чтобы таблица заполняла все отведенное ей пространство контейнера.

Изменять ширину столбцов можно вручную, перетаскивая границы мышью, или программно, методом setPreferredWidth() объекта TableColumn.

Класс TableColumn находится в пакете javax.swing.table и представляет один столбец таблицы. Ссылку на объект столбца можно получить методом таблицы getColumn().

Пример "Ширина столбцов". Программно задать ширину столбцов таблицы table.

⇓

```
// columnNames - массив заголовков столбцов
for (int i = 0; i < columnNames.length; i++) {
  // Получаем объект для очередного столбца
  TableColumn column =  table.getColumn(columnNames[i]);
  // Первый столбец делаем широким
  if (i == 0)
      column.setPreferredWidth(200);
  // Последний столбец делаем узким
  if (i == columnNames.length - 1)
      column.setPreferredWidth(20);
}
```

Отметим, что так мы задали только предпочтительную ширину столбцов, и последнее слово в определении их ширины останется за самой таблицей.

10.4. Выбор строк, столбцов и ячеек

По умолчанию в таблице можно выбрать один столбец и одну или несколько строк, не обязательно идущих подряд.

Пример "Анализ выбранного". Определить, какие строки и столбцы выбраны в таблице.

⇓

```
// Определить номер выбранного столбца
int col = table.getSelectedColumn();
// Определить номер первой выбранной строки
int row = table.getSelectedRow();
// Определить номера выбранных строк
int[] rows = table.getSelectedRows();
```

Изменить режим выбора можно методами таблицы:
setRowSelectionAllowed(boolean);
setColumnSelectionAllowed(boolean);

10.5. Модель таблицы

Все рассмотренные приемы работы с таблицей игнорировали модель таблицы, которая у нее есть, как у всякого компонента библиотеки Swing.

Моделью таблицы может служить любой объект, который реализует интерфейс javax.swing.table.TableModel. Методы этого интерфейса приведены в таблице.

Метод	Описание
int getColumnCount()	Возвращает количество столбцов таблицы
int getRowCount()	Возвращает количество строк таблицы
String getColumnName (int columnIndex)	Возвращает наименование столбца с номером columnIndex
Class getColumnClass (int columnIndex)	Возвращает тип столбца с номером columnIndex
boolean isCellEditable (int rowIndex, int columnIndex)	Определяет, можно ли редактировать ячейку таблицы
Object getValueAt (int rowIndex, int columnIndex)	Читает значение из ячейки
void setValueAt (Object value, int rowIndex, int columnIndex)	Пишет значение value в ячейку
void addTableModelListener (TableModelListener l)	Добавляет слушателя событий модели таблицы
void removeTableModelListener (TableModelListener l)	Удаляет слушателя событий модели таблицы

Некоторая универсальная модель таблицы частично реализована абстрактным классом AbstractTableModel. Чтобы получить конкретную модель достаточно унаследовать этот класс и реализовать три метода:

```
public int getRowCount();
public int getColumnCount();
public Object getValueAt(int row, int column).
```

Пример "Модель таблицы умножения". Создать модель таблицы умножения, которую нельзя редактировать.

⇓

```
TableModel model = new AbstractTableModel() {
  public int getColumnCount() { return 10; }
  public int getRowCount()    { return 10; }
  public Object getValueAt(int row, int col) {
      return new Integer(row*col);
  }
};
```

Чтобы увидеть, как это выглядит, надо назначить эту модель уже существующей таблице

⇓

```
table.setModel(model);
```

или создать новую таблицу с этой моделью.

⇓

```
JTable table = new JTable(model);
```

Пример "Создание таблицы-2". Разработаем модель таблицы для примера с языками программирования. Массивы заголовков (columnNames) и данных (data) станут теперь частью модели.

⇓

```
class MyModel extends AbstractTableModel {

  String[] columnNames =
      {"Язык", "Год рождения", "Популярность"};

  Object[][] data = {
      {"C++", new Integer(1985), "30%"},
      {"Java", new Integer(1995), "20%"},
      {"Pascal", new Integer(1970),"5%"},
      {"Lisp", new Integer(1960), "0.1%"}
  };

  public String getColumnName(int col) {
      return columnNames[col].toString();
  }
  public int getRowCount() {
      return data.length;
  }
  public int getColumnCount() {
      return columnNames.length;
  }
  public Object getValueAt(int row, int col) {
      return data[row][col];
  }
  public boolean isCellEditable(int row, int col) {
      return true;
  }
  public void setValueAt(Object value, int row, int col) {
      data[row][col] = value;
      fireTableCellUpdated(row, col);
```

```
    }
};
```

Создадим модель и таблицу с этой моделью.

⇓

```
MyModel model = new MyModel();
JTable table = new JTable(model);
```

Использование модели при работе с таблицей дает те же преимущества, что и использование модели при работе с текстом: улучшение структуры и увеличение гибкости программного кода.

Например, теперь легко сделать так, чтобы данные столбцов отображались в соответствии с их типом, т.е. числа выравнивались бы по правому краю ячейки, а не по левому, как строки. Для этого изменим метод модели getColumnClass().

⇓

```
public Class getColumnClass(int c) {
  // Типом столбца считаем тип данных в 0-ой строке
  return getValueAt(0, c).getClass();
}
```

Сделаем так, чтобы редактировать можно было не все столбцы, а только последний столбец. Для этого откорректируем метод isCellEditable().

⇓

```
public boolean isCellEditable(int row, int col) {
  return col == getColumnCount() - 1;
}
```

Хотя столбцы таблицы можно переставлять при помощи мыши, адреса ячеек в модели остаются неизменными и не зависят от того, где в данный момент столбец находится на экране.

10.6. Реакция на изменение данных

Если данные модели изменились программно, то сама таблица не отобразит изменения. Чтобы увидеть сделанные изменения, можно перерисовать всю таблицу методом repaint(), а можно действовать более избирательно, возбудив событие TableModelEvent, несущее информацию о произошедшем изменении.

Для возбуждения события TableModelEvent, но с разной начинкой, предназначены различные методы fireXxx() класса AbstractTableModel:

fireTableCellUpdated(),
fireTableChanged(),
fireTableDataChanged(),
fireTableRowsDeleted(),
fireTableRowsInserted(),
fireTableRowsUpdated() и
fireTableStructureChanged().

Пример "Уведомление об изменении". Изменить данные в ячейке модели и возбудить событие TableModelEvent.

⇓

```
model.data[1][0] = "Java-2";
model.fireTableCellUpdated(1,0);
```

Если же нужно отреагировать на изменение данных иначе, чем простым обновлением таблицы, следует создать и зарегистрировать слушателя событий TableModelEvent.

Пример "Слушатель событий модели".

⇓

```
model.addTableModelListener(
  new TableModelListener() {
      public void tableChanged(TableModelEvent e) {
          // Определяем координаты ячейки
          int col = e.getColumn();
          int row = e.getFirstRow();
          // Что-нибудь делаем с данными
          model.data[row][col] = "-" + model.data[row][col] + "-";
  }
  }
);
```

10.7. Отображение ячеек

Компонент JTable позволяет программисту индивидуально отображать содержимое каждой ячейки таблицы.

Для вывода на экран содержимого ячейки таблицы JTable служит объект javax.swing.table.DefaultTableCellRenderer, реализующий интерфейс TableCellRenderer. Этот интерфейс имеет единственный метод

Component getTableCellRendererComponent(JTable table, Object value,
boolean isSelected, boolean hasFocus, int row, int column),

и он вызывается рамкой всякий раз, когда надо отобразить содержимое конкретной ячейки таблицы. Метод принимает в виде параметров все данные, которые могут понадобиться для рисования ячейки, и возвращает настроенный компонент, который отобразит содержимое ячейки.

Если вы хотите нестандартно рисовать таблицу, нужно унаследовать класс DefaultTableCellRenderer и перекрыть этот метод.

Пример "Полосатая таблица". Для улучшения читабельности чередовать белый и серый фон в строках таблицы.

Снабдим таблицу, которую мы строили в примере "Создание таблицы-2", нестандартным классом для отображения ячеек.

⇓

```
class MyRenderer extends DefaultTableCellRenderer {

  public Component getTableCellRendererComponent(
      JTable table, Object value, boolean isSelected, boolean hasFocus,
      int row, int column)
  {
                    .
      setText(table.getValueAt(row, column).toString());
      if (row % 2 == 0)
          setBackground(Color.lightGray );
      else
          setBackground(Color.white);
      return this;
  }
}
```

Создадим объект этого класса и назначим его для отрисовки ячеек. Назначение выполняется отдельно для каждого столбца таблицы table.

⇓

```
MyRenderer renderer = new MyRenderer();
for (int i = 0, n = table.getColumnCount(); i < n; i++) {
  table.getColumnModel().getColumn(i).setCellRenderer(renderer);
}
```

10.8. Задание

1. Создать таблицу, которая показывает таблицу умножения для P-ичной системы счисления (P<=10). Заголовки столбцов таблицы должны содержать числа: 0, 2, 3, 4, …, P. Такие же числа должны находиться в первом столбце таблицы. В ячейках таблицы должны быть соответствующие произведения, записанные в системе счисления с основанием P.
2. Создать таблицу, которая способна хранить числа и автоматически подсчитывать суммы по строкам и столбцам. Суммы должны изменяться при изменении содержимого таблицы.
3. Создать таблицу с черными и белыми клетками, как на шахматной доске. В черных клетках символы должны быть белыми, а в белых - черными.
4. Создать таблицу размером 3 x 3, в каждой ячейке которой находится изображение крестика или нолика. Сделать так, чтобы по щелчку мышью по изображению крестика оно сменялось изображением нолика и наоборот.
5. Построить таблицу, которая показывает список файлов каталога. В таблице должны быть столбцы: "Name", "Size", "Type" и "Modified". Щелчок мышью по заголовку столбца должен вызывать сортировку строк по значениям этого столбца. Каталог задавать в командной строке.

11. ОТОБРАЖЕНИЕ ДЕРЕВЬЕВ

Различные иерархии встречаются в нашей жизни довольно часто. Это и подчиненность сотрудников в учреждении, и структура подразделений того же учреждения, и генеалогическое дерево человека, и федеративное устройство государства и многое другое. Для того, чтобы представлять эти иерархические структуры в программах, библиотека Swing имеет класс JTree - дерево.

Необходимые вспомогательные классы и интерфейсы находятся в пакете javax.swing.tree:

интерфейс TreeNode - узел дерева;
интерфейс MutableTreeNode - изменяемый узел дерева;
интерфейс TreeModel - модель дерева;
интерфейс TreeSelectionModel - модель выбора;
класс TreePath - массив узлов, первый из которых является корнем дерева.
класс DefaultMutableTreeNode - типовая реализация интерфейса MutableTreeNode;
класс DefaultTreeModel - типовая реализация интерфейса TreeModel;

класс DefaultTreeSelectionModel - типовая реализация интерфейса TreeSelectionModel;

11.1. Создание дерева

Узел дерева устроен так, что он может служить контейнером для других узлов. Это позволяет связывать разрозненные узлы в единую структуру, где дочерние узлы хранятся в родительском.

Пример "Построение дерева". Построим двухъярусное дерево, которое служит классификатором книг по программированию. На верхнем ярусе дерева будут расположены категории книг, на нижнем - собственно книги.

Вначале создадим корневой узел, а затем добавим в него дочерние узлы.

⇓

```
// Создать корневой узел
DefaultMutableTreeNode top =
  new DefaultMutableTreeNode("Корневой узел");

// Создать все другие узлы (код метода будет показан ниже)
createNodes(top);

// Создать дерево с узлом top в корне
JTree tree = new JTree(top);

// Поместить дерево в скроллер, чтоб не занимало много места
JScrollPane scroller = new JScrollPane(tree);
```

Код метода createNodes().

⇓

```
private void createNodes(DefaultMutableTreeNode top) {
  DefaultMutableTreeNode category = null;
  DefaultMutableTreeNode book = null;

  // Создать узел, который послужит категорией
  category = new DefaultMutableTreeNode("Книги по Java");
  top.add(category);

  // Создать узел, который послужит кногой
  book = new DefaultMutableTreeNode("Thinking in Java");
```

```
        category.add(book);

        // Еще одна книга
        book = new DefaultMutableTreeNode("Java 2");
        category.add(book);

        // Еще одна категория
        category = new DefaultMutableTreeNode("Книги по C++");
        top.add(category);
    }
```

В конструктор узла может быть передан любой пользовательский объект, не обязательно строка, как в примере.

11.2. Внешний вид дерева

Видом дерева можно управлять при помощи методов установки отдельных свойств дерева.

Показывать или нет корневой узел определяет метод

void setRootVisible(boolean show)

Показывать или нет линии между узлами:

метод putClientProperty("JTree.lineStyle", "Horizontal") - для горизонтальных линий;

метод putClientProperty("JTree.lineStyle", "Angled") - для угловых линий.

Метод putClientProperty( ) унаследован деревом от Component и служит для задания произвольных свойств компонента. Первый аргумент метода устанавливает ключ свойства, второй – значение свойства.

Дерево использует три иконки: для изображения листа, свернутой ветви и развернутой ветви. Стандартные иконки можно заменить. Для этого надо получить у дерева объект DefaultTreeCellRenderer и вызвать его методы: setLeafIcon() - для замены изображения листа, setOpenIcon() - для развернутой ветви, setClosedIcon() - для свернутой ветви.

Покажем это для листа дерева tree.

⇓

```
DefaultTreeCellRenderer renderer =
  (DefaultTreeCellRenderer) tree.getCellRenderer();
renderer.setLeafIcon(new ImageIcon("newLeaf.gif"));
```

Для более полного контроля над видом дерева необходимо унаследовать класс DefaultTreeCellRenderer и перекрыть его методы. Класс DefaultTreeCellRenderer, как и DefaultTableCellRenderer, является подклассом JLabel.

11.3. Реакция на выделение узла

Дело программиста - писать программы, поэтому он должен знать, как отследить действия пользователя, чтобы связать с ними определенный программный код.

Одно из обычных действий пользователя при работе с деревом - это выделение узла. Чтобы узнать об этом, надо в объекте дерева зарегистрировать слушателя события TreeSelectionEvent.

Пример "Реакция на выделение узла". Сделать так, чтобы при выделении узла дерева в окне сообщения появлялся путь к этому узлу. Путь к узлу - это последовательность узлов, которая начинается с корня и заканчивается заданным узлом.

Слушателем сделаем анонимный объект, который реализует интерфейс TreeSelectionListener. Интерфейс имеет единственный метод valueChanged().

⇓

```
tree.addTreeSelectionListener(
  new TreeSelectionListener() {
      public void valueChanged(TreeSelectionEvent e) {
          // Извлекаем путь к выделенному узлу
          TreePath path = e.getPath();
          // Отображаем путь в окне сообщения
          JOptionPane.showMessageDialog(null, path);
      }
  }
);
```

В дереве можно выделять единственный узел, группу соседних узлов, и группу произвольных узлов. Режим выделения устанавливается в модели выделения.

Получим модель выделения для дерева tree и установим режим, при котором можно выделять один узел.

⇓

```
tree.getSelectionModel().setSelectionMode
  (TreeSelectionModel.SINGLE_TREE_SELECTION);
```

11.4. Модель дерева

Модель дерева реализует интерфейс TreeModel. Методы интерфейса перечислены в таблице.

Метод	Описание
Objec getChild(Object parent, int index)	Возвращает сына узла parent с номером index. Нумерация ведется с 0
int getChildCount(Object parent)	Возвращает количество детей узла parent
int getIndexOfChild (Object parent, Object child)	Возвращает номер сына child узла parent
Object getRoot()	Возвращает корневой узел дерева
Boolean isLeaf(Object node)	Проверяет, является ли узел листом
void valueForPathChanged(TreePath path, Object newValue)	Вызывается, когда пользователь изменил значение узла. path - путь к узлу, newValue - новое значение узла
void addTreeModelListener (TreeModelListener listener)	Регистрирует слушателя listener для события TreeModelEvent, которое возникает, когда дерево изменилось
void removeTreeModelListener (TreeModelListener listener)	Удаляет слушателя listener события TreeModelEvent

В пакете javax.swing.tree имеется класс DefaultTreeModel, который реализует интерфейс TreeModel и добавляет ряд важных методов. Среди них:

- void insertNodeInto(MutableTreeNode newChild, MutableTreeNode parent, int index) - добавляет сыновний узел newChild к узлу parent в позицию index.
- void removeNodeFromParent(MutableTreeNode node) - удаляет узел node из дерева.

Если создать дерево без указания модели, то его моделью будет объект DefaultTreeModel.

11.5. Изменение состава дерева

Чтобы добавить или удалить узлы дерева, можно воспользоваться методами insert() и remove() класса DefaultMutableTreeNode, но лучше использовать методы аналогичного назначения, которые есть в модели дерева. Преимущество в том, что методы модели возбуждают событие TreeModelEvent, которое позволяет отследить изменения дерева, что необходимо, например, для автоматической перерисовки дерева.

Получить текущую модель можно методом класса JTree

TreeModel getModel().

Для изменения структуры дерева необходимо привести ссылку к действительному типу модели, т.к. именно там находятся нужные нам методы.

⇓

```
DefaultTreeModel model = (DefaultTreeModel)tree.getModel();
```

Пример "Изменение состава дерева". Изобразить дерево и две кнопки. По нажатию на одну кнопку к выбранному узлу добавляется сын. По нажатию на вторую кнопку выбранный узел удаляется.

Обработчик нажатия на кнопку добавления содержит код

⇓

```
// Получить ссылку на модель дерева
DefaultTreeModel model = (DefaultTreeModel) tree.getModel();

// Получить ссылку на выбранный узел
DefaultMutableTreeNode node =
  (DefaultMutableTreeNode)
tree.getSelectionPath().getLastPathComponent();

// Добавить новый узел в качестве первого сына выбранного узла
model.insertNodeInto(new DefaultMutableTreeNode("New Child"), node,
0);
```

Код удаления узла отличается от кода, приведенного выше, лишь последним оператором.

⇓

```
// Удалить выбранный узел
model.removeNodeFromParent(node);
```

11.6. Извещения об изменениях

Все изменения, происходящие в модели дерева, можно отследить при помощи слушателя TreeModelListener. Интерфейс TreeModelListener имеет следующие методы.

Метод	Описание
void treeNodesChanged (TreeModelEvent e)	Вызывается, если изменилось состояние узлов
void treeNodesInserted	Вызывается, если узлы добавлены в дерево

(TreeModelEvent e)	
void treeNodesRemoved (TreeModelEvent e)	Вызывается, если узлы удалены из дерева
void treeStructureChanged (TreeModelEvent e)	Вызывается, если изменилась структура дерева

Пример "Извещение о редактировании". По окончании редактирования узла новый текст узла записать прописными буквами.

По окончании редактирования узла, как, впрочем, и при других изменениях модели, возникает событие TreeModelEvent, которое может "услышать" объект TreeModelListener.

⇓

```
class MyTreeModelListener implements TreeModelListener {

    // Метод слушателя, вызываемый при изменении узла
    public void treeNodesChanged(TreeModelEvent e) {
        Object o = e.getTreePath().getLastPathComponent();
        DefaultMutableTreeNode node = (DefaultMutableTreeNode)o;
        // Если в событии есть список детей,  тогда измененный узел
        // является сыном узла node, полученного выше,
        // в противном случае измененный узел и есть node.
        try {
            int index = e.getChildIndices()[0];
            node = (DefaultMutableTreeNode) (node.getChildAt(index));
        } catch (NullPointerException exc) {}

        String s = (String)node.getUserObject().toUpperCase();
        node.setUserObject(s);
    }

    // Прочие методы интерфейса делаем пустыми
    public void treeNodesInserted(TreeModelEvent e) {}
    public void treeNodesRemoved(TreeModelEvent e) {}
    public void treeStructureChanged(TreeModelEvent e){}
}
```

Добавляем объект слушателя к модели дерева и, чтобы обеспечить возможность редактирования текста в узлах, вызываем метод JTree.setEditable() с параметром true.

⇓

```
tree.setEditable(true);
```

```
TreeModel model = tree.getModel();
model.addTreeModelListener(new MyTreeModelListener());
```

11.7. Задание

1. Задано символическое выражение "((a + b) * (c - d)) + (e - f) * g". Показать это же выражение в виде дерева, листьями которого являются имена переменных a,b,c,d,e,f,g, а в остальных узлах находятся арифметические операции.
2. Подставить собственные иконки вместо стандартных в дерево, отображающее символическое выражение.
3. Сделать так, чтобы при выборе узла в дереве выражения в заголовке формы отображалось бы подвыражение, соответствующее выбранному узлу.
4. Создать дерево, которое: а) отобразит структуру файлов и каталогов на дискете; б) будет иметь ту же структуру, что и главное меню формы.
5. Создать дерево, структуру которого может изменять пользователь при помощи трех пунктов меню: удалять узел, создавать брата выделенного узла, создавать сына выделенного узла.

12. АППЛЕТЫ

12.1. Что такое апплет

Апплет - это несамостоятельное приложение с некоторым ограничением в правах. Являясь частью web-страниц, апплеты расширяют их выразительные возможности, доводя их до уровня графического интерфейса пользователя, такого как в настольных приложениях.

Несамостоятельность апплета в том, что для его выполнения требуется работа другого приложения. Другим приложением может быть web-браузер, способный воспроизводить Java-апплеты, или специальная утилита JDK, предназначенная для просмотра апплетов в процессе их разработки - appletviewer.

Ограничение в правах выражается в том, что апплетам, попавшим на компьютер клиента с некоторого web-сервера:

- запрещен доступ к файловой системе клиента;
- запрещен доступ к большинству системных свойств;
- запрещен доступ к большинству сетевых операций;
- запрещен запуск процессов и загрузка динамических библиотек;

- запрещена печать, работа с буфером обмена, работа с системной очередью событий;
- запрещено использовать классы ClassLoader, Class, пакет java.sequrity.

Все эти ограничения вызваны безопасностью клиента и при желании могут быть ослаблены или устранены самим клиентом.

12.2. Устройство апплета

Формально апплетом является любой класс, который расширяет класс java.applet.Applet.

```
java.lang.Object
  |
  +-- java.awt.Componenet
        |
        +-- java.awt.Container
              |
              +-- java.awt.Panel
                    |
                    +-- java.applet.Applet
```

В отличие от обычного приложения апплет имеет не одну точку входа, метод main(), а несколько - методы start(), init() и другие, вызываемые исполняющей системой.

Методы апплета

void init() - вызывается однажды, после загрузки апплета в браузер или утилиту просмотра. Используется для инициализации, получения ресурсов и запуска потоков. В классе Applet не делает ничего.

void destroy() - вызывается перед выгрузкой апплета из браузера или утилиты просмотра. Используется для действий, противоположных инициализации.

void start() - вызывается всякий раз, когда апплет появляется в окне браузера (попадает в видимую область web-страницы).

void stop() - вызывается, когда апплет уходит из окна браузера. Методы start() и stop() часто используются в апплетах с анимацией и потоками.

String getAppletInfo() - вызывается для получения информации об апплете. Должен возвращать строку, подходящую для вывода в диалоговом окне About.

String[][] getParameterInfo(String parameterName) - вызывается для получения информации о параметре апплета с именем parameterName.

Image getImage(URL url) - загружает по сети изображение и возвращает объект Image.

AudioClip getAudioClip(URL url) - загружает по сети звуковой фрагмент и возвращает объект AudioClip.

void play(URL url) - если звуковой фрагмент найден по адресу url, он проигрывается.

String getParameter(String parameterName) - возвращает значение параметра parameterName из тэга <PARAM>.

URL getCodeBase() - возвращает URL загруженного класса апплета.

URL getDocumentBase() - возвращает URL web-страницы, которая вызывает апплет.

void showStatus(String message) - выводит сообщение message в строке состояния браузера или программы просмотра апплета.

AppletContext getAppletContext() - возвращает объект с интерфейсом AppletContext, который представляет окружение апплета. Дает доступ к другим апплетам в этом же документе.

Можно сказать, что методы init(), start(), pain(), stop(), destroy() составляют жизненный цикл апплета, впрочем, апплет не обязан перекрывать их все.

Пример "Hello". Предельно простой апплет.
⇓

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

public class SimpleApplet extends Applet {
  public void init() {
```

```
        add(new Label("Hello!"));
    }
}
```

Порядок отрисовки апплетов таков, как это определено для всех компонентов пакета AWT: при необходимости перерисовать апплет исполняющая система вызывает метод update(), который обновит фон и вызовет метод paint(). Если программист намерен форсировать перерисовку, он вызывает метод repaint(), сигнализирующий исполняющей системе, что апплет нуждается в перерисовке.

Действия, которые можно выполнить однократно, например, создание шрифта (Font) или создание графического образа (Image) следует выполнять в методе init().

Пример "Квадраты". Апплет, который рисует вложенные квадраты разного цвета.

⇓

```
import java.awt.*;
import java.applet.*;
public class MyApplet extends Applet {
  public void paint(Graphics g) {
      // Получить границы прямоугольника апплета
      java.awt.Rectangle r = getBounds();
      // Рисовать квадраты, начиная с наружного
      for (int d = r.height; d > 0; d -= 20) {
          // Выбрать цвет очередного квадрата
          g.setColor(new Color(0, 0, d % 255));
          // Нарисовать очередной квадрат
          g.fillRect((r.width - d) / 2, (r.height - d) / 2, d, d);
      }
  }
}
```

Апплет сам не управляет потоком выполнения, а лишь отвечает на вызовы своих методов. По этой причине методы апплета не должны выполняться долго, чтобы не парализовать браузер. Если апплету необходимо выполнить длительное действие (скачать файл, показать анимацию), он должен породить дочерний поток выполнения.

Пример "Квадраты-2". Апплет, который рисует вложенные квадраты, изменяя оттенки цвета с течением времени.

```
public class MyApplet extends Applet implements Runnable {
  boolean go;  // флаг потока анимации (true - поток работает)
  int green;     // зеленая составляющая цвета, которая меняется
  int timeout;  // пауза после перерисовки

  // Метод, выполняющий анимацию
  public void run() {
      while (go) {
          green = (green + 20) % 255;
          repaint();
          try {
              Thread.sleep(timeout);
          }
          catch (InterruptedException ex) { }
      }
  }

  // После загрузки зеленая составляющая цвета максимальна
  public void init() {
      green = 255;
      timeout = 500;
  }

  // Установить флаг и запустить поток анимации
  public void start() {
      go = true;
      (new Thread(this)).start();
  }

  // Сбросить флаг, чтобы поток анимации завершился
  public void stop() {
      go = false;
  }

  // Этот метод вызывает исполняющая система для перерисовки
  public void update(Graphics g) {
      java.awt.Rectangle r = getBounds();
      for (int d = r.height; d > 0; d -= 20) {
          // При установке цвета учитываем зеленую составляющую
          g.setColor(new Color(0, green, d % 255));
          g.fillRect((r.width - d) / 2, (r.height - d) / 2, d, d);
```

```
        }
    }
}
```

В данном примере был перекрыт метод update(), а не paint(). Базовый метод update() должен обновить фон и вызвать метод paint(). От обновления фона мы отказались, чтобы устранить вспышки фонового цвета во время перерисовки. От вызова paint() тоже отказались, т.к. весь код рисования поместили в метод update().

12.3. Выполнение апплетов в Web-браузере

Атрибуты апплета

Чтобы запустить апплет в браузере, на web-странице надо поместить тэг <APPLET> с необходимыми атрибутами:

- CODE – имя файла с откомпилированным кодом апплета. Имя файла должно соответствовать значению атрибута CODEBASE (см. ниже);
- WIDTH – ширина в пикселях прямоугольной области, которая будет выделена апплету при просмотре web-страницы в окне браузера;
- HEIGHT – высота области апплета в пикселях.

Кроме необходимых, имеются факультативные атрибуты:

- OBJECT – имя файла, содержащего сериализованный апплет. Перед сериализацией апплет должен быть инициализирован, поэтому выполнение такого апплета в браузере должно начинаться с метода start(), а не init(). Для апплета может быть задан только один из атрибутов: или CODE, или OBJECT.
- CODEBASE – содержит URL апплета, который указывает каталог с откомпилированным файлом апплета. Атрибут должен быть согласован с атрибутом CODE. Если атрибут отсутствует, используется URL текущего документа.
- ALT – текст, который должны показывать браузеры, не поддерживающие Java.
- NAME – имя апплета. Требуется для взаимодействия апплетов, расположенных на одной web-странице.

Пример. Тэг, который должен представлять апплет на web-странице.

На сервере файл MyApplet.class находится в том же каталоге, где и web-страница, поэтому параметр CODEBASE не требуется.

⇓

```
<APPLET CODE=MyApplet WIDTH=400 HEIGHT=300>
```

```
</APPLET >
```

Параметры апплета

Параметры играют для апплета ту же роль, что переменные окружения и аргументы командной строки для обычного приложения.

Параметры апплета задаются в документе с помощью тэгов <PARAM>, которые расположены внутри тэга <APPLET>. Каждый тэг <PARAM> имеет атрибуты:

- NAME – имя параметра;
- VALUE – значение параметра;

Апплет может получить значение параметров при помощи своего метода

String getParameter(String name).

Пример "Квадраты-3". Снабдить апплет двумя параметрами: цвет фона – backcolor и пауза после перерисовки – timeout.

Тэг <APPLET> будет содержать два параметра.

⇓

```
<APPLET CODE=MyApplet WIDTH=400 HEIGHT=300>
 <PARAM NAME="backcolor" VALUE="00FF00">
 <PARAM NAME="timeout" VALUE="100">
</APPLET >
```

В коде предыдущего примера изменится только метод init(). Инициализация полей будет происходить при помощи значений параметров апплета.

⇓

```
public void init() {
  green = 255;
  // Установить паузу
  String stimeout = getParameter("timeout");
   timeout = Integer.parseInt(stimeout);
  // Установить цвет фона
  String scolor = getParameter("bkColor");
  int icolor = Integer.parseInt(scolor, 16);
  setBackground(new Color(icolor));
}
```

12.4. Утилита appletviewer

В составе JDK имеется утилита appletviewer, которая позволяет просматривать апплеты без браузера. Она запускается из командной строки и получает в качестве параметра HTML-файл с тэгом <APPLET>.

⇓

```
>appletviewer MyApplet.html
```

Необходимый тэг <APPLET> можно поместить прямо в исходный код апплета в виде комментария.

⇓

```
// <applet code=MyApplet width=400 height=300>
// </applet>
```

Теперь утилиту можно запускать, передавая ей в командной строке файл исходного кода, а не html-документ, который в этом случае становится ненужным. Разумеется, необходимость в файле MyApplet.class остается.

12.5. Изображения и звуки

Изображения

Класс Applet имеет специальный метод getImage(), который получает URL файла изображения и возвращает объект Image. Последующая работа с изображением, связанным с объектом Image, будет происходить так, как это описано в разделе 9 "БИТОВЫЕ ИЗОБРАЖЕНИЯ" второй части книги.

Пример. Загрузка и вывод изображения, заданного в параметре апплета.

⇓

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

public class MyApplet extends Applet {

 private Image image;

 public void init() {
     // Инициализируем объект image
     image = getImage(getDocumentBase(), getParameter("image"));
 }
```

```
    public void paint(Graphics g) {
        // Рисуем изображение на поверхности апплета
        g.drawImage(image, 0, 0, this);
    }

    public void destroy() {
        // Высвобождаем ресурсы, занятые изображением
        image.flush();
    }
}
```

Звуки

Подобно методу getImage() метод getAudioClip() создает и возвращает объект AudioClip, связанный со звуковым файлом.

Пример. Расширим предыдущий пример, выполняя параллельно с загрузкой и выводом изображения загрузку и вывод звука.

⇓

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

public class MyApplet extends Applet {
  private Image image;
  private AudioClip sound;

  public void init() {
      image = getImage(getDocumentBase(), getParameter("image"));
      sound = getAudioClip(getDocumentBase(), getParameter("sound"));
  }

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

Для вызова апплета с web-страницы необходим тэг <APPLET>.

⇓

```
    <applet code = "javabasetasks.MyApplet.class" width = "400" height =
"300">
```

```
<param name=image value="sun.gif">
<param name=sound value="sun.ua" >
</applet>
```

12.6. JAR-файлы

Все файлы, необходимые для работы апплета (корневой класс, вспомогательные классы, изображения, звуки и т.п.), можно упаковать в архивный файл утилитой jar.

К тэгу <APPLET> надо будет добавить атрибут archive с указанием URL архивного файла.

Пример. Создать архивный файл myapplet.jar при помощи утилиты jar.

⇓

```
jar cf myapplet.jar *.class, sun.gif, sun.ua
```

Первый параметр командной строки – список ключей:

- c – создается новый файл;
- f - говорит, что имя jar-файла задается следующим параметром командной строки.

Добавить атрибут archive к тэгу <APPLET>

```
<APPLET archive=”soundmap.jar” …
```

12.7. Апплеты Swing

С появлением в JDK 1.2 библиотеки Swing появилась возможность создавать апплеты, расширяя класс javax.swing.JApplet. Такие апплеты поддерживают архитектуру компонентов Swing и для них справедливо все, что относится к контейнерам верхнего уровня библиотеки Swing (см. раздел "ФОРМЫ И ДИАЛОГИ"). Например, они имеют панель JRootPane с дочерними панелями и полосой меню, и добавление дочерних компонентов выполняется методом

```
applet.getContentPane().add(child);
```

и не

```
applet.add(child);
```

как в классе java.awt.Applet.

Пример "Hello-2". Простой Swing-апплет.

⇓

```
import javax.swing.*;
```

```
public class SimpleApplet extends JApplet {

  public void init() {
      this.getContentPane().add(new JLabel("Hello!"));
  }
}
```

Апплеты, расширяющие класс JApplet, по-разному вызываются из документов, предназначенных для браузеров компании Microsoft и Netscape. Более важно то, что для показа таких апплетов браузеры не располагают встроенными средствами.

Компания Sun предлагает эти средства бесплатно, но их необходимо скачать с сервера компании и включить в состав браузера, впрочем, все делается автоматически, пользователь лишь должен изъявить согласие и заплатить за Интернет.

Для вызова апплетов компания предлагает комбинацию тэгов, которая сработает в обоих браузерах и даже утилиту HTML-Convertor, которая автоматически переработает привычный тэг <APPLET> в равнозначную комбинацию тэгов.

Пример. Тег <APPLET> для вызова апплета "SimpleApplet" и результат его конверсии.

⇓

```
<APPLET CODE = SimpleApplet WIDTH = 400 HEIGHT = 200>
</APPLET>
```

```
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
WIDTH = 400 HEIGHT = 200
codebase="http://java.sun.com/products/plugin/1.3/jinstall-13-
win32.cab#Version=1,3,0,0">
<PARAM NAME = CODE VALUE = SimpleApplet >
<PARAM NAME="type" VALUE="application/x-java-
applet;version=1.3">
<PARAM NAME="scriptable" VALUE="false">
<COMMENT>
<EMBED type="application/x-java-applet;version=1.3"
CODE = SimpleApplet WIDTH = 400 HEIGHT = 200 scriptable=false
pluginspage="http://java.sun.com/products/plugin/1.3/plugin-
install.html"><NOEMBED></COMMENT>
```

```
</NOEMBED></EMBED>
</OBJECT>
```

12.8. Задание

1. Создать апплет-часы, который показывает текущее время. Номер часового пояса задавать в виде параметра апплета.
2. Расширить функции часов, превратив их в будильник. Время звонка (или нескольких звонков) задавать в параметрах апплета.
3. При помощи компонентов Swing сделать апплет, который показывает изображение. Имя файла с изображением задавать при помощи диалога открытия файла.

Часть 4. Разработка Internet-приложений

1. ПРОТОКОЛ HTTP

1.1. Идентификация ресурсов

Ресурсами в Internet считаются компьютеры, файлы, программы, службы.

Идентификация ресурсов выполняется с помощью URL (Uniform Resource Locator):

схема://адрес/путь_доступа

Примеры URL.

⇓

```
http://java.sun.com
http://tower.kture:8082/spages/index.jsp
```

Идентификатор может содержать не только имя ресурса, но и параметры, необходимые для его работы, если ресурс – программа.

Пример URL с параметрами.

⇓

```
http://tower.kture:8082/spages/index.jsp?name=bond&date=19%20oct
```

Параметры отделяются от пути знаком «?» и имеют форму «имя=значение». Друг от друга параметры отделяются амперсантом.

Строка параметров не должна содержать пробелов. Если в значениях параметров необходимы пробелы или др. запрещенные символы, они кодируются процентом и 16-м кодом символа. Так в вышеприведенном примере значением параметра date является строка "19 oct", которая закодирована как "19%20oct".

Допускаются параметры с одинаковыми именами.

1.2. Общие сведения об HTTP

HTTP это клиент-серверный протокол, реализующий модель запрос-ответ. Клиентом обычно является браузер, а сервером – Web-сервер.

Протокол HTTP/1.0 имеет несколько команд и не имеет состояний, т.е. работа ведется по схеме: установить соединение, передать запрос, получить ответ, разорвать соединение.

Пересылаемые данные состоят из заголовка сообщения (метаданных) и тела сообщения. Данные передаются в формате MIME (Multipurpose Internet Mail Extension). Структура передаваемых данных напоминает формат электронной почты, но HTTP допускает передачу двоичных данных, чего почта из-за 7-битного ограничения почтовых шлюзов не позволяет.

По умолчанию протокол HTTP использует 80-й порт, однако, могут использоваться и другие порты, явно указанные в URL.

HTTP не обязан использовать TCP/IP, он может функционировать с другими протоколами гарантированной доставки.

1.3. Запросы клиента

И запросы клиента, и ответы сервера имеют такой же формат, как текстовый ASCII-файл.

Запрос клиента состоит из строки запроса, заголовков и тела запроса.

Пример запроса на получение ресурса.

⇓

```
GET  /webapp/index.html  HTTP/1.0
Accept: text/html
Accept: image/gif
User-Agent: Mozilla
From: bond@hotmail.ru
[пустая строка]
```

В первой строке запроса указывается команда серверу, путь к ресурсу на сервере и версия протокола HTTP.

Замечание. Часть URL, которая содержит имя хоста и номер порта, используется протоколом нижнего уровня при установке соединения с сервером.

В остальных строках следуют поля заголовков, по одному полю в строке.

Заканчиваются заголовки пустой строкой, за которой может следовать тело запроса.

Команды

Согласно стандарту Web-сервер обязан выполнять только три команды: GET, HEAD и POST.

Команда GET

Осуществляет поиск ресурса на сервере и его передачу клиенту. Параметры запроса находятся в строке запроса.

Команда HEAD

В отличие от Get возвращает только метаданные. Применяется для проверки возможности доступа к ресурсу.

Команда POST

Используется для передачи данных от клиента к серверу. Параметры запроса находятся в теле запроса, формат параметров тот же, что в команде GET.

Запросы от HTML-форм передаются на сервер командой POST.

Пример. Написать Java-программу, которая выполнит роль HTTP-клиента.

Программа должна послать запрос и получить ответ от Web-сервера.

⇓

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

public class WebClient {

  public static void main(String[] args) throws Exception{

      // Задать параметры соединения
      final String HOST = "chief";
      final int PORT = 80;
      final String REQUEST = "GET /scripts/default.asp HTTP/1.0";

      // Установить соединение путем создания объекта java.net.Socket
      Socket client = new Socket(HOST, PORT);

      // Получить потоки для запроса (out) и ответа (in)
      PrintWriter out = new PrintWriter(client.getOutputStream(), true);
      Reader in = new InputStreamReader(client.getInputStream());
```

```
// Послать запрос без заголовков
out.println(REQUEST);
out.println();   // обязательная пустая строка
```

```
// Получить ответ и поместить его в строку response
// (действуем упрощенно, чтоб не загромождать пример)
char[] c = new char[10000];
int n  =  in.read(c);
String response = new String(c, 0, n);

// Закрыть соединение
client.close();

// Распечатать ответ для контроля
System.out.println(response);
  }
}
```

Как видно из этого примера, web-клиентом не обязан быть web-браузер.

1.4. Ответы сервера

Ответ сервера состоит из строки состояния, заголовков и запрашиваемых данных.

Пример минимально возможного ответа сервера.
⇓

```
HTTP/1.1 200 OK
Content-type text/plain
[пустая строка]
Hello World
```

Пример ответа сервера на запрос на получение ресурса.
⇓

```
HTTP/1.1 200 OK
Date: Wednesday, 20-Feb-02 23:00:00 GMT
MIME-version: 1.0
Content-type text/html
Content-length: 1230
[пустая строка]
```

```
<HTML><HEAD><TITLE>…
```

В первой строке ответа указывается версия протокола HTTP "HTTP/1.1", код состояния "200" и сообщение, соответствующее коду состояния "OK".

Далее следуют заголовки, пустая строка и, может быть, запрашиваемые данные.

Коды состояния

Код состояния представляет собой число, которое сопровождается коротким сообщением.

Имеются следующие категории кодов:

- Информационные 100-199
- Успешные 200-299
- Перенаправления 300-399
- Ошибка клиента 400-499
- Ошибка сервера 500-599

Поля заголовка

Заголовок HTTP-сообщения состоит из отдельных полей

Имя поля	Назначение поля
Accept	Клиент перечисляет возможные форматы данных при ответе на запрос
Allow	Сервер перечисляет поддерживаемые команды: GET, HEAD, …
Authorization	Применяют серверы, не разрешающие анонимный доступ
Content-Encoding	Указывает механизм кодирования
Content-Language	Указывает язык сообщения
Content-Length	Размер тела сообщения
Content-Type	Тип тела сообщения. Для HTML – “text/html”
Date	Дата и время создания сообщения
Expires	Дата и время, когда данные уже нельзя кэшировать. 0 – никогда нельзя
From	e-mail запрашивающего пользователя
If-Modified-Since	Дата и время для условной команды GET
Last-Modified	Дата последнего изменения данных
Link	Взаимосвязь данных с другим ресурсом

Location	URL прежнего местонахождения ресурса
MIME-Version	Номер версии используемого протокола MIME
Pragma	Многоцелевое поле для директив
Refer	URL, от которой получен запрашиваемый ресурс
User-Agent	Название и версия HTTP-клиента
WWW-Authenticate	Реализует доступ с аутентификацией

1.5. Протокол HTTP 1.1

Протокол HTTP 1.1 является развитием первой версии протокола. Основные отличия от HTTP 1.0 состоят в следующем:

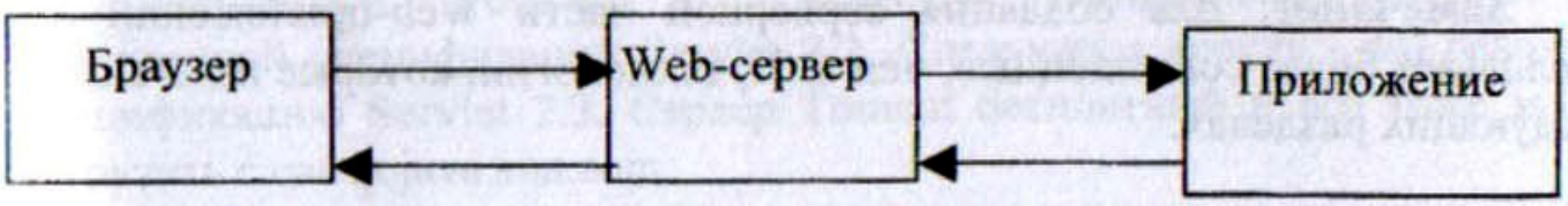
- создает постоянные соединения, которые могут использоваться для выполнения нескольких запросов, а не одного, как в HTTP 1.0;
- поддерживает сжатие данных;
- позволяет Web-серверу с одним IP-адресом иметь несколько доменных имен;
- допускает выборочную передачу участка файла, что позволяет продолжать передачу после потери и восстановления соединения.

Протокол HTTP 1.1 описан в документе RFC 2068.

1.6. Основы построения CGI-приложений

Интерфейс CGI, Common Gateway Interface, позволяет организовать обмен данными между браузером, Web-сервером и приложением, которое расширяет возможности Web-сервера (CGI-приложением).

В строке запроса указывается имя CGI-приложения или имя сценария, который CGI-приложение должно выполнить. В последнем случае приложение определяется по расширению файла сценария.



Сервер передает CGI-приложению заголовки через переменные среды, а тело запроса помещает в стандартный входной поток. Например, в переменной среды QUERY_STRING находится та часть URL, которая следует за первым знаком “?”.

Приложение выполняет обработку входных данных и выводит свой ответ в стандартный выходной поток. Как правило, в ответе задаются поля

Content-type, Location (если запрос переадресуется к другому ресурсу) и Status. Если статус не указан, сервер сам установит “200 OK”.

Пример. Написанное на C, CGI-приложение, посылающее ответ клиенту.

⇓

```
public class MyCgi {
  public static void main(String[] args) {
      // Установить заголовок Content-type
      System.out.println("Content-type: text/plain");
      // Отделить заголовки пустой строкой
      System.out.println();
      // Задать тело ответа
      System.out.print("Hello Internet!");
  }
}
```

Программу следует откомпилировать, построить исполнимый файл, например, mycgi.exe, и поместить виртуальный каталог web-сервера, предназначенный для исполняемых программ или скриптов, например, scripts.

Теперь в ответ на запрос

⇓

```
http://localhost/scripts/mycgi.exe
```

который можно сделать из web-браузера, мы увидим в окне браузера текст

"Hello Internet!"

Замечание. Для создания серверной части Web-приложений Java предлагает более совершенные, чем CGI, технологии, которые излагаются в следующих разделах.

1.7. Задание

1. Напишите клиентскую программу, которая посылает запрос серверу и показывает содержимое всех полей заголовка полученного сообщения в формате "имя=значение".
2. Создайте мини-браузер - графическое приложение, которое имеет два поля ввода: для текста запроса и для показа тела ответа в формате HTML.
3. Добавьте в предыдущее задание возможность показа сообщений в текстовом формате.

2. ЧТО ТАКОЕ СЕРВЛЕТ

Сервлет - это модуль, который расширяет возможности web-сервера подобно тому, как это делает CGI-приложение. По сравнению с CGI сервлеты имеют следующие преимущества:

- все сервлеты выполняются на одной виртуальной машине, а не в отдельном процессе, как CGI. В памяти может находиться много потоков - по одному для каждого запроса, но только одна копия сервлета.
- сервлет может легко сохранять состояние между запросами одного клиента, т.к. постоянно находится в памяти;
- сервлеты пишутся на том же языке, что и остальные части программы, т.е. на Java;
- сервлеты переносимы, т.к. пишутся на Java и являются частью платформы J2EE;
- сервлеты безопасны, т.к. пишутся на Java.
- сервлеты бесплатны, т.к. есть бесплатные web-серверы, которые их поддерживают.

API сервлетов не ориентировано на какой-либо протокол, но чаще всего сервлеты применяются для расширения web-серверов и используют протокол HTTP.

2.1. Установка программного обеспечения

Чтобы начать изучение сервлетов, нам понадобится API сервлетов и web-сервер, который их поддерживает.

Библиотека сервлетов входит в состав редакции J2EE или может быть загружена отдельно с сайта java.sun.com. Часто необходимое ПО ставится вместе с web-сервером.

Web-сервер Tomcat 3.x является официальной справочной реализацией спецификаций Servlet 2.2. Следующая версия - 4.0 реализует спецификацию Servlet 2.3. Сервер Tomcat бесплатный и его тоже можно загрузить с сайта java.sun.com.

Чтобы установить сервер на компьютере, надо скопировать необходимые файлы в отдельный каталог, например, C:/Tomcat и установить переменные среды:

TOMCAT_HOME =C:/Tomcat

JAVA_HOME=[Путь к классам JDK].

Настройки сервера хранятся в файле TOMCAT_HOME/conf/server.xml.

Здесь и далее под именем TOMCAT_HOME будем понимать каталог, в котором находятся файлы Tomcat.

В файле server.xml можно задать:

- номер порта (по умолчанию это - 8080);
- имя локального web-сервера (по умолчанию - localhost).

Запускается сервер командным файлом statrtup.bat из каталога TOMCAT_HOME/bin, останавливается командным файлом shutdown.bat, который находится в том же каталоге.

2.2. Пробный сервлет

Создадим сервлет, который возвращает web-страницу со словом "Hello".

Код сервлета

Построим сервлет в виде расширения класса javax.servlet.http.HttpServlet и поместим в файл HelloServlet.java.

⇓

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class HelloServlet extends HttpServlet {

  public void doGet(HttpServletRequest request,
  HttpServletResponse response) throws ServletException, IOException {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.println("<html> <body> Hello </body></html>");
      out.close();
  }
}
```

Компиляция сервлета

Компилируя сервлет, надо указать путь к библиотеке сервлетов. Например, если она расположена в файле C:\Tomcat\lib\servlet.jar, то команда на компиляцию будет такой

⇓

```
javac -classpath C:\Tomcat\lib\servlet.jar HelloServlet.java
```

Получившиеся после компиляции классы сервлетов надо размещать в каталоге

TOMCAT_HOME/webapps/ROOT/web-inf/classes,

где ROOT – каталог приложения или

TOMCAT_HOME/classes

или

TOMCAT_HOME/lib,

если классы содержатся в JAR файлах.

Запрос к сервлету

URL сервлета HelloServlet.class, который находится в каталоге TOMCAT_HOME/classes, будет

⇓

```
http://localhost:8080/servlet/HelloServlet
```

Слово servlet в начале пути не означает какой-то каталог, а является условным знаком для сервера.

Замечание. Сервер Tomcat кеширует классы сервлетов, поэтому после перекомпиляции сервлета надо остановить и снова запустить Tomcat, или изначально настроить его так, чтоб он не кешировал классы.

2.3. Структура сервлета

Чтобы быть сервлетом, способным расширять функции web-сервера, класс должен унаследовать класс javax.servlet.http.HttpServlet и переопределить метод doGet или doPost, в зависимости от того, хотим мы обрабатывать запросы GET или POST. Если оба запроса надо обрабатывать одинаково, можно переопределить метод doGet и вызывать его из doPost или поступить наоборот.

Оба метода имеют два аргумента - объекты
javax.servlet.http.HttpServletRequest и
javax.servlet.http.HttpServletResponse.

В классе HttpServletRequest есть методы, которые позволяют получить сведения о параметрах HTTP-запроса, заголовках запроса, имени хоста клиента и его IP-адресе.

Методы класса HttpServletResponse позволяют получить информацию, необходимую для ответа клиенту: код состояния, заголовки ответа

(Content-Type, Set-Cookie,...) и, что самое важное, позволяет получить поток PrintWriter для передачи тела ответа клиенту. Если надо передавать байты, а не символы, получают поток OutputStream. Закрытие потока означает для сервера, что ответ готов и его можно отсылать.

Класс HttpServlet обязаны наследовать только те сервлеты, которые расширяют возможности web-сервера. Сервлеты могут расширять электронную почту, FTP и другие типы серверов. Эти сервлеты должны наследовать некоторый класс Xxx, наследующий класс GenericServlet (Xxx – «брат» HttpServlet).

Пример. Сервлет для сложения чисел. В параметрах запроса GET сервлет получает два числа, складывает их и возвращает результат.

⇓

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class PlusServlet extends HttpServlet {
  static final private String CONTENT_TYPE =
       "text/html; charset=windows-1251";

public void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {

    // Получить параметры запроса
    String s1 = request.getParameter("x1");
    String s2 = request.getParameter("x2");

    // Вычислить сумму или диагностировать ошибку
    try {
        float sum = Float.parseFloat(s1) + Float.parseFloat(s2);
        sumstr = String.valueOf(sum);
    }
    catch (Exception ex) {
        sumstr = "ERROR";
    }

    // Сформировать ответ в виде HTML-страницы
    response.setContentType(CONTENT_TYPE);
    PrintWriter out = response.getWriter();
    out.println("<html><body>" + sumstr + "</body></html>");
```

```
        out.close();
    }
}
```

Хотя HTML самый распространенный тип содержания ответа (text/html), сервлет может отсылать и простой текст (text/plain), и изображения (image/gif), и другие типы содержания. Тип содержания устанавливается заголовком Content-Type.

Заголовки надо выводить перед содержанием, поскольку в HTML-сообщении они предшествуют содержанию. Порядок вывода заголовков произвольный.

2.4. Жизненный цикл сервлета

Каждый сервлет проходит следующие фазы существования:

1. Загружается контейнером сервлетов.
2. Выполняет запросы клиента.
3. Выгружается контейнером.

Сервлет загружается по первому запросу к нему или сразу после загрузки сервера, в зависимости от того, как он зарегистрирован на сервере. Сразу после загрузки сервер вызывает метод сервлета init().

Если после инициализации сервлет не работоспособен, необходимо выбросить исключение UnavailableException, а не вызывать метод System.exit(0).

Каждый запрос клиента заставляет сервер создавать отдельный поток и в нем вызывать метод service(). Если одновременно выполняется несколько запросов, то service() вызывается из нескольких потоков одновременно. Метод service() вызывает один из методов doXxx(), в зависимости от типа запроса.

Непосредственно перед выгрузкой сервлета сервер вызывает метод сервлета destroy(). Оба метода init() и destroy() вызываются лишь один раз в жизни сервлета.

Метод init()

Во время инициализации сервлет может получить от сервера именованные параметры.

Существуют 2 версии метода Init - с параметром класса ServletConfig или без параметров.

Тем не менее, параметры инициализации доступны из обоих методов, т.к. в классе GenericServlet есть метод getInitParameter(), который получает объект ServletConfig и вызывает его метод getInitParameter().

Способ задания параметров сервлета зависит от типа сервера. В Tomcat они задаются в формате XML в файле WEB-INF/web-xml.

Пример псевдокода соединения с базой данных в методе init().

⇓

```
public class DBServlet ... {

  Connection connection = null;

  public void init() throws ServletException {
      // Open a database connection to prepare for requests
      try {
          databaseUrl = getInitParameter("databaseUrl");
          // get user and password parameters the same way
           ...
          connection = DriverManager.getConnection  (databaseUrl,
                       user, password);
      } catch(Exception e) {
          throw new UnavailableException (this,
                "Could not open a connection to the database");
      }
  }
  ...
}
```

Метод service()

Метод service определяет тип запроса и вызывает один из методов doXxx(). Кроме doGet() и doPost(), в классе HttpServlet имеет методы doPut(), doDelete(), doTrace(). Метода doHead() нет, т.к. для ответа на запрос HEAD используются строка состояния и заголовок ответа, выдаваемого методом doGet().

Реализация всех методов в классе HttpServlet возвращает ответ с кодом состояния «400 Bad Request».

Замечание. Расширяя класс HttpServlet, следует перекрывать именно методы doXxx(), а не метод service().

Потоковая безопасность

Сервлет может быть затребован многими клиентами одновременно. Если сервлет имеет доступ к разделяемым ресурсам, он может применить блокировки (см. многопоточность) или же запретить одновременный доступ к себе нескольких клиентов. Последнее достигается реализацией сервлетом интерфейса-метки **SingleThreadModel.**

⇓

```
public class PlusServlet extends HttpServlet implements
SingleThreadModel
{...}
```

Метод destroy()

Метод destroy() предназначен для выполнения действий, противоположных тем, что выполнялись в методе init().

Пример псевдокода рассоединения с базой данных в методе destroy().

⇓

```
public class DBServlet ... {
  Connection connection = null;

  ... // the init method

  public void destroy() {
      // Close the connection and allow it to be garbage collected
      connection.close();
  }
}
```

Сервлет выгружается из памяти, когда истек период тайм-аута, т.е. запросов от клиентов не было достаточно долго или по команде контейнера.

К моменту выполнения метода destroy() обработка всех запросов должна быть завершена. Если при обработке запроса выполняется длительная операция, в сервлете необходимы специальные средства (счетчик выполняемых запросов), которые позволят корректно выгрузить сервлет. Для корректного прекращения длительной операции применяется флаг, который проверяется в цикле операции.

Замечание. Нельзя полагаться на метод destroy() в вопросах сохранения информации из памяти на диске. Сервер может рухнуть раньше и метод destroy() никогда не выполнится.

2.5. Отладка сервлетов

При отладке сервлетов полезны следующие мероприятия.

Проверяйте генерируемый код HTML.

Проверить правильность HTML-кода можно визуально, при помощи настольных приложений или при помощи онлайновых сервисов Worlg Wide Web Consortium http://validator.w3.org/ или Web Design Group Help http://www.htmlhelp.com/tools/validator/.

Чтобы проверка была возможной, сгенерированная страница должна иметь тэг <!DOCTYPE …>

Если сервлет генерирует страницу методом GET, то для проверки можно использовать URL сервлета.

Возвращайте клиенту страницу ошибок.

Сервлет должен описать ошибку и вернуть описание на отдельной странице или на стандартной странице методом объекта HttpServletResponse sendError(). Например, если сервлет PlusServlet получил не число, можно вызвать метод

⇓

```
response.sendError(498, "Not a number");
```

В ответ на этот вызов сервер вышлет стандартную страницу ошибок с указанием кода состояния (1-й параметр метода) и сообщением (2-й параметр метода).

Запускайте сервер из командной строки.

Если web-сервер запущен как отдельное приложение, а не как служба, что является обычной практикой, его можно останавливать и снова запускать при необходимости. Кроме того, можно в консольном окне видеть сообщения, которые выдает сервлет в стандартный поток и поток ошибок методами System.out.println(); и System.err.println().

Используйте файл регистрации.

В классе HttpServlet имеется метод log(), который выводит сообщения в файл регистрации, находящийся на сервере. Для Tomcat это TOMCAT_HOME\logs\servlet.log. Файл регистрации очищается с каждым запуском сервера.

Проверяйте данные запроса.

Прежде чем искать ошибку в сервлете, проверьте правильность запроса.

2.6. Пакет javax.servlet

Базовая часть библиотеки сервлетов находится в пакете javax.servlet. Пакет содержит следующие интерфейсы.

- **RequestDispatcher** – объект, который получает запросы пользователя и адресует их некоторому ресурсу (сервлету, HTML-странице, файлу) на сервере. Объекты создаются контейнером сервлетов (той частью web-сервера, которая отвечает за работу с сервлетами) и служат оболочками для любого ресурса на сервере.
- **Servlet** – набор методов абстрактного сервлета. Реализуется классом GenericServlet, базовым для класса HttpServlet.
- **ServletConfig** – объект создается сервером и содержит параметры инициализации сервлета.
- **ServletContext** – набор методов для связи между сервлетом и web-сервером, например, получение MIME-типа файла, запись в log-файл, получение диспетчера запроса.
- **ServletRequest** – методы для получения данных запроса.
- **ServletResponse** – методы, обеспечивающие отсылку информации клиенту.
- **SingleThreadModel** – интерфейс-метка для потокобезопасных сервлетов.

Пакет содержит следующие классы:

- **GenericServlet** – независимый от протокола сервлет;
- **ServletInputStream** – поток для чтения двоичных данных из тела запроса клиента, включает метод readLine для чтения одной строки;
- **ServletOutputStream** – поток для отсылки двоичных данных клиенту.

Методы интерфейса Servlet

Метод	Описание
void init(ServletConfig config)	Вызывается контейнером сервлетов после загрузки сервлета. Может выбрасывать исключение ServletException
void destroy()	Вызывается контейнером сервлетов, перед выгрузкой сервлета

void service(ServletRequest request, ServletResponse response)	Вызывается контейнером сервлетов, при необходимости обработать запрос. Выбрасывает исключение ServletException
ServletConfig getServletConfig()	Возвращает параметры сервлета в объекте ServletConfig
String getServletInfo()	Возвращает информацию типа "About"

Методы интерфейса ServletRequest

Метод	Описание
Object getAttribute(String name)	Возвращает атрибут запроса по имени атрибута
String getParameter(String name)	Возвращает параметр запроса по имени параметра
ServletInputStream getInputStream()	Возвращает тело запроса как поток ServletInputStream
BufferedReader getReader()	Возвращает тело запроса как поток BufferedReader
String getRemoteAddr()	Возвращает IP-адрес клиента, приславшего запрос

Интерфейс ServletRequest расширяется интерфейсом HttpServletRequest.

Методы интерфейса ServletResponse

Метод	Описание
ServletOutputStream getOutputStream()	Возвращает поток для отсылки клиенту двоичных данных
PrintWriter getWriter()	Возвращает поток для отсылки клиенту текста
void reset()	Очищает буфер потока отсылки
void setContentType(String type)	Устанавливает тип содержания. Должен предшествовать отсылке данных

Интерфейс ServletResponse расширяется интерфейсом HttpServletResponse.

2.7. Задание

1. Напишите сервлет, который в ответ на запрос к нему возвращает приветствие, зависящее от времени суток на сервере: "Доброе утро!", "Добрый день!", "Добрый вечер!", "Доброй ночи!".

2. На сервере имеется телефонный справочник – текстовый файл с фамилиями и номерами телефонов. Разработайте сервлет, который получает в запросе GET фамилию и возвращает все номера телефонов абонентов, которые носят эту фамилию.

3. Дополните сервлет возможностью добавлять и удалять абонентов из файла-справочника. Позаботьтесь о потоковой безопасности операций над справочником.

3. ЗАПРОСЫ И ОТВЕТЫ

3.1. Анализ запроса

HTTP-запрос инкапсулируется в объекте request, который передается в методы сервлета doGet(), doPost() и им подобные. Доступ к информации запроса возможен через методы объекта request, т.е. через методы интерфейса HttpServletRequest.

Параметры запроса

Параметры могут присутствовать в запросах GET и в запросах POST.

В команде GET они содержатся прмо в строке запроса, отделенные от URL вопросительным знаком, например

⇓

```
GET  /webapp/index.html?par1=val1&par1=val2  HTTP/1.0
```

В команде POST они находятся в теле запроса

⇓

```
POST  /webapp/index.html  HTTP/1.0
[загловки запроса]

par1=val1&par1=val2
```

Чтобы получить параметры запроса, их надо извлечь из строки запроса или из тела. Разбор строки параметров осложнен тем, что в ней могу быть параметры с одинаковым именем и параметры с пустым значением. Кроме того, некоторые символы в составе значения представлены в виде знака процента и 16-чного кода за ним. Имена параметров чувствительны к регистру.

Пример. В строке 4 параматера, два из них имеют одинаковое имя, значение последнего параметра - "v a l 3".

```
par1=val1&par1=val2&par2=&par3=v%20a%20l%203
```

Интерфейс ServletRequest имеет несколько методов для работы с параметрами.

Метод	Описание
String getParameter(String name)	Возвращает значение параметра с именем name или null, если параметра в запросе нет
String[] getParameterValues (String name)	Возвращает все значения параметра с именем name
Enumeration getParameterNames()	Возвращает коллекцию имен параметров из строки запроса
Map getParameterMap()	Возвращает словарь параметров из строки запроса
String getQueryString()	Возвращает сырую строку запроса

Тело запроса

Тело запроса может присутствовать в командах POST, PUT и DELETE. Из сервлета доступ к телу можно получить методами интерфейса ServletRequest.

Метод	Описание
BufferedReader **getReader()**	Возвращает поток для чтения текста
ServletInputStream **getInputStream()**	Возвращает поток для чтения двоичных данных

Заголовки запроса

Поскольку заголовки являются специфической чертой протокола HTTP, методы работы с ними содержатся в интерфейсе HttpServletRequest, расширяющем интерфейсе ServletRequest.

Метод	Описание
String getHeader(String name)	Возвращает содержание заголовка с именем name или null, если заголовка нет. Имена заголовков не чувствительны к регистру
Enumeration getHeaders(String name)	Возвращает содержание всех заголовков с именем name. Одноименных заголовков в запросе может быть несколько

Для чтения часто встречающихся заголовков применяют специальные методы, смысл которых понятен из их названий: getContentType(), getContentLength(), getAuthType(), getRemoteUser().

Строка запроса

Информацию, которая содержится в первой строке запроса, можно получить методами:

- String getMethod() – команда запроса (GET, POST, HEAD, ...);
- String getRequestURI() – путь доступа к ресурсу, например, "/webapp/index.html";
- String getProtocol() – версия протокола, обычно “HTTP/1.0” или “HTTP/1.1”.

3.2. Синтез ответа

HTTP-ответ состоит из кода состояния, заголовков и тела ответа (последнее не обязательно). Для синтеза ответа сервлет использует объект HttpServletResponse response, который передается в методы doGet(), doPost() и пр. как параметр.

Коды состояния

В первой строке ответа сервер посылает код состояния. Сервлет может установить код методом response.setStatus().

В классе HttpServletResponce определены числовые константы SC_... для кодов состояния, приведем некоторые из них.

100-199 - информационные

Клиент должен ответить на сообщение каким-нибудь действием.

SC_CONTINUE = 100. Если клиент спрашивает, можно ли получить документ в следующем запросе (заголовок запроса Expect = 100), сервер сообщает, что можно.

200-299 - успешные

Запрос успешно выполнен.

SC_OK = 200. Состояние, устанавливаемое сервлетом по умолчанию.

SC_CREATED = 201. Сервер создал новый документ. Заголовок ответа Location = URI нового документа.

SC_ACCEPTED = 202. Запрос выполняется, но его обработка еще не окончена.

SC_NO_CONTENT = 204. Браузер должен отображать предыдущий документ. Может пригодиться, если клиент периодически запрашивает новый документ, а он не изменился или не доступен.

300-399 – перенаправления

Для файлов, которые были перемещены. Ответ содержит заголовок Location с новым URL.

SC_MULTIPLE_CHOCES = 300. Запрашиваемый документ можно найти в нескольких местах. Предпочтительное для сервера место возвращается в заголовке ответа Location.

SC_MUVED_PERMANENTLY = 301. Новый URL документа возвращается в заголовке Location. Браузеры должны автоматически следовать по этой ссылке.

SC_MUVED_TEMPORARILY = 302. Новый URL документа возвращается в заголовке Location. Браузеры должны автоматически следовать по этой ссылке. В объекте response существует специальный метод sendRedirect(URL), который формирует ответ с кодом 302 и нужным заголовком Location.

SC_NOT_MODIFIED = 304. Посылается в ответ на условный запрос (запрос с заголовоком If-Modified-Since), если документ не был изменен после указанной даты.

400-499 - ошибка клиента

SC_BAD_REQUEST = 400. Неверный синтаксис запроса. Неперекрытые методы сервлета doXxx() отвечают этим кодом.

SC_NOT_FOUND = 404. Ресурс с данным URL не найден. В объекте response существует специальный метод sendError(«message»), по которому сервер автоматически генерирует страницу ошибок и отсылает в ответе с этим состоянием. Браузер Internet Explorer 5 может игнорировать эту страницу и выводить свою, но эту настройку можно изменить в меню Tools/Internet options/Advanced/Show friendly HTTP error message.

500-599 - ошибка сервера

SC_INTERNAL_SERVER_ERROR = 500. Обобщенный код ошибки. Устанавливается, если сервлет вернул неправильные заголовки или разрушился.

3.3. Заголовки ответа

Заголовки ответа устанавливаются то того, как первый байт документа послан в выходной поток.

В интерфейсе HttpServletResponse имеются общие методы для установки заголовков:

- void setHeader(String name, String value) – заголовок-строка;
- void setDateHeader(String name, Date value) – заголовок-дата;
- void setIntHeader(String name, Int value) – заголовок-целое число.

Протокол HTTP допускает несколько вхождений одного заголовка, например, Accept или Set-Cookie.

В версии сервлетов 2.1 методы, названия которых начинаются с "set", всегда добавляют новые заголовки, а в версии сервлетов 2.2 - замещают старые, если они были. Добавляют заголовки методы, названия которых начинаются с "add".

Узнать, задан ли уже какой-то заголовок, можно методом

boolean containsHeader(String name).

Для часто используемых заголовков в классе HttpServletResponse есть специальные методы:

- void setContentType(String type);
- void setContentLength(int length);
- void addCookie(Cookie cookie);
- void sendRedirect(String location).

Заголовок ContentType

Заголовок ContentType задает тип MIME (Multipurpose Internet Mail Extension) документа ответа. Этот тип необходим браузеру, чтобы правильно показать полученный документ.

Официально зарегистрированные типы имеют вид "тип/подтип", а не зарегистрированные - "тип/x-подтип".

Примеры MIME-типов:
text/plain - тип, задаваемый сервлетами по умолчанию;
text/html - web-страница;
image/gif - изображение в формате gif;
audio/basic - аудиофайл формата au или snd;
application/x-gzip - архив gzip;
application/x-java-archive - файл jar;
application/msword- документ MS Word;
application/vnd.ms-excel - документ MS Excel.

Строка описания типа может включать в себя и способ кодирования. Чтобы браузер при показе документа не пытался угадать правильную кодировку, ее можно указать явно.

⇓

```
responcse.setContentType("text/html; charset=windows-1251");
```

Заголовок Expires

Этот заголовок устанавливает время в минутах, по истечении которого документ будет признан устаревшим. Если значение заголовка равно -1, документ будет считаться устаревшим немедленно и не будет кешироваться у клиента.

⇓

```
response.setIntHeader("Expires", -1);
```

3.4. Тело ответа

После того как заголовки ответа установлены, сервлет может сформировать тело ответа, выводя необходимую информацию в поток данных.

Если информация символьная, следует получить ссылку на символьный поток методом response.getWriter(). Если информация двоичная, следует получить ссылку на байтовый поток методом response.getOutputStream()

Замечание. Ответ сервлета не обязан быть web-страницей. Сервлет может возвращать двоичные данные, например, изображение.

Пример "Изображение в формате GIF". В ответ на запрос создать и отослать изображение.

Для построения изображения можно создать объект окна и связать его с ресурсами операционной системы методом addNotify(), который обычно никогда не вызывается напрямую. Затем в окне можно построить изображение.

Чтобы вывести изображение в формате gif, можно воспользоваться классом GifEncoder, доступным на www.acme.com.java/.

⇓

```
public void doGet(HttpServletRequest request,
  HttpServletResponse response) throws ServletException, IOException {

  // Установить заголовок и получить поток
  response.setContentType("image/gif");
  OutputStream out = response.getOutputStream();

  // Сформировать изображение
  Frame f = new Frame();
  f.addNotify();
  int width = 400;
  int heigth = 300;
  Image img = f.createImage(width, heigth);
  Graphics g = img.getGraphics();
  g.fillRect(0,0,400, 300);
  g.drawString("Hello", 10, 10);

  // Отослать изображение в формате gif
  try {
      new GifEncoder(img, out).encode();
  } catch (IOException ex) {
      // Message Error
  }
}
```

Пример "Лист Excel". В ответ на запрос создать и отослать лист электронной таблицы Excel.

Для создания листа Excel достаточно установить заголовок ответа Content-Type в "application/vnd.ms-excel" и отослать тэг <TABLE> со всем содержимым.

⇓

```
public void doGet(HttpServletRequest request,
```

```
HttpServletResponse response) throws ServletException, IOException {
response.setContentType("application/vnd.ms-excel;
    charset=windows-1251");
PrintWriter out = response.getWriter();
out.println("<HTML> <BODY> <TABLE>" +
"<TR><TD>Клетка 1,1</TD> <TD>Клетка 1,2</TD></TR>" +
"<TR> <TD>Клетка 2,1</TD> <TD>Клетка 2,2</TD> </TR>" +
"</TBODY> </TABLE> </BODY></HTML>");
out.close();
}
```

Замечание. Тип содержания "application/vnd.ms-excel" не является стандартным, поэтому не все браузеры понимают его.

3.5. Передача сжатых web-страниц

Некоторые браузеры поддерживают отображение сжатых web-страниц в формате gzip. Если браузер имеет такую возможность, он сообщает об этом в заголовке запроса Accept-Encoding.

JDK поддерживает формат сжатия gzip, поэтому для отсылки сжатой страницы достаточно установить заголовок ответа Content-Encoding и направить вывод сервлета в поток GZIPOutputStream.

Пример фрагмента метода doGet(), который это делает.

⇓

```
...
String encodings = request.getHeader("Accept-Encoding");
PrintWriter out;

if (encodings != null && encodings.indexOf("gzip") != -1) {
  OutputStream out1 = response.getOutputStream();
  out = new PrintWriter(new GZIPOutputStream(out1), false);
  response.setHeader("Content-Encoding", "gzip");
}
...
```

3.6. Данные приложения

Один экземпляр сервлета выполняет запросы различных клиентов. Из этого следует, что разные клиенты могут разделять одни и те же данные, если они сохраняются в полях сервлета.

Другим местом, подходящим для хранения разделяемых данных является объект ServletContext, который создается сервером и является

общим для всех сервлетов. Интерфейс ServletContext имеет методы getAttribute() и setAttribute(), которые позволяют сохранить любые данные, ассоциированные с ключом.

Метод	Описание
void setAttribute(String key, Object object)	Связывает object с ключом key и сохраняет ссылку в объекте ServletContext
Object getAttribute(String key)	Получает ссылку на атрибут, по заданному ключу key

О данных, сохраняемых таким образом, говорят, что их область видимости - приложение.

3.7. Задание

1. Разработайте сервлет, который получает два числа и знак одной из 4-х арифметических операций: "+", "-", "*", "/". Сервлет должен выполнить над числами указанное действие и вернуть результат. Для формирования запроса к сервлету используйте HTML-форму с двумя текстовыми полями и списком из 4-х операций.

2. На сервере имеется каталог с несколькими текстовыми файлами. Создайте сервлет, который получает имя файла и возвращает его содержимое в виде простого текста (тип содержания ответа – text/plain).

3. Сделайте так, чтобы сервлет из предыдущего задания возвращал сжатый файл, в том случае, если браузер способен такие файлы принимать.

4. СЕАНСЫ И ПЕРЕАДРЕСАЦИЯ

4.1. Класс Cookie

Cookie (по-русски "пирожок") - это порция данных, которая отсылается клиенту и возвращается к серверу вместе с запросом от клиента. Чаще всего cookie используются для идентификации клиента и шифрования обмена данными между клиентом и сервером. Сервер может использовать cookie для адаптации своего поведения к конкретному клиенту и, в частности, для направленной рекламы.

Cookie содержит набор атрибутов, таких как время жизни, каталог на сервере в который попадет присланный клиентом cookie, доменное имя, комментарий, номер версии и т.п. Объем cookie может доходить до 4 К.

Cookie не представляют угрозы безопасности клиента (занесение вирусов или атака на систему), т.к. не интерпретируются и не выполняются. Обычно браузеры допускают не более 20 cookies на сайт и не более 300 всего, поэтому cookies не могут заполнить собой диск.

Cookie представляют угрозу конфиденциальности, поскольку серверы получают через них некоторые данные о пользователе, которые могут быть у них, серверов, украдены, поэтому клиент может отказаться от приема cookie.

В библиотеке сервлетов имеется класс javax.servlet.http.Cookie, конструктор которого содержит имя объекта и его значение в виде строки. Объекты, представляющие cookie, необходимо присоединить к ответу перед его отсылкой.

⇓

```
Cookie cookie = new Cookie("Dog", "Tuzik");
responce.addCookie(cookie);
```

Замечание. Ни имя, ни значение cookie не должны содержать пробелы, а также символы: [ ] = , " / ? @ ^ : ;

Cookie, присланные клиентом, извлекаются из объекта ответа для анализа.

⇓

```
Cookie[] cookies = request.getCookies();
if (cookies[0].getValue()) …
```

4.2. Организация сеансов

Как известно, протокол HTTP не сохраняет состояния между запросами клиента к серверу. Так, например, при игре в шахматы он способен передавать ходы, а помнить позицию не способен. Чтобы сохранять между запросами информацию, относящуюся к конкретному клиенту, web-сервер имеет механизм, называемый сеансами

Сеанс позволяет сохранять состояние клиента, который посылает на сервер несколько запросов. Состояние сохраняется в виде набора объектов произвольного типа. Все сервлеты разделяют информацию о состоянии клиента, т.е. сеанс.

Для чтения или записи информации сеанса необходимо получить у контейнера сервлетов объект HttpSession и прочитать или записать его атрибуты.

Чтобы отличать клиентов одного web-приложения друг от друга, их необходимо идентифицировать. Принципы идентификации клиента могут быть различными (cookie, модификация запроса, скрытые поля формы, IP-адрес клиента), но в библиотеке сервлетов это делается при помощи cookie или модификации URL.

Интерфейс HttpSession

Методы	Описание
Object getAttribute(String name)	Возвращает атрибут по его имени или null, если атрибута не существует
Enumeration getAttributeNames()	Возвращает список имен всех атрибутов
long getCreationTime()	Абсолютное время создания сеанса в мсек, считая с 1 января 1970 г.
String getId()	Уникальный идентификатор сеанса
long getLastAccessedTime()	Время последнего запроса в данном сеансе
int getMaxInactiveInterval()	Максимальный интервал времени, прошедшего между запросами в этом сеансе
void invalidate()	Закрывает сеанс и освобождает связанные с ним объекты
boolean isNew()	Возвращает истину, если сеанс новый или клиент отказался от него (отказался от cookie)
void removeAttribute(String name)	Удаляет объект атрибута с именем name из сеанса
void setAttribute(String name, Object value)	Добавляет атрибут с именем name в сеанс
void setMaxInactiveInterval(int interval)	Устанавливает время, через которое сеанс закрывается, если нет запросов (тайм-аут)

Чтобы избежать конфликта имен атрибутов, создаваемых в разных сервлетах, к имени атрибута рекомендуется добавлять полное имя сервлета.

Чтобы корректно работать с сеансом, необходимо получать объект HttpSession раньше, чем объект Writer для вывода ответа.

Пример. Создать сервлет, который бы считал запросы от каждого клиента отдельно.

Поскольку каждому клиенту соответствует отдельный сеанс, установим в сеансе атрибут, который будет хранить целое число и станем увеличивать это число на 1 при каждом запросе.

⇓

```
public void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {

    // Создать или получить объект сеанса
    HttpSession session = request.getSession(true);

    // Создать или получить счетчик запросов для данного сеанса
    Integer count  =
        (Integer)session.getAttribute("myPackage.MyServlet.count");

    // Увеличить счетчик и вновь сохранить
    int i = (count == null) ? 0 : count.intValue() + 1;

    session.putAttribute("myPacket.MyServlret.count", new Integer(i));
    . . .
}
```

Параметр true метода getSession() говорит, что если сеанса еще не существует, то будет создан новый сеанс.

Сеанс считается новым, если клиент еще не знает о нем или отказывается от участия в сеансе (путем соответствующей настройки браузера). Для нового сеанса метод isNew() возвращает значение true.

Сеанс может быть уничтожен программно или контейнером сервлетов по истечении обусловленного периода тайм-аута (30 мин. по умолчанию). Для принудительного удаления сеанса вызывают метод **invalidate()**.

Модификация URL

Альтернативой cookies для поддержки сеансов является добавление параметра с идентификатором сеанса во всякую ссылку, которую клиент может активизировать на web-странице. Контейнер сервлетов анализирует модифицированный URL, извлекает оттуда идентификатор сеанса и

сравнивает с имеющимися у него идентификаторами. При этом важно, чтобы все URL, доступные клиенту, были оснащены идентификатором сеанса.

Для модификации URL в сервлете используется метод объекта response

String encodeURL(String url)

Пример модификации адреса.

Ссылка до модификации

⇓

```
out.println( "<A HREF='/bookstore/bookdetails?bookId=" + bookId +
"'>");
```

Ссылка после модификации

⇓

```
out.println( "<A HREF='" +
response.encodeURL  ("/bookstore/bookdetails?bookId=" + bookId) +
"'>");
```

Метод encodeURL будет определять, используется ли в данное время перезапись URL, и добавлять в URL идентификатор сеанса при необходимости.

Еще одно место, где используется URL вашего собственного сайта, это вызов метода объекта responce sendRedirect(), когда нужно выполнить переадресацию запроса (о переадресации см. раздел 5). Чтобы модифицировать этот адрес, нужно предварительно закодировать адрес методом encodeRedirectURL(), а потом вызывать sendRedirect().

⇓

```
String encodedURL = response.encodeRedirectURL(originalURL);
response.sendRedirect(encodeRedirectURL);
```

Поскольку в глобальной сети нельзя знать, отключит или нет пользователь cookies в своем браузере, модифицировать URL рекомендуется всегда.

4.3. Переадресация

Чтобы удовлетворить запросы пользователя, сервлет может использовать ресурсы сети: другие сервлеты, HTML-страницы, разделяемые объекты и прочее.

Сервлет может получить доступ к ресурсам двумя способами:

- сделав HTTP-запрос через объект URL;
- сделав запрос через объект RequestDispatcher, если ресурс расположен на локальном сервере.

Ресурсы локального сервера

Для доступа к ресурсу на локальном сервере создается объект RequestDispatcher, который служит оболочкой для ресурса.

Получим ссылку на объект RequestDispatcher для данного ресурса

⇓

```
RequestDispatcher dispatcher =
  getServletContext().getRequestDispatcher("/bookstore/bookstore.html");
```

Аргументом метода getRequestDispatcher() является URL требуемого ресурса. URL может быть абсолютным или относительным (только в версии Servlets 2.2 или выше). Если ресурс недоступен, метод возвращает null.

Переадресация запроса методом forward( )

Если диспетчер с ассоциированным ресурсом получен, этому ресурсу можно переадресовать запрос для обработки

⇓

```
dispatcher.forward(request, response);
```

Внешние ресурсы

Переадресация запроса к ресурсам, находящимся за пределами сервера, выполняется методом объекта response sendRedirect().

Действие метода состоит в том, что в ответ записывается заголовок 302 - SC_MUVED_TEMPORARILY и новый URL. Получив такой ответ, браузер должен автоматически послать новый запрос уже по этому URL.

Такой вид переадресации можно применять и к ресурсам, расположенным в пределах сервера, но от браузера потребуется

дополнительное соединение, поэтому переадресацию к локальным ресурсам предпочтительней выполнять при помощи диспетчера ресурсов.

Переадресацию необходимо делать до того, как получен объект PrintWriter или ServletOutputStream. В противном случае получим исключение IllegalStateException.

Замечание. При генерации HTML-кода надо учитывать, что метод sendRedirct() получает новый ресурс вместе с новым контекстом, который включает Document Base, текущий каталог и прочее, а метод forward() получает новый ресурс в старом контексте. Если в тэгах <A> или <FORM> вы использовали относительные адреса, то при переадресации методом forward() абсолютные адреса будут определены неправильно.

4.4. Включение ресурсов

Если вы уже начали вывод в поток данных ответа, вместо метода forward() можно вызвать метод include().

⇓

```
dispatcher.include(request, response);
```

Выводить данные в поток ответа можно и до, и после вызова метода include(), но включаемый ресурс не должен изменять HTTP-заголовки ответа.

4.5. Задание

1. Разработать сервлет, который в упрощенной форме воспроизводит телеигру "Поле чудес". На форме имеется слово, буквы которого заменены звездочками, и поле для ввода буквы. После нажатия на кнопку "Submit" буква поступает к сервлету. Если буква в слове есть, то клиенту передается частично разгаданное слово, и т.д., пока все буквы не будут угаданы. Сервлет также ведет счет попыткам клиента угадать слово.

2. Сделать то же самое, использовав для идентификации сеанса механизм модификации URL. При проверке сервлета отключить возможность приема cookies в браузере.

3. Решить ту же задачу, не используя объект сессии и идентифицируя клиента при помощи скрытых полей формы. На сервере необходимо вести учет клиентов самостоятельно разработанными средствами.

5. АППЛЕТЫ В РОЛИ КЛИЕНТОВ СЕРВЛЕТОВ

До сих пор в роли клиентов сервлетов обычно выступали HTML-формы. Если клиенту требуется сложный пользовательский интерфейс, то вместо них можно применить апплеты. Апплет должен отослать запрос и принять ответ сервера.

5.1. Передача методом GET и получение новой страницы

Если данные, полученные от сервера, можно показать в виде отдельной HTML-страницы, а не непосредственно в апплете, то следует сформировать запрос, образовать объект URL и вызвать метод showDocument(URL). Это метод не апплета, а интерфейса AppletContext, который можно запросить у браузера методом апплета getAppletContext().

Показывать страницу, полученную от сервера, нужно в отдельном окне или в отдельном фрейме.

Пример "Коммуникация". Имеется апплет с полем ввода и кнопкой. При нажатии на кнопку содержимое поля ввода textField посылается сервлету MyServlet, там переводится с "молдавского" на русский и присылается обратно.

Обработчик нажатия на кнопку выполняет следующий код

⇓

```
try {

  // Заменить запрещенные символы разрешенными
  String mold = URLEncoder.encode(textField.getText());

  // Построить URL запроса
  URL codeBase = getCodeBase();
  URL url = new URL(codeBase,
      "/myapp/servlet/myapp.MyServlet?data=" + mold);

  // Разместить ответ в отдельном фрейме с именем answer
  getAppletContext().showDocument(url, "answer");

} catch (Exception  ex) {
  showStatus("Error. " + ex);
}
```

Предложение на молдавском присоединяется к URL запроса, поэтому должно быть предварительно переработано так, чтобы не иметь знаков

пробела и других запрещенных символов. Это делает статический метод encode() класса java.net.URLEncoder.

Метод getCodeBase() возвращает URL каталога, в котором расположены файлы классов выполняемого апплета. Относительно этого места достраиваем наш запрос таким образом, чтобы получился адрес нужного сервлета.

Клиентский HTML-документ содержит набор фреймов.
⇓

```
<frameset rows="70%, *">
  <frame src="MyApplet.html">
  <frame src="" name="answer">
</frameset>
```

Тэг <applet> в документе test.html
⇓

```
<applet
  codebase = "."
  code     = "myapp.MyApplet.class"
  name     = "MyApplet"
  width    = "400"
  height   = "300"
  hspace   = "0"
  vspace   = "0"
  align    = "middle">
</applet>
```

Серверная часть приложения представлена сервлетом, который находит в запросе параметр с именем “data”, переводит его значение с молдавского на русский и возвращает результат как простой текст.

Метод doGet() выполняет код
⇓

```
// Установить в заголовке Content-Type простой текст и кириллицу
response.setContentType("text/plain; charset=windows-1251");

// Получить выходной поток
PrintWriter out = response.getWriter();
```

```
// Получить исходные данные из запроса
String data = request.getParameter("data");

// Записать в поток перевод
out.println(TranslateFromMoldova(data));

out.close();
```

Эту же серверную часть будем использовать и в других примерах.

5.2. Передача методом GET и HTTP-тунеллирование

Результат запроса можно вернуть непосредственно апплету, чтобы он сам представил его пользователю. Этот подход называют HTTP-тунеллированием, т.к. пользовательский коммуникационный протокол встраивается в протокол HTTP.

Чтобы реализовать такой способ обмена данными, необходимо:

1) сформировать GET-запрос и присоединить к нему данные (как в предыдущем разделе);
2) в апплете установить HTTP-соединение с программой на стороне сервера (сервлетом, статической web-страницей, CGI);
3) чтение данных из потока байтов или потока объектов, как в сериализации.

Пример "Коммуникация-2". Кроме поля ввода и кнопки, апплет имеет метку label для отображения полученного от сервера перевода с молдавского.

Для данного примера не нужна web-страница с фреймами, нужна только страница с тэгом <applet>.

При нажатии на кнопку выполняется следующий код

⇓

```
// Собрать данные, небходимые для установки соединения
URL currentPage = getCodeBase();
String protocol = currentPage.getProtocol();  // версия протокола
String host = currentPage.getHost(); // имя хоста
int port = currentPage.getPort();       // номер порта

// Заменить запрещенные символы разрешенными
String mold = URLEncoder.encode(text1.getText());

// Сформировать путь запроса и параметры
```

```
String urlSuffix = "/myapp/servlet/myapp.MyServlet?data=" + mold;

try {
  // Создать URL запроса
  URL url = new URL(protocol, host, port, urlSuffix);

  // Установить соединение
  URLConnection connection = url.openConnection();

  // Отказаться от кеширования данных на клиенте
connection.setUseCaches(false);

// Задать заголовки запроса, если это нужно
// connection.setRequestProperty("header", "value");

// Получить поток для чтения байтов
BufferedReader in = new BufferedReader(
    new InputStreamReader(connection.getInputStream()));

// Прочитать данные в буфер
StringBuffer buf = new StringBuffer();
String line;
while ((line = in.readLine()) != null) {
    buf.append(line);
}

    // Отобразить данные в апплете
    label.setText(new String(buf));

  } catch (IOException  ex) {
showStatus("Error. " + ex);
  }
```

В запросе можно использовать полный URI серверного ресурса, но, поскольку апплету запрещено устанавливать связь с машинами, кроме той, что является его домашним сервером, целесообразно выделить адрес этого домашнего сервера.

Спецификация протокола HTTP говорит, что сервер автоматически закрывает соединение, когда закончено считывание документа.

5.3. Передача методом POST и HTTP-тунеллирование

Предыдущий пример не позволял апплету посылать на сервер произвольные двоичные данные, а только символьные, поскольку двоичные данные не могут быть параметрами GET-запроса.

Метод POST позволяет воспользоваться установленным соединением для обмена данными в обоих направлениях.

Пример "Коммуникация-3". Запрос выполняется методом POST, а не методом GET.

При нажатии на кнопку в апплете выполняется следующий код.

⇓

```
// // Собрать данные, небходимые для установки соединения
URL currentPage = getCodeBase();
String protocol = currentPage.getProtocol();
String host = currentPage.getHost();
int port = currentPage.getPort();

// Сформировать путь запроса
String urlSuffix = "/myapp/servlet/myapp.MyServlet;

try {
  // Создать URL запроса
  URL dataURL = new URL(protocol, host, port, urlSuffix);

  // Установить соединение
  URLConnection connection = dataURL.openConnection();

  // Отказаться от кеширования данных на клиенте
  connection.setUseCaches(false);

  // Сделать соединение пригодным и для чтения, и для записи
  connection.setDoOutput(true);

  // Создать поток байтов для буферизации посылаемых данных
  ByteArrayOutputStream bytes = new ByteArrayOutputStream(512);
  PrintWriter out = new PrintWriter(bytes, true);

  // Собрать данные в буфере для определения их количества
  String data = "data=" + text.getText();
  out.print(data);
  out.flush();
```

```
// Задать количество байт в заголовке Content-Length
connection.setRequestProperty("Content-Length",
    String.valueOf(bytes.size()));

// Задать тип данных в заголовке Content-Type
connection.setRequestProperty("Content-Type",
    "application/x-www-form-urlencoded");

// Послать данные на сервер
bytes.writeTo(connection.getOutputStream());

// Принять ответ сервера
BufferedReader in = new BufferedReader(
    new InputStreamReader(connection.getInputStream()));
String line;
StringBuffer buf = new StringBuffer();
while ((line = in.readLine()) != null) {
    buf.append(line);
}

    // Отобразить данные в апплете
    label.setText(new String(buf));

  } catch (Exception ex) {
showStatus("Error. " + ex);
  }
```

Заголовок Content-Type = "application/x-www-form-urlencoded" по умолчанию устанавливается в IE. Здесь он установлен, чтобы обеспечить независимость от браузера.

5.4. Задание

1. Разработать апплет, который показывает текущее время на сервере при помощи часов с циферблатом и стрелками. Показания часов обновляются, если щелкнуть мышью по циферблату.

2. Разработать апплет, который позволяет открыть и показать любое из изображений в формате gif, находящихся в определенном каталоге на сервере.

6. СЕРВЕРНЫЕ СТРАНИЦЫ JAVA

У технологии сервлетов есть много достоинств, но есть и один несомненный недостаток. Он в том, что разработчику трудно формировать требуемый HTML-ответ при помощи одного только метода print().

Технология серверных страниц Java, (JSP - Java Server Pages) устраняет этот недостаток, существенно упрощая создание web-страниц на стороне сервера. При этом сохраняются все достоинства сервлетов: переносимость, эффективность и использование языка Java.

Технология JSP позволяет комбинировать в одном файле программный код на Java и HTML-представление web-страницы. JSP поддерживают те же web-серверы, что поддерживают сервлеты: Tomcat, Resin и т.п.

6.1. Жизненный цикл JSP

Исходный код JSP сначала транслируется в исходный код сервлета, а последний транслируется в код класса. Когда клиент запрашивает JSP, то, прежде всего, проверяется, что старше - исходный код JSP или код порожденного сервлета. Если страница была изменена, код сервлета окажется старше и это вызовет новую трансляцию страницы.

Ошибки трансляции

На обеих фазах трансляции страницы могут быть обнаружены ошибки, которые проявят себя, когда на страницу поступит запрос.

Если ошибка обнаружена на первой фазе трансляции, например, встретился неправильный JSP-элемент, сервер выбрасывает исключение ParseException, а исходный код сервлета остается пуст или неполон. Последняя неполная строка укажет на место ошибки в коде JSP.

Если ошибка произошла на второй фазе трансляции, сервер выбрасывает исключение JasperExcetion с сообщением, которое содержит имя сгенерированного сервлета и номер его ошибочной строки в исходном коде сервлета.

Выполнение

Когда JSP-сервлет загружен впервые, выполняется его метод jspInit().

Запросы, поступающие к JSP-странице, обрабатываются методом _jspService().

Перед удалением сервлета вызывается метод jspDestroy().

6.2. Содержание страницы

JSP-страницу можно рассматривать как обычную HTML-страницу, в код которой вставлены элементы сценария, выполняемого на сервере.

Иными словами, на странице чередуются представительский код (тэги HTML) и код, написанный на Java. После превращения в сервлет представительский код займет место параметров в вызовах метода print(), а сценарий, написанный на Java, станет частью кода сервлета.

Пример. Страница, которая возвращает приветствие в зависимости от времени суток

⇓

```
<%@ page contentType="text/html; charset=windows-1251" %>
<%
 java.util.Calendar rightNow = java.util.Calendar.getInstance();
 int hour = rightNow.get(java.util.Calendar.HOUR_OF_DAY);
 String greeting;
 if (hour < 12)
   greeting = "Доброе утро!";
 else if (hour < 17)
   greeting = "Добрый день!";
 else if (hour < 22)
   greeting = "Добрый вечер!";
 else
   greeting = "Доброй ночи!";
%>
<html>
<body>
<%=greeting%>
</body>
</html>
```

Замечание. Технология JSP рассчитана на широкий спектр типов содержания, по умолчанию это - HTML, но может быть и другим.

Иной тип устанавливается директивой, например, для WML

⇓

```
<%@ page contentType="text/vnd.wap.wml"%>
```

6.3. Элементы сценария

Сценарий JSP-страниц пишут на языке высокого уровня. По умолчанию это Java, но есть возможность использования и других языков.

Элементы сценария заключают в специальные тэги, которые бывают следующих типов:

- **<%! %>** - декларация (объявление переменных – полей, статических полей или методов класса JSP-сервлета)

- <% %> - скриплет - операторы Java, которые вставляются в код метода service() сгенерированного сервлета. Переменные, объявленные в скриплете, являются локальными переменными метода service() и доступны на данной странице.
- <%= %> - выражение, которое станет параметром метода out.print(…). Точка с запятой в выражениях запрещена.
- <%-- --%> - комментарий, который остается на сервере.
- <%@ %> - директивы контейнеру страниц.

Другой язык сценария устанавливается директивой

⇓

```
<%@ page language="scripting language" %>
```

Поскольку скрипты транслируются в код сервлета, все необходимые пакеты надо импортировать директивой

⇓

```
<%@ page import="packagename.*,fully_qualified_classname" %>
```

Рекомендуется минимальное использование элементов сценария на странице JSP. Выполняемый код следует размещать в java-бинах, а также применять пользовательские тэги (см. раздел "Библиотеки тэгов JSP").

6.4. Использование объектов

Встроенные объекты JSP

При написании кода сценария программист может и должен использовать уже готовые объекты. Перечень их приведен в таблице.

Тип и объект	Описание
ServletContext application	Основное применение - хранение глобальных объектов приложения. Объекты хранятся в виде именованных атрибутов, для доступа к которым имеет методы getAttribute() и setAttribute()
HttpServletRequest request	Объект запроса
HttpServletResponse response	Объект ответа

jsp.PageContext pageContext	Характеризует текущее состояние контейнера сервлетов и обеспечивает доступ к другим объектам (сеансу, приложению, выходному потоку). Сам по себе используется редко
HttpSession session	Объект сессии. Позволяет хранить объекты, специфичные для данного сеанса. Для доступа к ним имеет методы getAttribute() и setAttribute()
Throwable exception	Объект доступен только на странице ошибок
jsp.JspWriter out	Поток ответа
ServletConfig config	Данные о конфигурации порожденного сервлета
Object page	Реализация порожденного сервлета. То же, что this

Объекты пользователя

Объекты пользователя должны быть основным вместилищем серверного кода web-приложения, это облегчит работу над пользовательским интерфейсом.

Имеется несколько способов создать объект на JSP.

Объект может быть полем или статическим полем JSP-сервлета. Такой объект создается в декларации, а затем используется в скриплетах и выражениях.

⇓

```
<%! MyObject myObject = new MyObject(); %>
```

Объект может быть локальной переменной метода сервлета, обрабатывающего запрос. Такие объекты создаются в скриплетах.

⇓

```
<%= (new MyObject()).getSomeValue() %>
```

Если объект является Java-бином, его можно создать и использовать при помощи JSP-элемента <jsp:useBean>.

⇓

```
<jsp:useBean id="beanName"
 class="fully_qualified_classname" scope="scope">
 <jsp:setProperty .../>
```

```
</jsp:useBean>
```

О JSP-элементах будет сказано позже.

Разделение объектов

Если нужно разделять объект между запросами одного или всех клиентов, его делают атрибутом встроенных объектов области видимости: application, session, request, page.

6.5. Директива page

Директивы влияют на структуру сервлета, который получается из JSP. На странице используются три директивы:

- include – включение кода перед компиляцией сервлета;
- taglib – говорит, что страница будет использовать библиотеку тэгов;
- page – задает настройки, задаваемые различными атрибутами.

Рассмотрим некоторые атрибуты директивы page.

Атрибут import

Атрибут import формирует оператор import сервлета.
По умолчанию сервлет будет импортировать имена из пакетов: javax.servlet.*, javax.servlet.jsp.*, javax.servlet.http.*.

Пример импорта из двух пакетов.
⇓

```
<%@ page import="mypack.jsp.*, hispack.jsp.*" %>
```

Импортируемые пакеты должны располагаться в каталогах, определяемых контейнером сервлетов. Для Tomcat это "app/web-inf/classes" (app – корневой каталог web-приложения) или "tomcat_home/classes". JAR-файлы могут располагаться в каталоге "tomcat_home/lib".

Если файлы в этих каталогах изменяются, сервер надо перезапустить.

Атрибут contentType

Устанавливает заголовок ответа Content-Type, который задает MIME-тип документа, посылаемого клиенту.

Помимо MIME-типа, заголовок может задавать набор символов.
⇓

```
<%@ page contentType="text/html; charset=windows-1251" %>
```

см. раздел "Ответ сервера. Заголовки ответа".

Атрибут isThreadSafe

Подобно сервлету страница может быть объявлена потокобезопасной.

Если страница объявлена потокобезопасной
⇓

```
<%@ page isThreadSafe="true" %>
```

то она должна сама синхронизировать доступ к полям своего сервлета (но не к локальным переменным метода service, которые создаются заново при каждом вызове метода).

По умолчанию страница не потокобезопасна и не может разделяться разными потоками, что может отрицательно сказаться на производительности приложения.

Пример. Потокобезопасная страница, на которой каждый пользователь получает уникальный идентификатор.
⇓

```
<%@ page isThreadSafe="true" %>
...
<%! private int idNum = 0 %>
<%
 sinchonized(this) {
    out.println("Your Id is " + idNum);
    idNum++;
 }
%>
...
```

Атрибут session

Атрибут session показывает, что предопределенный объект session должен быть связан с существующим сеансом, если такой имеется. В противном случае должен быть создан новый сеанс.
⇓

```
<%@ page session="true" %>
```

Значение false показывает, что сеансы не будут использоваться. По умолчанию сеансы используются.

Другие атрибуты

Атрибут	Что задает
buffer = "Xkb"	Размер буфера
autoflash = "true | false"	Автоматическую очистку буфера при заполнении
extends = "package.class"	Наследуемый пользовательский класс сервлета
info = "Some Message"	Строку, возвращаемую методом сервлета getServletInfo()
errorPage="RelativeURL"	Адрес страницы ошибок
isErrorPage="true | false"	Может ли страница служить страницей ошибок
language="java"	Язык написания сценария. Пока это только Java

6.6. Обработка ошибок JSP

Ошибки могут происходить во время компиляции в сервлет или во время запроса к jsp. Первые обрабатывает контейнер сервлетов и JSP и сообщает о них клиенту как об ошибках сервера с кодом 500.

Вторые могут происходить как в самом сервлете, так и в вызываемых бинах и выбрасывают исключения. Исключения могут быть обработаны на самой вызываемой странице или могут быть переданы через объект exception на другую страницу, специально предназначенную для обработки ошибок (error page).

Страница ошибок назначается директивой расположенной на основной странице.

⇓

```
<%@ page errorPage="errorpage.jsp" %>
```

Пример. Простая страница ошибок.

⇓

```
<html>
<body text="red">
<%@ page isErrorPage="true" %>
Error: <%= exception.getMessage() %> has been reported.
</body>
</html>
```

На самой странице ошибок находится директива, которая отличает ее от обычных страниц.

⇓

```
<%@ page isErrorPage="true" %>
```

Механизм использования страниц ошибок состоит в том, что при возникновении ошибки на основной странице создается объект exception с информацией об ошибке, помещается в запрос в качестве атрибута, после чего запрос перенаправляется странице ошибок. Там объект исключения извлекается и обрабатывается.

Объект исключения попадает в запрос, благодаря методу

pageContext.handlePageException(ex);

который вызывается в сервлете основной страницы. Этот же метод выполняет переадресацию.

6.7. XML-элементы страницы

Помимо представительского кода, элементов сценария и директив на страницах JSP могут находиться элементы, имеющие XML-подобный синтаксис. Некоторые из них дают альтернативное представление директив, например, элемент

<jsp:directive page isErrorPage="true" />

означает то же, что директива

<%@ page isErrorPage="true" %>.

Другие имеют самостоятельное значение.

Чтобы правильно записывать XML-элементы, необходимо помнить, что:

- каждый тэг должен иметь парный закрывающий либо косую черту в конце одиночного тэга;
- значения атрибутов обязательно заключаются в кавычки.

6.8. Задание

1. Разработать серверное приложение, которое предоставляет возможность ввести коэффициенты квадратного уравнения, решает его и показывает корни. В приложении использовать 2 JSP - одна для задания коэффициентов, другая для демонстрации результатов.

2. Предъявить клиенту прайс-лист, где указаны товары и цены. Клиент должен сделать заказ, проставив количество экземпляров против каждого интересующего его товара. Подсчитать и сообщить клиенту суммарную стоимость сделанного им заказа.

3. Организовать простую конференцию, для небольшого количества участников. Каждый участник может высказываться и читать протокол, в котором собраны все ранее сделанные высказывания в хронологическом порядке.

7. ПЕРЕАДРЕСАЦИЯ И ВКЛЮЧЕНИЕ

7.1. Переадресация запроса

Локальная переадресация

Переадресация запроса к другой странице того же приложения выполняется JSP-элементом **jsp:forward.**

⇓

```
<jsp:forward page="/mypage.jsp" />
```

При включении и переадресации другой странице передается объект request, содержащий параметры запроса. К ним можно добавить новые параметры при помощи элемента **jsp:param**.

```
<jsp:include page="..." >
  <jsp:param name="param1"  value="value1"/>
</jsp:include>
```

Действие элемента jsp:forward аналогично методу forward диспетчера ресурсов. В JSP этот метод можно вызвать через pageContext.forward(destURL), не получая явно ссылку на диспетчер.

Глобальная переадресация

Переадресация за пределы сервера выполняется методом
response.sendRedirect(otherURL);

7.2. Включение содержания в JSP

Включить содержание в страницу можно двумя способами: директивой и JSP-элементом.

Директива **<%@ include file="filename" %>** приводит к тому, что перед трансляцией страницы в сервлет означенный файл вставляется на место директивы. Таким способом обычно вставляют баннеры, копирайт и т.п.

JSP-элемент **<jsp:include page="includedPage" flush="true"/>** во время выполнения запроса вызывает другую страницу, которой передаются объекты request и response, а полученный ответ вставляется в выходной поток данной страницы. Оба параметра директивы обязательны.

7.3. Включение апплетов в JSP

Включение апплетов выполняется при помощи элемента **jsp:plugin**, который позволяет единым образом включать апплеты AWT и Swing в страницы, предназначенные для разных браузеров.

⇓

```
<jsp:plugin type="applet" code="MyApplet.class" width="475"
height="350" >
</jsp:plugin>
```

Атрибуты элемента jsp:plugin:

- type = applet | bean;
- code = файл класса апплета;
- codebase = каталог с файлом класса апплета;
- align = left | right | top | bottom | middle – выравнивание апплета на странице;
- hspace = свободное место слева и справа от апплета;
- vspace = свободное место вверху и внизу апплета;
- archive = файл jar, из которого апплет загружает файлы и изображения;
- name = идентификатор апплета в клиентских скриптах;
- title = текст всплывающей подсказки;
- jreversion=версия Java Runtime Environment. По умолчанию 1.1;
- iepluginurl=URL, с которого будет загружаться плагин для браузера Internet Explorer. По умолчанию сайт компании Sun Microsystems. Для интранет нужно указать на локальную копию плагина;
- nspluginurl=то же для браузера Netscape.

Параметры апплета задаются при помощи элементов **jsp:params** и **jsp:param**, которые вложены в элемент jsp:plugin.

⇓

```
<jsp:params>
 <jsp:param name="PARAM1" value="VALUE1" />
 <jsp:param name="PARAM2" value="VALUE2" />
</jsp:params>
```

7.4. Задание

1. Разработайте приложение в помощь молодым родителем. Приложение содержит 3 страницы: титульная, страница со списком мужских имен, страница со списком мужских имен. Родитель вводит пол ребенка на титульной странице и получает одну из двух страниц в зависимости от пола. Используйте переадресацию.

2. Включите апплеты, из заданий 5-го раздела не в HTML-документы, а в JSP.

8. ИСПОЛЬЗОВАНИЕ BEAN-КОМПОНЕНТОВ

Как ранее упоминалось, наилучшей стратегией при разработке JSP-страниц является их максимальная разгрузка от кода сценария и перенесение его в классы Java. Если эти классы к тому же поддерживают архитектуру bean-компонентов, то их взаимодействие с кодом страницы еще более упрощается.

8.1. Помещение компонента на страницу

Вкратце требования к компонентам Java bean, помещаемым на страницу, сводятся к следующему:

- bean-компонент должен иметь конструктор без параметров;
- bean-компонент не должен иметь открытых полей;
- bean-компонент должен иметь свойства – пары функций setXxx() и getXxx() (для булевых свойств 2-я функция называется isXxx()).

Элемент **jsp:useBean** загружает компонент для использования на странице.

⇓

```
<jsp:useBean id="myname" class="mypackage. MyClass" />
```

Этот элемент можно считать эквивалентом скриплета,

⇓

```
<% package.Class myname = new mypackage.MyClass(); %>
```

если не обращать внимания на то, что компонент создается только при первом обращении к этому коду. Если компонент уже существует, ссылка на него просто помещается в локальную переменную myname метода _jspService().

Вместо атрибута class разрешается использовать атрибут beanName. Последний может ссылаться и на класс, и на файл, содержащий сериализованный bean-объект. Значение атрибута beanName передается в метод instantiate() класса java.beans.Bean.

Можно сделать так, чтобы ссылка на bean-объект имела тип его предка или реализованного в нем интерфейса. Это достигается с помощью атрибута type.

⇓

```
<jsp:useBean id="myname" class="MyClass" type="Runnable" />
```

Есть несколько символов и комбинаций символов, которые требуют специального представления внутри значений атрибутов:

символ ' представляется \',
символ " представляется \",
символ \ представляется \\,
символы %> представляются %\>,
символы <% представляются <\%.

8.2. Получение и задание свойств компонентов

Поместить значение свойства bean-компонента на страницу можно двумя способами

⇓

```
<%--1-й способ --%>
<jsp:getProperty name="myObject" property="foo" />

<%--2-й способ --%>
<%= myObject.getFoo() %>
```

Значение атрибута name должно быть таким же, как значение атрибута id элемента jsp:useBean.

Задать значение свойства можно тоже двумя способами

⇓

```
<%--1-й способ --%>
<jsp:setProperty name="myObject" property="foo"  value="Any string"/>

<%--2-й способ --%>
<% myObject.setFoo("Any string"); %>
```

Если в элементе jsp:setProperty вместо атрибута value использовать атрибут param="paramName", то значением свойства станет значение параметра запроса с именем paramName.

⇓

```
<jsp:setProperty name="someObject" property="foo" param="paramName"/>
```

Простые преобразования строкового значения параметра к типу свойства выполнятся автоматически, например, если свойство имеет тип Integer, а значение параметра - stringValue, выполнится преобразование

Integer.valueOf(stringValue);

если свойство имеет тип int, выполнится преобразование

Integer.valueOf(stringValue).intValue().

8.3. Разделение Bean-компонентов

Помимо того, что ссылка на Bean-компонент помещается в локальную переменную метода _jspService(), она может быть помещена в один из объектов области видимости, откуда может быть извлечена методом объекта getAttribyte().

Нужный объект выбирается в элементе jsp:useBean при помощи атрибута scope:

- scope="page" – на время выполнения запроса ссылка помещается в объект page класса pageContext. Эта область видимости действует по умолчанию.
- scope="request" – на время выполнения запроса ссылка помещается в объект request класса HttpServletRequest. Практически то же, что page.
- scope="session" – ссылка будет храниться в объекте session. Если директива page говорит, что данная страница не участвует в сеансе, будет выброшено исключение.
- scope="application" – ссылка будет храниться в объекте application класса ServletContext, который используется совместно всеми сервлетами одного приложения.

8.4. Условная инициализация компонентов

Вместо элемента

⇓

```
<jsp:useBean id="name" class="package.Class" />
```

можно использовать элемент

⇓

```
<jsp:useBean id="name" class="package.Class">
 // здесь расположены элементы jsp:setProperty
</jsp:useBean>
```

Элементы внутри тэга выполнятся только, если на странице происходит создание компонента. Если происходит связывание, элементы игнорируются. Это гарантирует то, что свойства компонента будут установлены лишь однажды.

8.5. Задание

Выполните задания раздела 6, максимально разгрузив код JSP-страниц за счет использования bean-компонентов.

Учебное издание

Бондарев В. М.

Учебное пособие по программированию на Java

Редактор: Попко Е. Г.

Компьютерная верстка и дизайн: Горб В. С.

Художник обложки: Денисенко А. А.

Підп. до друку Формат

Умов. друк. арк.

Облік вид. арк. Зам. № Тираж прим.

Ціна договірна.

Видавництво «Компанія СМІТ».
61166, м. Харків, просп. Леніна, 14.
Тел.: 8-(0572)-17-54-94, 702-08-16
Факс: 8-(0572)-14-23-66
E-mail: book@smit.kharkov.ua

Свідоцтво про внесення суб'єкта видавничої справи до державного реєстру видавців, виготівників і розповсюджувачів видавничої продукції ДК № 435 від 24.04.2001