КУРС ЛЕКЦИЙ

ОБЪЕКТНО – ОРИЕНТИРОВАННЫЕ ЯЗЫКИ ПРОГРАММИРОВАНИЯ

ЛЕКЦИЯ . ТЕМА: ОСНОВНЫЕ КОНСТРУКЦИИ

План: Переменные и константы; Операции; Указатели и ссылки; Структура программы; Именованные константы; Преобразование типов.

Переменные и константы

Переменные. Имя переменной идентификатор то есть последвательность из латинских букв, арабских цифр, знака подчеркивания и знака доллара начинающийся с буквы или знака. Java — язык, чувствительный к регистру букв. Это означает, что, к примеру, Value и VALUE — различные идентификаторы.

Основные встроенные типы: **byte,char, short, int, long, float, double, boolean.**

Модификатор типа **unsigned** в Java не используется.

Простейшая форма определения переменных:

<тип> <список_имен_переменных>;

Переменным можно присваивать начальные значения, явно указывая их в определениях:

<тип> <имя_переменной> = <инициализатор>;

Этот прием называется инициализацией.

Примеры:

float pi = 3.14 , cc=1.3456;

int year = 1999;

Переменные, для которых начальные значения не указаны, автоматически инициализируются нулем.

Константы. Константа - это значение, которое не может быть изменено:

Целая константа может быть десятичной восмеричной или шестнадцатеричной. Десятичная константа это последовательность десятичных цифр не начинающийся с нуля (пример: 8, 0, 192345); Восьмеричная константа это последоательность восмеричных цифр начинающихся с нуля(пример: 016, 01); Шестнадцатеричная константа это последовтельность шестнадцатеричных цифр начинающихся с символов 0x или 0X(пример: 0xA, 0X00F);

Символьная константа это взятая в апостриф один (пример: ‘ғ’,’r,’6’) или несколько (пример:‘\n’, ‘\0xF5’) символов. Символы которые начинаются с слеш символа ‘\’ называются управляющими или эскейп символами.

Вещественная константа может быть в двух формах с фиксированной точкой (пример: 5.7, .0001, 41.) и с плавающей точкой (пример: 0.5e5, .11e-5, 5E3).

Логическая константа можеть иметь два значения true(истина) или false(ложь). Внутренняя форма представления false – 0, любое другое значение интерпретируется как true.

Строчные константы в Java выглядят точно также, как и во многих других языках — это произвольный текст, заключенный в пару двойных кавычек ("").Строчные константы в Java должны начинаться и заканчиваться в одной и той же строке исходного кода.

Операции

Арифметические операции. Арифметические операции делятся на бинарные и унарные операции. К бинарным операциям относятся: сложение +, вычитание –, умножение *, деление / и взятие модуля %.

Например: 20/3=6; (-20)/3=-6; 5%2=1;

К унарным операциям относятся унарный минус – и унарный плюс +; инкремент ++ и декремент--. Операции инкремент и декремент можно использовать в префиксной то есть ++i, или постфиксной то есть i++ форме. Например если i=2, то 3+(++i)= 6, 3+i++=5. В обеих случаях i=3.

Операции сравнения: К операциям сравнения относятся равно ==; не равно !=; больше или равно > =; меньше или равно <=;больше >; меньше <. Значения операций сравнения является логическим.

Логические операции. К логическим операциям относятся конъюнкция &&, дизъюнкция ||, отрицание !.

Разрядные операции. К разрядным операциям относятся разрядная конъюнкция &, разрядняя дизъюнкция |, разрядняя иключающая дизъюнкция ^, разрядное отрицание !, разрядный сдвиг влево << и разрядный сдвиг вправо >>. Например 5 в двоичной системе равен 101 а 6 равен 110:

6&5= 4=100; 6|5= 7= 111; 6^5=3=011; ~6=4=010.

5<<2=20 или 101<<2=10100; 5>>2=1 или 101>>2=001=1.

Операции присваивания. Простая операция присваиваивания является бинарной операцией где обычно левый операнд переменная, а правый операнд выражение:

переменная = выражение;

Z=4.7+3.34; C=y=f=4.2+2.8;

Сложная операция присваивания является унарной операцией и имеет форму:

Переменная операция = выражение;

Здесь операция одно из следующих операций *,/,%,+,-, &,^,|, <<,>>.

Например: x+=4 эквивалентно x=x+4;

x>>=4 эквивалентно x=x>>4;

Условная операция. Условная операция называется тернарной и состоит из трех операнд:

<1-выражение>?<2-выражение>:<3-выражение> например:a<b?a:b .

Структура программы

Язык Java требует, чтобы весь программный код был заключен внутри поименованных классов. Класс это сложный тип объединяющий переменные и методы. Определение класса начинается со служебного слова class. Стандартные классы языка расположены в различных пакетах. Таким образом стандартные пакеты являются стандартными библиотеками классов.

Пример:

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

Приведенный выше текст примера надо записать в файл HelloWorld.java.

Для того, чтобы оттранслировать этот пример необходимо запустить транслятор Java — javac, указав в качестве параметра имя файла с исходным текстом:

C: \> javac HelloWorld.Java

Транслятор создаст файл HelloWorld.class с независимым от процессора байт-кодом нашего примера. Для того, чтобы исполнить полученный код, необходимо иметь среду времени выполнения языка Java (в нашем случае это программа java), в которую надо загрузить новый класс для исполнения. Подчеркнем, что указывается имя класса, а не имя файла, в котором этот класс содержится.

C: > java HelloWorld

Hello World

В данном случае модификатор доступа public означает, что метод main виден и доступен любому классу.

С помощью слова static объявляются методы и переменные класса, используемые для работы с классом в целом. Методы, в объявлении которых использовано ключевое слово static, могут непосредственно работать только с локальными и статическими переменными.

Если не требуется возвращать значение из метода main, используется модификатор void.

Все существующие реализации Java-интерпретаторов, получив команду интерпретировать класс, начинают свою работу с вызова метода main. Java-транслятор может оттранслировать класс, в котором нет метода main. А вот Java-интерпретатор запускать классы без метода main не умеет.

Элемент String args[] объявляет параметр с именем args, который является массивом объектов — представителей класса String.

Для вывода на экран используется метод println объекта out. Объект out объявлен в классе OutputStream и статически инициализируется в классе System.

Именованные константы

В языке Java для обозначения констант используется ключевое слово final , например:

```
public class Constants
public static void main(String[] args)
final double CM_PER_INCH = 2.54;
double paperWidth = 8.5;
double PaperHeight = 11;
System.out.println("Размер страницы в сантиметрах: "
+ paperWidth * CM_PER_INCH + "на"
+ paperheight * CM_PER_INCH);
```

Ключевое слово final означает, что присвоить какое-либо значение данной переменной можно лишь один раз и навсегда. Использовать в именах констант только прописные буквы необязательно.

В языке Java часто необходимы константы, доступные нескольким методам внутри одного класса. Обычно они называются *константами класса* (class constants). Константы класса объявляются с помощью ключевых слов static final. Вот пример использования константы класса.

```
public class Constants2
{
public static final double CM_PER_INCH = 2.54;
public static void main(String [] args)
{
double paperWidth = 8.5;
double PaperHeight = 11;
System.out.printIn("Размер страницы в сантиметрах: "
+ paperWidth * CM_PER_INCH + "на"
+ paperHeight * CM_PER_INCH);
```

Отметим тот факт, что константа класса задается *вне* метода main. Таким образом,ее можно использовать в других методах того же класса. Более того, если (как в данном примере) константа объявлена как public , методы из других классов также могут получить к ней доступ. В примере это можно сделать с помощью выражения

Constants2.CM_PER_INCH.

В языке Java слово const является зарезервированным, однако сейчас оно уже не употребляется. Для объявления констант следует использовать ключевое слово final .

Преобразование типов

Приведение типов (type casting) используется, когда величина какого-то определенного типа, присваивается переменной другого типа. Для некоторых типов это можно проделать и без приведения типа, в таких случаях говорят об автоматическом преобразовании типов. В Java автоматическое преобразование возможно только в том случае, когда точности представления чисел переменной-приемника достаточно для хранения

исходного значения. Такое преобразование происходит, например, при занесении литеральной константы или значения переменной типа byte или short в переменную типа int. Это называется *расширением (widening)* или *повышением (promotion)*, поскольку тип меньшей разрядности расширяется (повышается) до большего совместимого типа. Размера типа int всегда достаточно для хранения чисел из диапазона, допустимого для типа byte, поэтому в подобных ситуациях оператора явного приведения типа не требуется. Обратное в большинстве случаев неверно, поэтому для занесения значения типа int в переменную типа byte необходимо использовать оператор приведения типа. Эту процедуру иногда называют *сужением (narrowing)*, поскольку явно сообщается транслятору, что величину необходимо преобразовать, чтобы она уместилась в переменную нужного типа. Для приведения величины к определенному типу перед ней нужно указать этот тип, заключенный в круглые скобки. Например

int a = 100;

byte b = (byte) a;

Если нужно *округлить* число с плавающей точкой до *ближайшего* целого числа (что во многих случаях является намного более полезным), используется метод Math.round.

double x = 9.997;

int nx = (int)Math.round(x) ;

Теперь переменная nx равна 10. При вызове метода round по-прежнему нужно выполнять приведение (int) , поскольку возвращаемое им значение имеет тип long, и присвоить его переменной типа i n t можно лишь с помощью явного приведения.

При попытке привести число одного типа к другому результат может выйти за пределы допустимого диапазона. В этом случае результат будет усечен. Например, выражение (byte) 300 равно 44. Поэтому рекомендуется явно проверять заранее, будет ли результат лежать в допустимом диапазоне после приведения типов.

Приведение между булевским и целыми типами невозможно. Это предотвращает появление ошибок. В редких случаях для того, чтобы привести булевское значение к числовому типу, можно использовать условное выражение b ? 1 : 0.

Автоматическое преобразование типов в выражениях

Если в выражении используются переменные типов byte, short и int, то во избежание переполнения тип всего выражения автоматически повышается до int. Если же в выражении тип хотя бы одной переменной — long, то и тип всего выражения тоже повышается до long. По умолчанию Java рассматривает целые константы, как имеющие тип int. Все целые константы, в конце которых стоит символ L или 1, имеют тип long.

Если выражение содержит операнды типа float, то и тип всего выражения автоматически повышается до float. Если же хотя бы один из

операндов имеет тип double, то тип всего выражения повышается до double. По умолчанию Java рассматривает все константы с плавающей точкой, как имеющие тип double. Все вещественные константы, в конце которых стоит F или f имеют тип float.

Контрольные вопросы:

1. Перечислите основные типы.
2. Дайте определение понятию указателя.
3. Укажите все операции.
4. В каких случаях нет необходимости явного приведения типов?
5. Как осуществляется автоматическое преобразование типов в выражениях?

Литература:

1. Смирнов Н.И. Java -2. Учебное пособие. М.:«Три Л», 2000. -320 с.
2. Арнольд К. и др. Язык программирования JAVA/ Вильямс. 2001. 624 с.
3. Брюс Эккель. Философия Java . Библиотека программиста. 2003.

ЛЕКЦИЯ 3. ТЕМА: ОПЕРАТОРЫ

План: Комментарии; Составные операторы; Считывание ввода; Операторы выбора; Операторы циклов; Операторы перехода

Комментарии

Комментарии являются невыполняемыми операторами и могут быть блочными (Например **/*это комментарий */**) или строчными (Например **//это комментарий)**. Комментарии, выделяемые символами /* и */, в языке Java не могут быть вложенными. Это значит, что фрагмент кода нельзя отключить, просто окружив его символами /* и * /, поскольку отключаемый код сам может содержать разделители */ и */.

Составные операторы

К составным операторам относят собственно составные операторы и блоки. В обоих случаях это последовательность операторов, заключенная в фигурные скобки. Блок отличается от составного оператора наличием определений в теле блока.

Однако в языке Java невозможно объявить одинаково названные переменные в двух вложенных блоках. Например, приведенный ниже фрагмент кода содержит ошибку и не будет скомпилирован.

```
public static void main(String[] args)
int n;
{
…
{
int k;
int n; // Ошибка -- невозможно переопределить переменную
// n во внутреннем блоке.
…
}
}
```

Считывание ввода

Чтобы считать информацию со "стандартного устройства ввода" (т.е. клавиатуры) необходимо создать диалоговое окно для ввода данных с клавиатуры.

Вызов метода JOptionPane.showInputDialog(promptString) выводит на экран диалоговое окно, в котором пользователь может набрать свои данные. Этот метод возвращает строку, набранную пользователем в этом окне.

Например, здесь показано, как запросить имя пользователя вашей программы.

```
String name = JOptionPane.showInputDialog("What is your name?");
```

Метод JOptionPane. showInputDialog возвращает строку, а не число. Для преобразова ния этой строки в число нужно использовать метод Integer .parselnt или Double.parseDouble, например:

String input = JOptionPane.showInputDialog("How old are you?");

int age = Integer.parselnt(input);

Если пользователь наберет на клавиатуре число 40, то строковой переменной input будет присвоена строка "40". Метод Integer .parselnt преобразовывает строку в число 40.

Когда программа вызывает метод JOptionPane. showsInputDialog, ее работу необходимо завершать вызовом метода System.exit (0). В основном это вызвано техническими причинами. Вывод на экран диалогового окна запускает новый поток управления. При завершении работы метода main этот новый поток управления не прекращает свою работу автоматически. Чтобы закрыть все потоки, нужно вызвать метод System.exit. Метод System.exit получает целочисленный параметр, представляющий собой "код выхода" из программы. По умолчанию, если работа программы завершилась нормально, ее код выхода равен 0, в противном случае этот код не равен нулю. Для индикации разных ошибочных ситуаций можно использовать разные коды выхода.

Класс JOptionPane определен в пакете javax. swing. Используя класс, не определенный в основном пикете java. lang, нужно применять директиву import.

Пример

```
import javax.swing.*;
public class InputTest
 {
public static void main(String [] args)
 {
// Первый ввод.
String name = JOptionPane.showInputDialog("Назовите Ваше имя");
 // Второй ввод.
String input = JOptionPane.showInputDialog ("Сколько Вам лет?");
 // Преобразовать строку в целое число.
int age = Integer.parseInt(input);
 // Вывести результат на консоль.
 System.out.println("Привет, " + name +
". В следующем году Вам будет " + (age + 1 ) ) ;
System.exit(0) ;
}
}
```

Операторы выбора

Условный оператор. Условный оператор имеет полную и сокращенную форму:

if (< условие>) <оператор>; //сокращенная форма

Если значение < условия> истинно, то выполняется оператор.

if (<выражение-условие>) <оператор1>; //полная форма
else <оператор2>;

Если значение < условия> истинно, то выполняется оператор1, иначе выполняется оператор2.

Пример1. Программа, для определения является ли символ двоичным.

```
class IfElse {
public static void main(String args[]) { char ch = ‘1’;
String s;
if (ch==’1’ || ch == ‘0’) {
s = "binary";
} else
s= "no binary";
System.out.println( "Simvol " + s + ".");
}
}
```

После выполнения программы будет получен следующий результат:

C: \> java IfElse
Simvol binary.

Переключатель. Переключатель определяет множественный выбор и имеет форму:

```
switch (<выражение>)
{
case <константа1> : <оператор1 >;
case <константа2> : <оператор2 >;
. . . . . .
default: <операторы>;
}
```

Если значение выражения, совпадает с константой записанной следом за case, то выполняются операторы, помеченные данной меткой и операторы всех следующих вариантов, пока не появится оператор перехода или не закончится переключатель. Если значение выражения, не совпало ни с одной константой, то выполняются операторы, которые следуют за меткой default. Метка default может отсутствовать.

программа, для определения является ли символ двоичным.

```
class SwitchDemo { public static void main(String args[]) {
char ch = ‘1’;
String s;
switch (month) {
case ‘0’: // FALLSTHROUGH
case ‘1’: // FALLSTHROUGH
s = "binary";
break;
default:
s = "no binary";
}
System.out.println("Simvol " + s + ".");
}
}
```

Операторы циклов

Цикл с предусловием. Имеет форму:

while (<условие>) <тело_цикла> ;

Если < условие> истинно, то тело цикла выполняется до тех пор пока < условие> не станет ложным.

Ниже приведен пример цикла для печати десяти строк «tick».

```
class WhileDemo {
public static void main(String args[]) {
int n = 10;
while (n > 0) {
System.out.println("tick " + n);
n--;
}
}
}
```

Цикл с постусловием. Имеет форму:

Do <тело_цикла>; while (<условие>);

Тело цикла выполняется до тех пор, пока < условие> истинно.

Ниже приведен пример цикла для печати десяти строк «tick».

```
class DoWhile {
public static void main(String args[]) {
int n = 10;
do {
System.out.println("tick " + n);
} while (--n > 0);
}
}
```

Цикл с параметром. Имеет форму:

for (<выражение_1>;< условие>;<выражение_2>)

тело_цикла;

Сначала выполняется <выражение_1> и затем пока <условие> истинно вқполняется тело цикла и <выражение_2>. Любое выражение может отсутствовать, но разделяющие их « ; » должны быть обязательно.

Следующий пример — вариант программы, ведущей отсчет.

```
class ForDemo {
public static void main(String args[]) {
for (int i = 1; i <= 10; i++)
System.out.println("i = " + i);
}
}
```

Операторы перехода

Операторы перехода выполняют безусловную передачу управления.

Оператор **break -** оператор прерывания цикла.

Т. е. оператор break целесообразно использовать, когда условие продолжения итераций надо проверять в середине цикла.

Например, в следующей программе имеется три вложенных блока, и у каждого своя уникальная метка. Оператор break, стоящий во внутреннем блоке, вызывает переход на оператор, следующий за блоком b. При этом пропускаются два оператора println.

```
class Break {
public static void main(String args[]) { boolean t = true;
a: { b: { c: {
System.out.println("Before the break"); // Перед break
if (t)
break b;
System.out.println("This won't execute"); // Не будет выполнено }
System.out.println("This won't execute"); // Не будет выполнено }
System.out.println("This is after b"); //После b
}
}
}
```

В результате исполнения программы будет получен следующий результат:

```
C:\> Java Break
Before the break
This is after b
```

Оператор **continue** - переход к следующей итерации цикла. Он используется, когда тело цикла содержит ветвления.

Как и в случае оператора break, в операторе continue можно задавать метку, указывающую, в каком из вложенных циклов вы хотите досрочно прекратить выполнение текущей итерации. Пример программа, использующая оператор continue с меткой для вывода треугольной таблицы умножения для чисел от 0 до 9:

```
class ContinueLabel {
public static void main(String args[]) {
outer: for (int i=0; i < 10; i++) {
for (int j = 0; j < 10; j++) {
if (j > i) {
System.out.println("");
continue outer;
}
System.out.print(" " + (i * j));
}
}
}
}
```

Оператор continue в этой программе приводит к завершению внутреннего цикла со счетчиком j и переходу к очередной итерации внешнего цикла со счетчиком i. В процессе работы эта программа выводит следующие строки:

```
C:\> Java ContinueLabel
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81
```

Контрольные вопросы:

1. Как указывается комментарий?
2. Дайте определение составного оператора.
3. Каким образом осуществляется считывание ввода?
4. Укажите операторы выбора.
5. Укажите виды циклов.
6. Для чего используются операторы break и continue?

Литература:

1. Смирнов Н.И. Java -2. Учебное пособие. М.:«Три Л», 2000. -320 с.
2. Арнольд К. и др. Язык программирования JAVA/ Вильямс. 2001. 624 с.
3. Брюс Эккель. Философия Java . Библиотека программиста. 2003.

ЛЕКЦИЯ 4. ТЕМА: МАССИВЫ

План: Одномерные массивы; Копирование массивов; Сортировка массивов; Многомерные массивы; Параметры командной строки

Одномерные массивы

Массив это индексированная переменная. Простейшая форма определения массива:

<тип> <имя_переменной>>[<константа_выражение>] = <инициализатор>;

Значение индекса начинается с нуля.

Для того, чтобы зарезервировать память под массив, используется специальный оператор new.

int a[];

int a=new int[6];

При инициализации массива самая левая размерность может не указываться.

Например:

double d[] = {1, 2, 3, 4, 5};

Эту синтаксическую конструкцию удобно применять для повторной инициализации массива без образования новой переменной.

Например

smallPrimes = new int{ 17, 19, 23, 29, 31, 37 };

метод length используется чтобы подсчитать количество элементов в массиве,. Например,

for (int i = 0; i < a.length; i++)

System.out.println(a[i]) ;

После создания массива изменить его размер невозможно (хотя можно, конечно,изменять отдельные его элементы).

В языке Java оператор [ ] по умолчанию проверяет диапазон изменения индексов.

Массив можно объявить двумя способами:

int [] a;

или

int a[];

Копирование массивов

Один массив можно скопировать в другой, но при этом обе переменные будут ссылаться на один и тот же массив.

int[] luckyNumbers = smailPrimes;

luckyNuimbers[5] = 12; // Теперь элемент smailPrimes[5]также равен 12.

Если необходимо скопировать все элементы одного массива в другой, следует использовать метод аггаусору из класса **java.lang.System**.

Например, показанные ниже операторы, создают два массива, а затем копируют последние четыре элемента первого массива во второй.

Копирование начинается со второй позиции в исходном массиве, а копируемые элементы помещаются в целевой массив, начиная с третьей позиции.

```
int[] smailPrimes = {2, 3, 5, 7, 11, 13};
int[] luckyNumbers = {1001, 1002, 1003, 1004, 1005, 1006, 1007};
System.arraycopy(smailPrimes, 2, luckyNumbers, 3, 4);
for (int i = 0; i < luckyNumbers.length; i++)
System.println(i +.": " + luckyNumbersfi]);
```

Выполнение этих операторов приводит к следующему результату.

```
0: 1001
1: 1002
2: 1003
3: 5
4: 7
5: 11
6: 13
```

Сортировка массива

Если нужно упорядочить массив чисел, можно применить метод sort из класса **java.util.Arrays**.

Метод упорядочивает массив с помощью алгоритма быстрой сортировки.

```
Int []a = new int[10000];
Arrays.sort(a);
```

Этот метод использует усовершенствованную версию алгоритма быстрой сортировки, которая считается наиболее эффективной для большинства множеств.

Сдедующая программа создает рабочий массив. Она генерирует случайную комбинацию чисел для лотереи.

```
import java.util.*;
import javax.swing.*;
public class LotteryDrawing
{
public static void main(String[] args)
{
String input = JOptionPane.showInputDialog
("СКОЛЬКО номеров нужно угадать?");
int к = Integer.parselnt(input);
input = JOptionPane.showInputDialog
("Чему равен наибольший из возможных номеров?");
int n = Integer.parselnt(input);
// Заполняем массив числами 1 2 3. . .п.
int[] numbers = new int[n];
```

```
for (int i = 0; i < numbers.length; i++)
numbers[i] = i + 1;
// Генерируем к чисел и помещаем их во второй массив.
int[] result = new int[к];
for (int i = 0; i < result.length; i++)
{
// Генерируем случайный индекс от 0 до n - 1.
int r = (int)(Math.random() * n);
// Помещаем элемент в случайную ячейку.
result[i] = numbers[r];
// Перемещаем последний элемент в случайную ячейку.
numbers[r] = numbers[п - 1];
n— ;
}
// Выводим на печать упорядоченный массив.
Arrays.sort(result);
System.out.println
("Поставьте на следующую комбинацию — не пожалеете!")
for (int i = 0; i < result.length; i++)
System.out.println(result[i]);
44. System.exit(0);
}
}
```

Например, если нужно выиграть "6 из 49", программа может напечатать следующее сообщение.

Поставьте на следующую комбинацию. Она сделает вас богатым!
4
7
8
19
30
44

Многомерные массивы

На самом деле, настоящих многомерных массивов в Java не существует. Зато имеются массивы массивов, которые ведут себя подобно многомерным массивам, за исключением нескольких незначительных отличий. Приведенный ниже код создает традиционную матрицу из шестнадцати элементов типа double, каждый из которых инициализируется нулем. Внутренняя реализация этой матрицы — массив массивов double.

```
double matrix [][] = new double [4][4];
```

Следующий фрагмент кода инициализирует такое же количество памяти, но память под вторую размерность отводится вручную.

```
double matrix [][] = new double [4][];
matrix [0] = new double[4];
matrix[1] = new double[4];
matrix[2] = new double[4], matrix[3] = { 0, 1, 2, 3 };
```

В других случаях, если элементы массива известны заранее, можно использовать сокращенную запись для инициализации многомерного массива, в которой не используется оператор new. Например, так.

```
int[][] magicSquare = {{16, 3, 2, },(5, 10, 11},{9, 6, 7},{4, 15, 14}};
```

Поскольку к строкам массива открыт самостоятельный доступ, их можно легко переставлять!

```
double[] temp = balance [i];
balance[i] = balance[i+1];
balance[i=l] = temp;
```

Кроме того, в языке Java легко создавать "неровные" массивы, т.е массивы, у которых разные строки имеют разную длину.

Чтобы создать неровный массив, сначала разместим в памяти массив, хранящий его строки.

```
int[][] adds = new int[NMAX+1][];
```

Затем поместим туда сами строки.

```
for (n=0; n<=NMAX; n++)
adds[n] = new int[ n + l ] ;
```

Теперь в памяти размещен весь массив, так что мы можем обращаться к его элементам, как обычно, при условии, что индексы не выходят за пределы допустимого диапазона.

```
for (n=0; n<odds.length; n++)
for (k=0; k<odds[n].length; k++)
// Вычисление шансов.
adds[n][k] = lotteryOdds;
```

Приведем стандартный пример. Создадим массив, в котором элемент, стоящий на пересечении i-й строки и j-го столбца, равен количеству возможностей "выбрать j чисел среди i лотерейных номеров".

Поскольку число j не может превышать i, получается треугольная матрица. В i-й строке этой матрицы стоит i+1 элементов

```
public class LotteryArray
{
public static void main(String[] args)
(
final int NMAX' = 10;
// Размещение треугольной матрицы.
int[][] odds = new int[NMAX + 1][];
for (int n = 0; n <= NMAX; n++)
```

```
odds[n] = new int[n + 1];
// Заполнение треугольной матрицы,
for (int n = 0; n < odds.length; n++)
for (int k = 0; k < odds[n].length; k++)
{
Вычисление биномиальных коэффициентов.
n * (n - 1) * (n - 2) * . . . * (n - k + 1)
*/
int lotteryOdds = 1;
for (int i = 1; i <= k;
lotteryOdds = lotteryOdds * (n - i + 1). / i;
odds[n][k] = lotteryOdds;
// Печать треугольной матрицы.
for (int n = 0; n < odds.length; n++)
{
for (int к = 0; к < odds[n].length; k++)
{
// Расстановка пробелов в выводе.
String output = " " + odds[n][k];
// Результат выводится в поле шириной 4 символа,
output = output.substring(output.length() - 4);
System.out.print(output);
}
System.out.println();
}
}
}
```

Параметры командной строки

Мы уже рассматривали пример программы для работы с массивами на языке Java,которая повторялась несколько раз. В каждой программе на языке Java есть метод main с параметром String [ ] args. Этот параметр означает, что метод main получает массив строк, а именно: аргументы, указанные в командной строке.

Например, рассмотрим следующую программу.

```
public class Message
public static void main(String[] args)
if (args[0].equals("-h");
System.out.print ("Здравствуй,");
else if (args[0].equals("-g"))
System.out.print("Прощай,");
// Печатает другие аргументы командной строки,
for (int i = 1; i < args.length; i++)
System.out.print(" " + args[i]);
```

```
System.out.print("!");
}
}
```

При следующем вызове программы

Java Message -g жестокий мир

массив args будет состоять из таких элементов.

args[0] "-g"

args[l] "жестокий"

args[2] "мио"

Программа выведет на печать сообщение

Прощай, жестокий мир!

В методе main программы на языке Java ее имя не хранится в массиве args. Например, после запуска программы Message с помощью команды

Java Message -h мир

из командной строки элемент args[0] будет равен "- h" , а не "Message" или "Java".

Контрольные вопросы:

1. Укажите способы инициализации массивов.
2. Каким образом массивы могут быть определены в качестве формальных параметров?
3. Для чего используются операции new?
4. Как осуществляется копирование и сортировка массивов?
5. Укажите способ формирования динамических массивов.

Литература:

1. Смирнов Н.И. Java -2. Учебное пособие. М.:«Три Л», 2000. -320 с.
2. Арнольд К. и др. Язык программирования JAVA/ Вильямс. 2001. 624 с.
3. Брюс Эккель. Философия Java . Библиотека программиста. 2003.

ЛЕКЦИЯ 5. ТЕМА: СТРОКИ

План: Работа со строками; Конструкторы; Создание строк; Слияние строк, Извлечение символов; Сравнение; Упорядочивание; Равенство; Поиск символа или подстроки.

Работа со строками

В пакет java.lang встроен класс, инкапсулирующий структуру данных, соответствующую строке. Этот класс, называемый **String**, не что иное, как объектное представление неизменяемого символьного массива. В этом классе есть методы, которые позволяют сравнивать строки, осуществлять в них поиск и извлекать определенные символы и подстроки. Класс **StringBuffer** используется тогда, когда строку после создания требуется изменять.

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

Конструкторы

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

String s = new String():

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

char chars[] = { 'a', 'b', 'c' }:
String s = new String(chars);
System.out.println(s):

Этот фрагмент кода выводит строку «abc».

Данный конструктор имеет три параметра:

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

Пример данного способа инициализации:

char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' }:
String s = new String(chars,2,3);
System.out.println(s);

Этот фрагмент выведет «cde».

Создание строк

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

String s = "abc";

System.out.println(s);

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

String s = "abc";

System.out.println(s.length);

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

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

Слияние строк

Строку

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

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

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

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

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

Последовательность выполнения операторов

Давайте еще раз обратимся к нашему последнему примеру:

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

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

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

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

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

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

Извлечение символов

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

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

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

```
C:\> java getCharsDemo
demo
```

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

Сравнение

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

```
class equalDemo {
public static void main(String args[]) {
String s1 = "Hello";
String s2 = "Hello";
String s3 = "Good-bye";
```

```
String s4 = "HELLO";
System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));
System.out.println(s1 + " equals " + s3 + " -> " + s1.equals(s3));
System.out.println(s1 + " equals " + s4 + " -> " + s1.equals(s4));
System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " +
s1.equalsIgnoreCase(s4));
} }
```

Результат запуска этого примера :

C:\> java equalsDemo

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

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

Равенство

Метод equals и оператор == выполняют две совершенно различных проверки. Если метод equal сравнивает символы внутри строк, то оператор == сравнивает две переменные-ссылки на объекты и проверяет, указывают ли они на разные объекты или на один и тот же. В очередном нашем примере это хорошо видно — содержимое двух строк одинаково, но, тем не менее, это — различные объекты, так что equals и == дают разные результаты.

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

Вот результат запуска этого примера:

C:\> java EqualsNotEqualTo

Hello equals Hello -> true
Hello == Hello -> false

Упорядочение

Зачастую бывает недостаточно просто знать, являются ли две строки идентичными. Для приложений, в которых требуется сортировка, нужно знать, какая из двух строк меньше другой. Для ответа на этот вопрос нужно воспользоваться методом compareTo класса String. Если целое значение,

возвращенное методом, отрицательно, то строка, с которой был вызван метод, меньше строки-параметра, если положительно — больше. Если же метод compareTo вернул значение 0, строки идентичны. Ниже приведена программа, в которой выполняется пузырьковая сортировка массива строк, а для сравнения строк используется метод compareTo. Эта программа выдает отсортированный в алфавитном порядке список строк.

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

Поиск символа или подстроки

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

```
class indexOfDemo {
public static void main(String args[]) {
String s = "Now is the time for all good men " +
"to come to the aid of their country " +
"and pay their due taxes.";
System.out.println(s);
System.out.println("indexOf(t) = " + s.indexOf('f'));
System.out.println("lastIndexOf(t) = " + s.lastIndexOf('f'));
System.out.println("indexOf(the) = " + s.indexOf("the"));
System.out.println("lastIndexOf(the) = " + s.lastIndexOf("the"));
System.out.println("indexOf(t, 10) = " + s.indexOf('f' , 10));
System.out.println("lastIndexOf(t, 50) = " + s.lastIndexOf('f' , 50));
System.out.println("indexOf(the, 10) = " + s.indexOf("the", 10));
```

```
System.out.println("lastlndexOf(the, 50) = " + s.lastlndexOf("the", 50));
} }
```

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

C:> java indexOfDemo

Now is the time for all good men to come to the aid of their country and pay their due taxes.
indexOf(t) = 7
lastlndexOf(t) = 87
indexOf(the) = 7
lastlndexOf(the) = 77
index0f(t, 10) = 11
lastlndex0f(t, 50) = 44
index0f(the, 10) = 44
lastlndex0f(the, 50) = 44

Контрольные вопросы:

1. Чем отличается строка от символьного массива?
2. Укажите особенности слияния строк.
3. Каким образом осуществляется извлечение символов?
4. Каким образом осуществляется сравнение строк?
5. Укажите способ поиска символа или подстроки.

Литература:

1. Смирнов Н.И. Java -2. Учебное пособие. М.:«Три Л», 2000. -320 с.
2. Арнольд К. и др. Язык программирования JAVA/ Вильямс. 2001. 624 с.
3. Брюс Эккель. Философия Java . Библиотека программиста. 2003.

ЛЕКЦИЯ 6. ТЕМА: ОСОБЕННОСТИ ИСПОЛЬЗОВАНИЯ СТРОК

План: Модификация строк при копировании; класс StringBuffer; конструкторы; Класс StringTokenizer.

Модификация строк при копировании

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

substring

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

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

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

concat

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

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

replace

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

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

toLowerCase и toUpperCase

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

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

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

trim

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

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

valueOf

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

StringBuffer

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

Конструкторы

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

```
class StringBufferDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("Hello");
System.out.println("buffer = " + sb);
System.out.println("length = " + sb.length());
System.out. println("capacity = " + sb.capacity());
} }
```

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

```
C:\> java StringBufferDemo
buffer = Hello
```

length = 5
capacity = 21

ensureCapacity

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

setLength

Если вам вдруго понадобится в явном виде установить длину строки в буфере, воспользуйтесь методом setLength. Если вы зададите значение, большее чем длина содержащейся в объекте строки, этот метод заполнит конец новой, расширенной строки символами с кодом нуль.

charAt и setCharAt

Одиночный символ может быть извлечен из объекта StringBuffer с помощью метода **charAt**. Другой метод **setCharAt** позволяет записать в заданную позицию строки нужный символ. Использование обоих этих методов проиллюстрировано в примере:

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

Вот вывод, полученный при запуске этой программы.

```
C:\> java setCharAtDemo
buffer before = Hello
charAt(1) before = e
buffer after = Hi
charAt(1) after = i
```

append

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

позволяет выстраивать в цепочку последовательные вызовы метода, как это показано в очередном примере.

```
class appendDemo {
public static void main(String args[]) {
String s;
int a = 42;
StringBuffer sb = new StringBuffer(40);
s = sb.append("a = ").append(a).append("!").toString();
System.out.println(s);
} }
```

Вот вывод этого примера:

```
C:\> Java appendDemo
a = 42!
```

insert

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

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

При запуске эта программа выводит следующую строку:

```
C:\> java insertDemo
hello there world!
```

StringTokenizer

Обработка текста часто подразумевает разбиение текста на последовательность лексем - слов (tokens). Класс StringTokenizer предназначен для такого разбиения, часто называемого лексическим анализом или сканированием. Для работы StringTokenizer требует входную строку и строку символов-разделителей. По умолчанию в качестве набора разделителей используются обычные символы-разделители: пробел, табуляция, перевод строки и возврат каретки. После того, как объект StringTokenizer создан, для последовательного извлечения лексем из входной строки используется его метод nextToken. Другой метод — hasMoreTokens — возвращает true в том случае, если в строке еще остались неизвлеченные лексемы. StringTokenizer также реализует интерфейс Enumeration, а это

значит, что вместо методов hasMoreTokens и nextToken вы можете использовать методы hasMoreElements и nextElement, соответственно.

Ниже приведен пример, в котором для разбора строки вида “ключ=значение” создается и используется объект StringTokenizer. Пары “ключ=значение” разделяются во входной строке двоеточиями.

```
import java.util.StringTokenizer;
class STDemo {
static String in = "title=The Java Handbook:" + "author=Patrick Naughton:" + "isbn=0-07-882199-1:" + "ean=9 780078 821998:" + "email=naughton@starwave. corn";
public static void main(String args[]) {
StringTokenizer st = new StringTokenizer(in, "=:");
while (st.hasMoreTokens()) {
String key = st.nextToken();
String val = st.nextToken();
System.out.println(key + "\t" + val);
}
} }
```

Контрольные вопросы:

1. Чем отличаются классы String и StringBuffer?
2. Как создаются объекты класса StringBuffer?
3. Перечислите методы класса StringBuffer.
4. Какой метод вызывается при конкатенации строк?
5. Для чего используется класс StringTokenizer?

Литература:

1. Смирнов Н.И. Java -2. Учебное пособие. М.:«Три Л», 2000. -320 с.
2. Арнольд К. и др. Язык программирования JAVA/ Вильямс. 2001. 624 с.
3. Брюс Эккель. Философия Java . Библиотека программиста. 2003.

ЛЕКЦИЯ 7. ТЕМА: КЛАССЫ И ОБЪЕКТЫ

План: Класс; Доступность компонентов класса; Вызов метода Указатель this; Конструктор; Совмещение методов; Статические компоненты класса.

Класс

С точки зрения синтаксиса, класс в JAVA - это структурированный тип, образованный на основе уже существующих типов.

В простейшем случае класс можно определить с помощью конструкции:

class <имя_класса>{<список_членов_класса>};

где

список_членов_класса – определения и описания типизированных данных и принадлежащих классу функций.

Функции – это методы класса, определяющие операции над объектом.

Данные – это поля объекта, образующие его структуру. Значения полей определяет состояние объекта.

Для описания объекта класса (экземпляра класса) используется конструкция

<имя_класса> <имя_объекта>;

Обращаться к данным объекта и вызывать функции для объекта можно с помощью конструкции:

<имя_объекта>. <имя_данного>

<имя_объекта>. <имя_функции>

В JAVA объект – это динамическая структура. Переменная-объект содержит не данные, а ссылку на данные объекта.

Доступность компонентов класса

В языке Java есть четыре модификатора доступа, управляющих областью видимости.

1. Область видимости ограничена классом (private) .
2. Область видимости не ограничена (public).
3. Область видимости ограничена пакетом и всеми подклассами (protected).
4. Область видимости ограничена пакетом (к сожалению— по умолчанию).

Вызов метода

Методы - это подпрограммы, присоединенные к конкретным определениям классов. Они описываются внутри определения класса на том же уровне, что и переменные объектов. При объявлении метода задаются тип возвращаемого им результата и список параметров. Общая форма объявления метода такова:

тип имя_метода (список формальных параметров) {

тело метода:

В Java отсутствует возможность передачи параметров *по ссылке* на примитивный тип. В Java все параметры примитивных типов передаются *по значению*, а это означает, что у метода нет доступа к исходной переменной, использованной в качестве параметра. Заметим, что все объекты передаются по ссылке, можно изменять содержимое того объекта, на который ссылается данная переменная.

Указатель this

Когда функция-член класса вызывается для обработки данных конкретного объекта, этой функции автоматически и неявно передается ссылка на тот объект, для которого функция вызвана. Этот указатель имеет имя **this**.

```
class Point { int x, y;
void init(int x, int y) {
this.x = x;
this.y = y } }
class TwoPointsInit {
public static void main(String args[]) {
Point p1 = new Point();
Point p2 = new Point();
p1.init(10,20);
p2.init(42,99);
System.out.println("x = " + p1.x + " y = •• + p-l.y);
System.out.printlnC'x = " + p2.x + " y = •• + p2.y);
} }
```

Конструктор

Для инициализации объектов класса в его определение можно явно включить специальную компонентную функцию, называемую **конструктором**.

Имя этой компонентной функции должно совпадать с именем класса. Такая функция автоматически вызывается при определении или размещении в памяти с помощью оператора new каждого объекта класса. Конструктор выделяет память для объекта и инициализирует данные - члены класса.

Формат определения конструктора следующий:

<имя_класса>(<список_форм_параметров>)
{<операторы_тела_конструктора>}

Для конструктора не определяется тип возвращаемого значения. Даже тип void не допустим.

Конструктор всегда существует для любого класса, причем, если он не определен явно, он создается автоматически. По умолчанию создается конструктор без параметров. Если конструктор описан явно, то конструктор

по умолчанию не создается. По умолчанию конструкторы создаются общедоступными (**public**).

Параметром конструктора может быть объект его собственного класса.

Без явного указания программиста конструктор всегда автоматически вызывается при определении (создании) объекта. В этом случае вызывается конструктор без параметров.

Совмещение методов

Язык Java позволяет создавать несколько методов с одинаковыми именами, но с разными списками параметров. Такая техника называется совмещением методов (**method overloading**). Методы, использующие совмещение имен, не обязательно должны быть конструкторами.

Пример:

```
class Point { int x, y;
Point(int x, int y) {
this.x = x;
this. y = y;
}
Point() {
x = 0;
y = 0;
}
double distance(int x, int y) {
int dx = this.x - x;
int dy = this.y - y;
return Math.sqrt(dx*dx + dy*dy);
}
double distance(Point p) {
return distance(p.x, p.y);
}
}
class PointDist {
public static void main(String args[]) {
Point p1 = new Point(0, 0);
Point p2 = new Point(30, 40);
System.out.println("p1 = " + pi.x + ", " + p1.y);
System.out.println("p2 = " + p2.x + ", " + p2.y);
System.out.println("p1.distance(p2) = " + p1.distance(p2));
System.out.println("p1.distance(60, 80) = " + p1.distance(60, 80));
}}
```

Ниже приведен результат работы этой программы:

C:\> java PointDist

p1 = 0, 0

p2 = 30, 40
p1.distance(p2) = 50.0
p1.distance(60, 80) = 100.0

finalize

В Java существует возможность объявлять методы с именем **finalize**. Методы finalize аналогичны деструкторам в C++ (ключевой знак **~)** и Delphi (ключевое слово **destructor**). Исполняющая среда Java будет вызывать его каждый раз, когда сборщик мусора соберется уничтожить объект этого класса.

Статические компоненты класса

В классах возможно определение компонент, которые являются общими для всех объектов данного класса. Такие компоненты должны быть определены в классе, как **статические** (**static**). Статические данные классов не дублируются при создании объектов, т.е. каждый статический компонент существует в единственном экземпляре.

Статические методы могут непосредственно обращаться только к другим статическим методам, в них ни в каком виде не допускается использование ссылок this. Переменные также могут иметь тип static, они подобны глобальным переменным, то есть доступны из любого места кода. Внутри статических методов недопустимы ссылки на нестатические переменные.

Доступ к статическим компонентам возможен не только через имя объекта, но и через имя класса

<имя_класса> . <имя_компонента>

Однако так можно обращаться только к ***public*** компонентам.

Для обращения к собственной статической компоненте извне можно использовать **статические компоненты-функции**. Эти функции можно вызвать через имя класса.

<имя_класса> . <имя_статической_функции>

Пример.

```
class StaticClass {
static int a = 42;
static int b = 99;
static void callme() {
System.out.println("a = " + a);
} }
class StaticByName {
public static void main(String args[]) {
StaticClass.callme();
```

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

А вот и результат запуска этой программы:

C:\> Java StaticByName

a = 42 b = 99

Контрольные вопросы:

1. Как определяется класс?
2. Можно ли перегружать функции-члены? Приведите примеры.
3. Объясните назначение конструкторов.
4. Можно ли создавать объекты массивов? Какие конструкторы вызываются при их создании?
5. Могут ли статические члены класса быть типа **private**? Как инициализировать такие члены-данные?

Литература:

1. Смирнов Н.И. Java -2. Учебное пособие. М.:«Три Л», 2000. -320 с.
2. Арнольд К. и др. Язык программирования JAVA/ Вильямс. 2001. 624 с.
3. Брюс Эккель. Философия Java . Библиотека программиста. 2003.

ЛЕКЦИЯ 8. ТЕМА: ОСОБЕННОСТИ ИСПОЛЬЗОВАНИЯ КЛАССОВ И ОБЪЕКТОВ

План: Runtime; Управление памятью; Выполнение других программ; Класс System; Свойства окружения; Класс Date; Класс Math; Класс Random.

Runtime

Класс Runtime инкапсулирует интерпретатор Java. Вы не можете создать нового представителя этого класса, но можете, вызвав его статический метод, получить ссылку на работающий в данный момент объект Runtime. Обычно апплеты и другие непривелигированные программы не могут вызвать ни один из методов этого класса, не возбудив при этом исключения SecurityException. Одна из простых вещей, которую вы можете проделать с объектом Runtime — его останов, для этого достаточно вызвать метод exit(int code).

Управление памятью

Хотя Java и представляет собой систему с автоматической сборкой мусора, вы для проверки эффективности своего кода можете захотеть узнать, каков размер “кучи” и как много в ней осталось свободной памяти. Для получения этой информации нужно воспользоваться методами totalMemory и freeMemory.

При необходимости вы можете “вручную” запустить сборщик мусора, вызвав метод gc. Если вы хотите оценить, сколько памяти требуется для работы вашему коду, лучше всего сначала вызвать gc, затем free-Memory, получив тем самым оценку свободной памяти, доступной в системе. Запустив после этого свою программу и вызвав freeMemory внутри нее, вы увидите, сколько памяти использует ваша программа.

Выполнение других программ

В безопасных средах вы можете использовать Java для выполнения других полновесных процессов в своей многозадачной операционной системе. Несколько форм метода exec позволяют задавать имя программы и ее параметры.

В очередном примере используется специфичный для Windows вызов exec, запускающий процесс notepad — простой текстовый редактор. В качестве параметра редактору передается имя одного из исходных файлов Java. Обратите внимание — exec автоматически преобразует в строке-пути символы "/" в разделители пути в Windows — "\".

```
class ExecDemo {
public static void main(String args[]) {
Runtime r = Runtime. getRuntime();
Process p = null;
```

```
String cmd[] = { "notepad", "/java/src/java/lang/Runtime.java" };
try {
p = r.exec(cmd);
} catch (Exception e) {
System.out.println("error executing " + cmd[0]);
}
} }
```

System

Класс System содержит любопытную коллекцию глобальных функций и переменных. В большинстве примеров для операций вывода мы использовали метод System.out.println().

Метод **currentTimeMillis** возвращает текущее системное время в виде миллисекунд, прошедших с 1 января 1970 года.

Метод **arraycopy** можно использовать для быстрого копирования массива любого типа из одного места в памяти в другое. Ниже приведен пример копирования двух массивов с помощью этого метода.

```
class ACDemo {
static byte a[] = { 65, 66, 67, 68, 69, 70, 71, 72, 73, 74 };
static byte b[] = { 77, 77, 77, 77, 77, 77, 77, 77, 77, 77 };
public static void main(
String args[]) {
System.out.println("a = " + new String(a, 0));
System.out.println("b = " + new String(b, 0));
System.arraycopy(a, 0, b, 0, a.length);
System.out.println("a = " + new String(a, 0));
System.out.println("b = " + new String(b, 0));
System.arraycopy(a, 0, a, 1, a.length - 1);
System.arraycopy(b, 1, b, 0, b.length - 1);
System.out.println("a = " + new String(a, 0));
System.out.println("b = " + new String(b, 0));
} }
```

Как вы можете заключить из результата работы этой программы, копирование можно выполнять в любом направлении, используя в качестве источника и приемника один и тот же объект.

```
C:\> java ACDemo
a = ABCDEFGHIJ
b = MMMMMMMMM
a = ABCDEFGHIJ
b = ABCDEFGHIJ
a = AABCDEFGHI
b = BCDEFGHIJJ
```

Свойства окружения

Исполняющая среда Java предоставляет доступ к переменным окружения через представителя класса Properties, с которым можно работать с помощью метода System.getProperty. Для получения полного списка свойств можно вызвать метод System.getProperties() или см. таблицу.

Таблица

Стандартные системные свойства

Имя	**Значение**	**Доступ для апплета**
Java.version	Версия интерпретатора Java	да
Java.vendor	Строка идентификатора, заданная разработчиком	да
java.vendor.url	URL разработчика	да
java.class.version	Версия Java API	да
java.class.path	Значение переменной CLASSPATH	нет
java.home	Каталог, в котором инсталлирована среда Java	нет
java.compiler	Компилятор JIT	нет
os.name	Название операционной системы	да
os.arch	Архитектура компьютера, на котором выполняется программа	да
os.version	Версия операционной системы Web-узла	да
file.separator	Зависящие от платформы разделители файлов (/ или \)	да
path.separator	Зависящие от платформы разделители пути (: или ;)	да
line.separator	Зависящие от платформы разделители строк (\n или \r\n)	да
user.name	Имя текущего пользователя	нет
user.home	Домашний каталог пользователя	нет
user.dir	Текущий рабочий каталог	нет
user.language	2-символьный код языка для местности по умолчанию	нет
user.region	2-символьный код страны для местности по умолчанию	нет
user.timezone	Временной пояс по умолчанию	нет
user.encoding	Кодировка сиволов для местности по умолчанию	нет
user.encoding.pkg	Пакет, содержащий конверторы для преобразования символов из местной	нет

	кодировки в Unicode	

Date

Класс Date используется для операций с датой и временем. Через него вы можете получить доступ к дате, месяцу, году, дню недели, часам, минутам, секундам. У объектов этого класса — несколько конструкторов. Самый простой — Date() — инициализирует объект текущими датой и временем. Три остальных конструктора предлагают дополнительные возможности задавать начальные значения для нового объекта.

- Date(year, month, date) — устанавливает указанную дату, при этом время устанавливается в 00:00:00 (полночь).
- Date(year, month, date, hours, minutes) — устанавливает указанные дату и время, секунды устанавливаются в 0.
- Date(year, month, date, hours, minutes, seconds) — наиболее полное задание времени, в объекте устанавливаются указанные дата и время, в том числе и секунды.

get и set

Класс Date включает в себя набор методов для получения и установки отдельных атрибутов, хранящихся в объекте. Каждая из функций семейства get — getYear, getMonth, getDate, getDay, getHours, getMi-nutes и getSeconds — возвращает целое значение. Каждой из функций семейства set — setYear, setMonth, setDate, setHours, setMinutes и setSeconds — в качестве параметра передается целое значение. Вы также можете получить представление объекта Date в виде значения типа long с помощью метода getTime. Возвращаемое этим методом значение представляет собой число миллисекунд, прошедших после 1 января 1970 года.

Сравнение

Если у вас есть два объекта типа Date, и вы хотите их сравнить, то можете преобразовать хранящиеся в них даты в значения типа long, и сравнить полученные даты, выраженные в миллисекундах. Класс Date включает в себя три метода, которые можно использовать для прямого сравнения дат: — before, after и equals. Например, вызов

new Date(96, 2, 18).before(new Date(96, 2, 12)

возвращает значение true, поскольку 12-й день месяца предшествует 18-му.

Строки и часовые пояса

Объекты Date можно конвертировать в текстовые строки различных форматов. Прежде всего, обычный метод toString преобразует объект Date в строку, которая выглядит, как “Thu Feb 15 22:42:04 1996”. Метод toLocaleString преобразует дату в более короткую строку, выглядящую примерно так: “02/15/96 22:42:04”. И, наконец, метод toGMTString возвращает дату в формате среднего времени по Гринвичу: “16 Feb 1996 06:42:04 GMT”.

Math

Класс Math содержит функции с плавающей точкой, которые используются в геометрии и тригонометрии. Кроме того, в нем есть две константы, используемые в такого рода вычислениях: — E (приблизительно 2.72) и PI (приблизительно 3.14159).

Тригонометрические функции.

Приведенные ниже три функции имеют один параметр типа double, представляющий собой угол в радианах, и возвращают значение соответствующей тригонометрической функции.

- sin(double a) возвращает синус угла a, заданного в радианах.
- cos(double a) возвращает косинус угла a, заданного в радианах.
- tan(double a) возвращает тангенс угла a, заданного в радианах.

Следующие четыре функции возвращают угол в радианах, соответствующий значению, переданному им в качестве параметра.

- asin(double r) возвращает угол, синус которого равен r.
- acos(double r) возвращает угол, косинус которого равен r.
- atan(double r) возвращает угол, тангенс которого равен r.
- atan2(double a, double b) возвращает угол, тангенс которого равен отношению a/b.

Степенные, показательные и логарифмические функции

- pow(double y, double x) возвращает y, возведенное в степень x. Так, например, pow(2.0, 3.0) равно 8.0.
- exp(double x) возвращает e в степени x.
- log(double x) возвращает натуральный логарифм x.
- sqrt(double x) возвращает квадратный корень x.

Округление

- ceil(double a) возвращает наименьшее целое число, значение которого больше или равно a.
- floor(double a) возвращает наибольшее целое число, значение которого меньше или равно a.
- rint(double a) возвращает в типе double значение a с отброшенной дробной частью.
- round(float a) возвращает округленное до ближайшего целого значение a.
- round(double a) возвращает округленное до ближайшего длинного целого значение a.

Кроме того, в классе Math имеются полиморфные версии методов для получения модуля, нахождения минимального и максимального значений, работающие с числами типов int, long, float и double:

- abs(a) возвращает модуль (абсолютное значение) a.
- max(a, b) возвращает наибольший из своих аргументов.
- min(a, b) возвращает наименьший из своих аргументов.

Random

Класс Random — это генератор псевдослучайных чисел. Используемый в нем алгоритм был взят из раздела 3.2.1 “Искусства программирования” Дональда Кнута. Обычно в качестве начального значения используется текущее время, что снижает вероятность получения повторяющихся последовательностей случайных чисел.

Из объекта класса Random можно извлекать 5 типов случайных чисел. Метод nextInt возвращает целое число, равномерно распределенное по всему диапазону этого типа. Аналогично, метод nextLong возвращает случайное число типа long. Методы nextFloat и nextDouble возвращают случайные числа соответственно типов float и double, равномерно распределенные на интервале 0.0..1.0. И, наконец, метод nextGaussian возвращает нормально распределенное случайное число со средним значением 0.0 и дисперсией 1.0.

Контрольные вопросы:

1.Как определить размер свободной памяти?
2.Как получить доступ к переменным окружения?
3.Какой класс служит для работы с датой и временем?
4.Укажите методы математического класса.
5. Какой класс служит для генерации случайных чисел.

Литература:

1. Смирнов Н.И. Java -2. Учебное пособие. М.:«Три Л», 2000. -320 с.
2. Арнольд К. и др. Язык программирования JAVA/ Вильямс. 2001. 624 с.
3. Брюс Эккель. Философия Java . Библиотека программиста. 2003.

ЛЕКЦИЯ 9. ТЕМА: НАСЛЕДОВАНИЕ В КЛАССАХ

План: Производный класс; Наследование конструкторов; Виртуальные методы; Абстрактные классы.

Производный класс

Наследование дает возможность объявить производный класс, который наследует свойства, данные, методы и события всех своих предшественников в иерархии классов, а также может объявлять новые характеристики и перегружать некоторые из наследуемых функций.

В JAVA непосредственный родитель класса называется его суперклассом.

Обобщенный синтаксис объявления производного класса:

class <имя класса> extends <имя родительского класса> {…}

Наследование конструкторов

Поскольку конструкторы не наследуются, при создании производного класса наследуемые им данные-члены должны инициализироваться конструктором базового класса. Конструктор базового класса вызывается автоматически и выполняется до конструктора производного класса. Параметры конструктора базового класса указываются в определении конструктора производного класса с помощью ключевого слова **super**.

Пример:

```
class Point3D extends Point { int z;
Point3D(int x, int y, int z) {
super(x, y); // Здесь мы вызываем конструктор суперкласса this.z=z;
public static void main(String args[]) {
Point3D p = new Point3D(10, 20, 30);
System.out.println( " x = " + p.x + " y = " + p.y +
" z = " + p.z);
} }
```

Вот результат работы этой программы:

```
C:\> java Point3D
x = 10 y = 20 z = 30
```

Объекты класса конструируются снизу вверх: сначала базовый, потом компоненты-объекты (если они имеются), а потом сам производный класс.

Таким образом порядок уничтожения объекта противоположен по отношению к порядку его конструирования.

Виртуальные методы

К механизму виртуальных функций обращаются в тех случаях, когда в каждом производном классе требуется свой вариант некоторой компонентной функции. Классы, включающие такие функции, называются **полиморфными** и играют особую роль в ООП.

Виртуальные функции предоставляют механизм **позднего** (**отложенного**) или **динамического связывания.**

При позднем связывании адреса определяются динамически во время выполнения программы, а не статически во время компиляции, как в традиционных компилируемых языках, в которых применяется *раннее связывание*.

Виртуальность наследуется. После того как функция определена как виртуальная, ее повторное определение в производном классе (с тем же самым прототипом) создает в этом классе новую виртуальную функцию.

В JAVA все методы являются виртуальными.

Пример.

Давайте в качестве примера рассмотрим два класса, у которых имеют простое родство подкласс / суперкласс, причем единственный метод суперкласса замещен в подклассе.

```
class A { void callme() {
System.out.println("Inside A's callrne method");
} }
class B extends A { void callme() {
System.out.println("Inside B's callme method");
}
 }
class Dispatch {
public static void main(String args[]) {
A a = new B();
a.callme();
}
}
```

Обратите внимание — внутри метода main мы объявили переменную a класса A, а проинициализировали ее ссылкой на объект класса B. В следующей строке мы вызвали метод callme. При этом транслятор проверил наличие метода callme у класса A, а исполняющая система, увидев, что на самом деле в переменной хранится представитель класса B, вызвала не метод класса A, а callme класса B. Ниже приведен результат работы этой программы:

C:\> Java Dispatch

Inside B's calime method

В примере используется механизм, который называется *динамическим назначением методов* (**dynamic method dispatch**).

Замещение методов

Новый подкласс Point3D класса Point наследует реализацию метода distance своего суперкласса (пример PointDist.java). Проблема заключается в том, что в классе Point уже определена версия метода distance(mt x, int y), которая возвращает обычное расстояние между точками на плоскости. Мы должны *заместить* (**override**) это определение метода новым, пригодным для случая трехмерного пространства. В следующем примере проиллюстрировано и *совмещение* (overloading), и *замещение* (overriding) метода distance.

```
class Point { int x, y;
Point(int x, int y) {
this.x = x;
this.y = y;
}
double distance(int x, int y) {
int dx = this.x - x;
int dy = this.y - y:
return Math,sqrt(dx*dx + dy*dy);
}
double distance(Point p) {
return distance(p.x, p.y);
}
}
class Point3D extends Point { int z;
Point3D(int x, int y, int z) {
super(x, y);
this.z = z;
(
double distance(int x, int y, int z) {
int dx = this.x - x;
int dy = this.y - y;
int dz = this.z - z;
return Math.sqrt(dx*dx + dy*dy + dz*dz);
}
double distance(Point3D other) {
return distance(other.x, other.y, other.z);
}
double distance(int x, int y) {
double dx = (this.x / z) - x;
double dy = (this.y / z) - y;
return Math.sqrt(dx*dx + dy*dy);
}
}
```

```
class Point3DDist {
public static void main(String args[]) {
Point3D p1 = new Point3D(30, 40, 10);
Point3D p2 = new Point3D(0, 0, 0);
Point p = new Point(4, 6);
System.out.println("p1 = " + p1.x + ", " + p1.y + ", " + p1.z);
System.out.println("p2 = " + p2.x + ", " + p2.y + ", " + p2.z);
System.out.println("p = " + p.x + ", " + p.y);
System.out.println("p1.distance(p2) = " + p1.distance(p2));
System.out.println("p1.distance(4, 6) = " + p1.distance(4, 6));
System.out.println("p1.distance(p) = " + p1.distance(p));
} }
```

Ниже приводится результат работы этой программы:

C:\> Java Point3DDist

p1 = 30, 40, 10
p2 = 0, 0, 0
p = 4, 6
p1.distance(p2) = 50.9902
p1.distance(4, 6) = 2.23607
p1.distance(p) = 2.23607

Обратите внимание — мы получили ожидаемое расстояние между трехмерными точками и между парой двумерных точек. В примере используется механизм, который называется *динамическим назначением методов* (**dynamic method dispatch**).

final

Все методы и переменные объектов могут быть замещены по умолчанию. Если же вы хотите объявить, что подклассы не имеют права замещать какие-либо переменные и методы вашего класса, вам нужно объявить их как final (в Delphi / C++ не писать слово **virtual**).

final int FILE_NEW = 1;

По общепринятому соглашению при выборе имен переменных типа final — используются только символы верхнего регистра (т.е. используются как аналог препроцерных констант C++). Использование final-методов порой приводит к выигрышу в скорости выполнения кода — поскольку они не могут быть замещены, транслятору ничто не мешает заменять их вызовы *встроенным* (in-line) кодом (байт-код копируется непосредственно в код вызывающего метода).

Абстрактные классы

Абстрактным классом называется класс, в котором есть хотя бы один абстрактный метод. Абстрактным называется компонентная функция, которая имеет следующее определение:

abstract<тип> <имя > (<список_формальных_параметров>);

Объекты такого класса создать нельзя. Абстрактный класс может использоваться только в качестве базового для производных классов.
Любой класс, содержащий методы abstract, также должен быть объявлен, как abstract. Поскольку у таких классов отсутствует полная реализация, их представителей нельзя создавать с помощью оператора new. Кроме того, нельзя объявлять абстрактными конструкторы и статические методы. Любой подкласс абстрактного класса либо обязан предоставить реализацию всех абстрактных методов своего суперкласса, либо сам должен быть объявлен абстрактным.

```
abstract class A {
abstract void callme();
void metoo() {
System.out.println("Inside A's metoo method");
} }
class B extends A {
void callme() {
System.out.println("Inside B's callme method");
} }
class Abstract {
public static void main(String args[]) {
A a = new B():
a.callme():
a.metoo():
} }
```

В нашем примере для вызова реализованного в подклассе класса А метода callme и реализованного в классе А метода metoo используется динамическое назначение методов, которое мы обсуждали раньше.

C:\> Java Abstract

Inside B's callrne method Inside A's metoo method

Контрольные вопросы:

1. Почему при вызове конструкторов производного класса соблюдается такой порядок, при котором сначала вызываются конструкторы базовых классов?
2. Что такое динамическое связывание методов?
3. Чем отличается замещение от совмещения методов?
4. Как используется наследование при построении библиотек?
5. Как используются абстрактные классы?

Литература:

1. Смирнов Н.И. Java -2. Учебное пособие. М.:«Три Л», 2000. -320 с.
2. Арнольд К. и др. Язык программирования JAVA/ Вильямс. 2001. 624 с.
3. Брюс Эккель. Философия Java . Библиотека программиста. 2003.

ЛЕКЦИЯ 10. ТЕМА: ОСОБЕННОСТИ ИПОЛЬЗОВАНИЯ НАСЛЕДОВАНИЯ

План: Глобальный суперкласс; Методы equals и toString; Обобщенное программирование.

Object: глобальный суперкласс

Класс Object является предком всех классов— каждый класс в языке Java расширяет класс Object. Однако писать class Employee extends Oblects не нужно. Если суперкласс явно не указан, им считается класс Object. Поскольку в языке Java каждый класс расширяет класс Object, очень важно знать, какими возможностями обладает сам класс Object.

Переменную типа Object можно использовать в качестве ссылки на объект любого типа:

Object obj = new Employee("Гарри Хакер", 35000);

Разумеется, переменная этого класса полезна лишь в качестве настраиваемой переменной, предназначенной для хранения значений произвольного типа. Чтобы сделать с этим значением что-то конкретное, нужно знать его исходный тип, а затем выполнить приведение типов:

Employee e = (Employee) obj;

Методы equals и toString

Метод equals класса Object проверяет, одинаковы ли два объекта. Поскольку метод equals реализован в классе Object, он определяет лишь, занимают ли они одну и ту же область памяти. Это практически бесполезная проверка. Если нужно на самом деле убедиться, что объекты эквивалентны, необходимо заменить метод equals более осмысленным методом сравнения. Вот рецепт для создания более совершенного метода equals.

1. Вызвать явный параметр otherObject — впоследствии его тип нужно будет привести к типу другой переменной, которую назовем other.

2. Проверить, идентичны ли ссылки this и otherObject:

i f (this == otherObject) return true;

На практике чаще всего так и делают. Проще проверить идентичность ссылок,чем сравнивать поля объектов.

3. Проверить, является ли ссылка otherObject нулевой (null). Если да, вернуть значение false. Эту проверку делать нужно обязательно.

if (otherObject == null) return false;

4. Проверить, принадлежат ли объекты this и other одному и тому же классу.

Эта проверка является обязательной по "правилу симметричности".

if (getClass() != otherObject.getClass()) return false;

5. Преобразовать объект otherOb j ect в переменную требуемого класса:

ClassName other = (ClassName)otherObject;

6. Теперь нужно сравнить между собой все поля. Для полей основных типов используется оператор ==, для объектных полей — метод equals. Если все поля двух объектов совпадают друг с другом, возвращается значение true, в противном случае — значение false.

return fieldl == other.fieldl

&& field.2. equals (other . field2)

Например, так.

```
class Employee{
public boolean equals(Object otherObject) {
// Быстрая проверка идентичности объектов,
if (this == otherObject) return true;
// Если явный параметр — null, возвращается значение false,
if (otherObject == null) return false;
// Если классы не совпадают, они не эквивалентны.
if (getClass () != otherObject.getClassO) return false;
// Теперь мы знаем, что объект otherObject
// имеет тип Employee и не является нулевым.
Employee other = (Employee) otherObject;
// Проверим, хранятся ли в полях объектов идентичные значения,
return name.equals(other.name)
&& salary = other.salary
&& hireDay.equals(other.hireDay);
}
}
```

Метод getClass определяет тип объекта. Чтобы объекты были равны между собой, они должны для начала быть объектами одного и того же класса.

Внутри подкласса нужно сначала вызвать метод equals из суперкласса. Если эта проверка возвращает значение false, значит, объекты не совпадают. Если же проверка завершается успешно, можно приступать к сравнению полей подкласса.

Например, так.

```
class Manager extends Employee
{
public boolean.equals(Object otherObject)
{
if (!super.equals(otherObject)) return false;
Manager.other = (Manager)otherObject;
// Метод super.equals проверяет, принадлежат ли
// объекты this и otherObject одному и тому же классу.
return bonus == other.bonus;
} }
```

Другим важным методом класса Object является метод toString, возвращающий значение объекта в виде строки. Этот метод замещается почти во всех классах и предназначен для вывода на печать текущего состояния объекта.

Большинство (но не все) методы toString состоят из имени класса, за которым указываются значения его полей в квадратных скобках. Ниже приведена реализация метода toString для класса Employee.

```
public String toString()
{
return "Employee[name" + name
+ ",salary =" + salary
+ ",hireDay =" + hireDay
}
```

На самом деле этот метод можно усовершенствовать. Не будем встраивать имя класса в метод toString , а просто вызовем метод getClass () . getName () и получим строку, содержащую имя класса.

```
public String toString()
{
return getClass().geName()
+ "[name=" + name
+ ",salary=" + salary
+ ",hireday=" + hireDay
}
```

Теперь метод toString работает и с подклассами.

Разумеется, программист, создающий подкласс, должен определить свой собственный метод toString и добавить поля подкласса. Если в суперклассе используется вызов getClass () . getNamef), подкласс просто вызывает метод super.ToString ( ). Вот пример метода toString из класса Manager.

```
class manager extends Employee
{
public String toString()
{
return super.toString()+ "[bonus=" + bonus
```

Теперь состояние объекта класса Manager выводится на печать следующим образом:

Manager[name=...,salary=...,hireDay=...][bonus=...]

Метод toString вездесущ. Для этого есть важная причина: как только объект конкатенируется со строкой с помощью оператора "+", компилятор языка Java автоматически вызывает метод toString, чтобы получить представление его текущего состояния.

Допустим, что x — произвольный объект, и программист вызывает метод System.out.println(x);

В этом случае метод println просто вызовет метод x.toString() и выведет строку результата.

Метод toString , определенный в классе Object, выводит на печать имя класса и адрес объекта. Например, при вызове

System.out.println(System.out);

возникает следующая строка

java.io.PrintStream@2f668 4

Это происходит потому, что разработчики класса PrintStream не позаботились о замещении метода toString .

Этот метод — отличное средство отладки. Во многих классах из стандартной библиотеки метод toString определен так, чтобы с его помощью можно было получать полезную отладочную информацию. Некоторые отладчики позволяют вызывать метод toString для отображения на экране текущего состояния объектов, поэтому при трассировке программы можно всегда пользоваться выражениями

System.out.println{"Текущее положение = " + position);

Обобщенное программирование

В переменных типа Object могут храниться любые значения переменных любого класса, в частности, класса String:

Object obj = "Привет"; // Правильно.

Однако числа, символы и булевские переменные не являются объектами.

obj = 5 ; // Неправильно.

obj = false; // Неправильно.

Более того, все типы массивов, независимо от того, хранятся в них объекты или переменные основных типов, относятся к классам, производным от класса Object.

Employee staff [] = new Employee[10];

Object arr = staff; // Правильно.

arr = new int[10]; // Правильно.

Массив объектов, относящихся к какому-либо классу, можно преобразовать в массив объектов класса Object . Например, массив класса Employee[] можно передать в качестве параметра методу, ожидающему массив класса Object[ ]. Это преобразование весьма полезно для обобщенного программирования (generic programming).

Вот простой пример, иллюстрирующий концепцию обобщенного программирования. Допустим, вы хотите определить индекс элемента в массиве. Это ситуация, в которой можно применять обобщенное программирование, используя вместо переменных любого класса переменные типа Object .

static int find(Object[] a, Object key)
(int i;
For

```
 (i =0; i < a.length; i++)
if (a[i].equals(key)) return i;
return -1; // Индекс не найден.
}
```

Например,

```
Employee staff[] = new Employee[10];
Employee harry;
int n = find(staff, harry);
```

Заметим, что в массив типа Object[ ] можно преобразовать лишь массив объектов некоего класса. Преобразовать массив типа int[ ] в массив типа Obj ect[] невозможно. (Однако, как указывалось ранее, элементы обоих массивов можно преобразовать в элементы типа Object.)

Если массив, состоящий из объектов некоего класса, преобразовывается в массив типа Object[ ], обобщенный массив продолжает хранить информацию о своем исходном типе на протяжении всего времени выполнения программы. Поместить в этот массив элемент постороннего типа не удастся.

Контрольные вопросы:

1. Наследниками, какого класса являются все классы?
2. Укажите методы глобального суперкласса.
3. Как создается сосбственный метод сравнения?
4. Для чего используется метод toString?
5. Объясните сущность обобщенного программирования.

Литература:

1. Смирнов Н.И. Java -2. Учебное пособие. М.:«Три Л», 2000. -320 с.
2. Арнольд К. и др. Язык программирования JAVA/ Вильямс. 2001. 624 с.
3. Брюс Эккель. Философия Java . Библиотека программиста. 2003.

ЛЕКЦИЯ 11. ТЕМА: ИНТЕРФЕЙСЫ

План: Определение интерфейса; Оператор implements; Переменные в интерфейсах.

Определение интерфейса

Интерфейс — это явно указанная спецификация набора методов, которые должны быть представлены в классе, который реализует эту спецификацию. Реализация же этих методов в интерфейсе отсутствует. Подобно абстрактным классам интерфейсы обладают замечательным дополнительным свойством — их можно многократно наследовать. Конкретный класс может быть наследником лишь одного суперкласса, но зато в нем может быть реализовано неограниченное число интерфейсов.

Интерфейсы

Интерфейсы Java созданы для поддержки динамического выбора (resolution) методов во время выполнения программы. Интерфейсы похожи на классы, но в отличие от последних у интерфейсов нет переменных представителей, а в объявлениях методов отсутствует реализация. Класс может иметь любое количество интерфейсов. Все, что нужно сделать — это реализовать в классе полный набор методов всех интерфейсов. Сигнатуры таких методов класса должны точно совпадать с сигнатурами методов реализуемого в этом классе интерфейса. Интерфейсы обладают своей собственной иерархией, не пересекающейся с классовой иерархией наследования. Это дает возможность реализовать один и тот же интерфейс в различных классах, никак не связанных по линии иерархии классового наследования. Именно в этом и проявляется главная сила интерфейсов. Интерфейсы являются аналогом механизма множественного наследования в C++, но использовать их намного легче.

Оператор interface

Определение интерфейса сходно с определением класса, отличие состоит в том, что в интерфейсе отсутствуют объявления данных и конструкторов. Общая форма интерфейса приведена ниже:

```
interface имя {
тип_результата имя_метода1(список параметров);
тип имя_final1-переменной = значение;
}
```

Обратите внимание — у объявляемых в интерфейсе методов отсутствуют операторы тела. Объявление методов завершается символом ; (точка с запятой). В интерфейсе можно объявлять и переменные, при этом они неявно объявляются final - переменными. Это означает, что класс реализации не может изменять их значения. Кроме того, при объявлении переменных в интерфейсе их обязательно нужно инициализировать константными значениями. Ниже приведен пример определения интерфейса,

содержащего единственный метод с именем callback и одним параметром типа int.

```
interface Callback {
void callback(int param);
}
```

Оператор implements

Оператор implements — это дополнение к определению класса, реализующего некоторый интерфейс(ы).

class имя_класса [extends суперкласс]
[implements интерфейс0 [, интерфейс1...]] { тело класса }

Если в классе реализуется несколько интерфейсов, то их имена разделяются запятыми. Ниже приведен пример класса, в котором реализуется определенный нами интерфейс:

```
class Client implements Callback {
void callback(int p) {
System.out.println("callback called with " + p);
} }
```

В очередном примере метод callback интерфейса, определенного ранее, вызывается через переменную - ссылку на интерфейс:

```
class TestIface {
public static void main(String args[]) { Callback c = new client();
c.callback(42);
} }
```

Ниже приведен результат работы программы:

C:\> Java TestIface
callback called with 42

Переменные в интерфейсах

Интерфейсы можно использовать для импорта в различные классы совместно используемых констант. В том случае, когда вы реализуете в классе какой-либо интерфейс, все имена переменных этого интерфейса будут видимы в классе как константы. Это аналогично использованию файлов-заголовков для задания в C и C++ констант с помощью директив #define или ключевого слова const в Pascal / Delphi.

Если интерфейс не включает в себя методы, то любой класс, объявляемый реализацией этого интерфейса, может вообще ничего не реализовывать. Для импорта констант в пространство имен класса предпочтительнее использовать переменные с модификатором final. В приведенном ниже примере проиллюстрировано использование интерфейса для совместно используемых констант.

```
import java.util.Random;
interface SharedConstants { int NO = 0;
```

```
int YES = 1;
int MAYBE = 2;
int LATER = 3;
int SOON = 4;
int NEVER = 5; }
class Question implements SharedConstants {
Random rand = new Random();
int ask() {
int prob = (int) (100 * rand.nextDouble());
if (prob < 30) return NO; // 30%
else if (prob < 60) return YES; // 30%
else if (prob < 75) return LATER; // 15%
else if (prob < 98) return SOON; // 13%
else return NEVER; // 2% } }
class AskMe implements SharedConstants {
static void answer(int result) {
switch(result) {
case NO:
System.out.println("No");
break;
case YES:
System.out.println("Yes");
break;
case MAYBE:
System.out.println("Maybe");
break;
case LATER:
System.out.println("Later");
break;
case SOON:
System.out.priniln("Soon");
break;
case NEVER:
System.out.println("Never");
break;
} }
public static void main(String args[]) {
Question q = new Question();
answer(q.ask());
answer(q.ask());
answer(q.askO);
answer(q.ask()); } }
```

Обратите внимание на то, что результаты при разных запусках программы отличаются, поскольку в ней используется класс генерации случайных чисел Random пакета java.util.

C:\> Java AskMe
Later
Scon
No
Yes

Контрольные вопросы:

1. Приведите определение интерфейса.
2. Чем отличается интерфейс от класса?
3. Для чего используется оператор implements?
4. Объясние назначение интерфейса.
5. Как используется интерфейс для импорта в различные классы совместно используемых констант?

Литература:

1. Смирнов Н.И. Java -2. Учебное пособие. М.:«Три Л», 2000. -320 с.
2. Арнольд К. и др. Язык программирования JAVA/ Вильямс. 2001. 624 с.
3. Брюс Эккель. Философия Java . Библиотека программиста. 2003.

ЛЕКЦИЯ 12. ТЕМА: ПАКЕТЫ

План: Определение пакета*;* Пакеты*;* Оператор package*;* Трансляция классов в пакетах*;* Оператор import; Ограничение доступа.

Определение пакета

Пакет (**package**) — это некий контейнер, который используется для того, чтобы изолировать имена классов. Например, вы может е создать класс List, заключить его в пакет и не думать после этого о возможных конфликтах, которые могли бы возникнуть если бы кто-нибудь еще создал класс с именем List.

Пакеты

Все идентификаторы, которые мы до сих пор использовали в наших примерах, располагались в одном и том же пространстве имен (name space). Это означает, что нам во избежание конфликтных ситуаций приходилось заботиться о том, чтобы у каждого класса было свое уникальное имя. Пакеты — это механизм, который служит как для работы с пространством имен, так и для ограничения видимости. У каждого файла .java есть 4 одинаковых внутренних части, из которых мы до сих пор в наших примерах использовали только одну. Ниже приведена общая форма исходного файла Java.

одиночный оператор package (необязателен)

любое количество операторов import (необязательны)

одиночное объявление открытого (public) класса

любое количество закрытых (private) классов пакета (необязательны)

Оператор package

Первое, что может появиться в исходном файле Java — это оператор package, который сообщает транслятору, в каком пакете должны определяться содержащиеся в данном файле классы. Пакеты задают набор раздельных пространств имен, в которых хранятся имена классов. Если оператор package не указан, классы попадают в безымянное пространство имен, используемое по умолчанию. Если вы объявляете класс, как принадлежащий определенному пакету, например,

package java.awt.image;

то и исходный код этого класса должен храниться в каталоге java/awt/image.

ЗАМЕЧАНИЕ

Каталог, который транслятор Java будет рассматривать, как корневой для иерархии пакетов, можно задавать с помощью переменной окружения CLASSPATH. С помощью этой переменной можно задать несколько корневых каталогов для иерархии пакетов (через ; как в обычном PATH).

Трансляция классов в пакетах

При попытке поместить класс в пакет, вы сразу натолкнетесь на жесткое требование точного совпадения иерархии каталогов с иерархией пакетов. Вы не можете переименовать пакет, не переименовав каталог, в котором хранятся его классы. Эта трудность видна сразу, но есть и менее очевидная проблема.

Представьте себе, что вы написали класс с именем PackTest в пакете test. Вы создаете каталог test, помещаете в этот каталог файл PackTest.Java и транслируете. Пока — все в порядке. Однако при попытке запустить его вы получаете от интерпретатора сообщение «can't find class PackTest» («Не могу найти класс PackTest»). Ваш новый класс теперь хранится в пакете с именем test, так что теперь надо указывать всю иерархию пакетов, разделяя их имена точками - test.PackTest. Кроме того Вам надо либо подняться на уровень выше в иерархии каталогов и снова набрать «java test.PackTest», либо внести в переменную CLASSPATH каталог, который является вершиной иерархии разрабатываемых вами классов.

Оператор import

После оператора package, но до любого определения классов в исходном Java-файле, может присутствовать список операторов import. Пакеты являются хорошим механизмом для отделения классов друг от друга, поэтому все встроенные в Java классы хранятся в пакетах. Общая форма оператора **import** такова:

import пакет1 [.пакет2].(имякласса|*);

Здесь *пакет1* — имя пакета верхнего уровня, *пакет2* — это необязательное имя пакета, вложенного в первый пакет и отделенное точкой. И, наконец, после указания пути в иерархии пакетов, указывается либо имя класса, либо метасимвол звездочка. Звездочка означает, что, если Java-транслятору потребуется какой-либо класс, для которого пакет не указан явно, он должен просмотреть все содержимое пакета со звездочкой вместо имени класса. В приведенном ниже фрагменте кода показаны обе формы использования оператора import :

import java.util.Date
import java.io.*;

ЗАМЕЧАНИЕ

Но использовать без нужды форму записи оператора import с использованием звездочки не рекомендуется, т.к. это может значительно увеличить время трансляции кода (на скорость работы и размер программы это не влияет).

Все встроенные в Java классы, которые входят в комплект поставки, хранятся в пакете с именем java. Базовые функции языка хранятся во вложенном пакете java.lang. Весь этот пакет автоматически импортируется транслятором во все программы. Это эквивалентно размещению в начале каждой программы оператора

import java.lang.*;

Если в двух пакетах, подключаемых с помощью формы оператора import со звездочкой, есть классы с одинаковыми именами, однако вы их не используете, транслятор не отреагирует. А вот при попытке использовать такой класс, вы сразу получите сообщение об ошибке, и вам придется переписать операторы import, чтобы явно указать, класс какого пакета вы имеете ввиду.

class MyDate extends Java.util.Date { }

Ограничение доступа

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

- Подклассы в том же пакете.
- Не подклассы в том же пакете.
- Подклассы в различных пакетах.
- Классы, которые не являются подклассами и не входят в тот же пакет.

В языке Java имеется три уровня доступа, определяемых ключевыми словами: private (закрытый), public (открытый) и protected (защищенный), которые употребляются в различных комбинациях. Содержимое ячеек таблицы определяет доступность переменной с данной комбинацией модификаторов (столбец) из указанного места (строка).

	private	модификатор отсутствует	private protected	protected	public
тот же класс	да	да	Да	да	да
подкласс в том же пакете	нет	да	Да	да	да
независимый класс в том же пакете	нет	да	Нет	да	да
подкласс в другом пакете	нет	нет	Да	да	да
независимый класс в другом пакете	нет	нет	Нет	нет	да

На первый взгляд все это может показаться чрезмерно сложным, но есть несколько правил, которые помогут вам разобраться. Элемент, объявленный public, доступен из любого места. Все, что объявлено private, доступно только внутри класса, и нигде больше. Если у элемента вообще не указан модификатор уровня доступа, то такой элемент будет виден из подклассов и классов того же пакета. Именно такой уровень доступа используется в языке Java по умолчанию. Если же вы хотите, чтобы элемент был доступен извне пакета, но только подклассам того класса, которому он принадлежит, вам нужно объявить такой элемент protected. И наконец, если вы хотите, чтобы элемент был доступен только подклассам, причем независимо от того,

находятся ли они в данном пакете или нет — используйте комбинацию private protected.

Ниже приведен довольно длинный пример, в котором представлены все допустимые комбинации модификаторов уровня доступа. В исходном коде первого пакета определяется три класса: Protection, Derived и SamePackage. В первом из этих классов определено пять целых переменных — по одной на каждую из возможных комбинаций уровня доступа. Переменной n приписан уровень доступа по умолчанию, n_pri — уровень private, n_pro — protected, n_pripro — private protected и n_pub — public. Во всех остальных классах мы пытаемся использовать переменные первого класса. Те строки кода, которые из-за ограничения доступа привели бы к ошибкам при трансляции, закомментированы с помощью однострочных комментариев (//) — перед каждой указано, откуда доступ при такой комбинации модификаторов был бы возможен. Второй класс — Derived — является подклассом класса Protection и расположен в том же пакете p1. Поэтому ему доступны все перечисленные переменные за исключением n_pri. Третий класс, SamePackage, расположен в том же пакете, но при этом не является подклассом Protection. По этой причине для него недоступна не только переменная n_pri, но и n_pripro, уровень доступа которой — private protected.

```
package p1;
public class Protection {
int n = 1;
private int n_pri = 2;
protected int n_pro = 3;
private protected int n_pripro = 4;
public int n_pub = 5;
public Protection() {
System.out.println("base constructor");
System.out.println("n = " + n);
System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pripro = " + n_pripro);
System.out.println("n_pub = " + n_pub);
} }
class Derived extends Protection {
Derived() {
System.out.println("derived constructor");
System.out.println("n = " + n);
// только в классе
// System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pripro = " + n_pripro);
System.out.println("n_pub = " + n_pub);
} }
```

```
class SamePackage {
SamePackage() {
Protection p = new Protection();
System.out.println("same package constructor");
System.out.println("n = " + p.n);
// только в классе
// System.out.println("n_pri = " + p.n_pri);
System.out.println("n_pro = " + p.n_pro);
// только в классе и подклассе
// System.out.println("n_pripro = " + p.n_pripro):
System.out.println("n_pub = " + p.n_pub):
} }
```

Контрольные вопросы

1. Для чего используются пакеты?
2. Приведите определение пакета.
3. Для чего используется оператор import?
4. Опишите четыре категории видимости между элементами классов.
5. Сколько имеется уровней доступа?

Литература:

1. Смирнов Н.И. Java -2. Учебное пособие. М.:«Три Л», 2000. -320 с.
2. Арнольд К. и др. Язык программирования JAVA/ Вильямс. 2001. 624 с.
3. Брюс Эккель. Философия Java . Библиотека программиста. 2003.

ЛЕКЦИЯ 13. ТЕМА: ВСТРОЕННЫЕ КЛАССЫ

План: Встроенные классы; Встроенные классы и интерфейсы верхнего уровня; Классы-члены.

Встроенные классы

Самым большим усовершенствованием языка Java версии 1.1 являются так называемые *встроенные* классы. С их появлением стало возможно определять классы в качестве членов других классов, подобно тому, как в классах определяются переменные и методы.

С одной стороны, добавление встроенных классов упорядочивает синтаксис языка Java, а с другой, использование таких классов сопряжено с возникновением нескольких особых ситуаций и целого ряда новых правил. Тем не менее, с практической точки зрения, встроенные классы оказываются элегантным и крайне полезным дополнением языка, если избегать неординарных ситуаций. Особенно часто их применяют вместе с новой моделью обработки событий, которая определена для AWT Java 1.1.

Обзор встроенных классов

Java 1.0 позволяет определять классы и интерфейсы только на верхнем уровне как члены пакетов. В Java 1.1 добавлен новый тип классов и интерфейсов верхнего уровня, а также три новых типа встроенных классов, находящихся на более низком уровне. В последующих параграфах подробно описывается каждый тип классов и интерфейсов и приводятся примеры их использования.

Встроенные классы и интерфейсы верхнего уровня

Встроенные классы и *интерфейсы верхнего уровня* определяются как статические члены класса или интерфейса, которому они принадлежат. При определении встроенного класса верхнего уровня используется модификатор static, как при определении статических методов или статических переменных. Встроенные интерфейсы всегда являются статическими неявно (хотя их можно объявлять с модификатором static, чтобы определять их статическими в явном виде) и всегда находятся на верхнем уровне. Встроенные классы и интерфейсы верхнего уровня ведут себя, как обычные классы и интерфейсы, т.е. являются членами пакетов. Различие заключается лишь в том, что имя встроенного класса или интерфейса верхнего уровня всегда включает в себя имя класса, к которому относится данный встроенный класс или интерфейс. Так, для класса LinkedList можно определить встроенный интерфейс верхнего уровня Linkable. Тогда на этот интерфейс необходимо ссылаться следующим образом: LinkedList.Linkable. Встроенные классы и интерфейсы верхнего уровня удобно использовать для группировки взаимосвязанных классов.

В следующем примере LinkedList.java продемонстрировано, как определить встроенный вспомогательный интерфейс верхнего уровня. Обратите внимание на использование ключевого слова static в объявлении этого интерфейса. Из примера также видно, как применяется данный интерфейс в классе, внутри которого он объявлен, и в других внешних классах.

```
public class LinkedList {
// Этот встроенный интерфейс верхнего уровня
// определяется так же, как и статический член.
public interface Linkable {
public Linkable getNext();
public void setNext (Linkable node);
}
// Заголовком списка является объект Linkable.
Linkable head;
// Тела методов опущены.
public void insert (Linkable node) {...}
public remove (Linkable node) { ... }
}
// В этом классе определяется тип узла, с которым предполагается
// работать в связанном списке. Обратите внимание, как
// записывается имя встроенного интерфейса.
class LinkableInteger implements LinkedList.Linkable
{
// Ниже приведены переменные и конструктор для узла.
int i;
public LinkableInteger (int i) { this.i = i; }
// Ниже приведены переменные и методы, необходимые для
// реализации интерфейса.
LinkedList.Linkable next;
public LinkedList.Linkable getNext() { return next; }
public void setNext (LinkedList.Linkable node) {
next=node;
} }
```

Если откомпилировать файл *LinkedList.java*, в результате будут созданы два файла классов. Именем первого, как и следовало ожидать, будет имя *LinkedList.class*. Однако создается и другой файл с названием *LinkedList$Linkable.class*. Символ $ автоматически вставляется в название компилятором Java.

Виртуальной машине Java ничего не известно о встроенных классах и интерфейсах верхнего уровня или о внутренних классах различных типов. Поэтому компилятор Java должен привести новые типы к стандартным, не вложенным файлам классов, чтобы интерпретатор Java мог их понимать. Это

достигается путем преобразований исходного текста программы, при котором в имена встроенных классов вставляется символ $.

Классы-члены

Классы-члены также определяются как члены некоторого включающего их класса, но в отличие от встроенных классов верхнего уровня при их определении не используется модификатор static. Это означает, что они представляют собой внутренние классы, а не классы верхнего уровня. Интерфейс-член по аналогии объявить нельзя, такое понятие отсутсвует (да это и противоречило бы всей концепции интерфейсов). Во многих отношениях класс-член аналогичен другим членам класса — переменным и методам. Класс-член интересен тем, что его код может неявно ссылаться на любую переменную или метод, в том числе на переменные и методы, объявленные как private, класса, членом которого является данный класс. Каждый экземпляр класса-члена связан с экземпляром включающего его класса. В связи с этим к экземпляру включающего класса в язык Java добавлено несколько новых синтаксических особенностей.

Как и встроенные классы высокого уровня, классы-члены обычно используются в качестве вспомогательных классов, необходимых включающему их классу. Классы-члены применяются вместо встроенных классов высокого уровня в тех случаях, когда необходим доступ к переменным экземпляра включающего класса или когда каждый экземпляр вспомогательного класса должен быть связан с определенным экземпляром включающего класса. При использовании классов-членов такое соответствие соблюдается автоматически.

Вернемся к рассмотренному выше примеру класса LinkedList. Предположим, что в нем необходимо организовать циклическое обращение к элементам связанного списка с помощью интерфейса java.util.Enumeration. Чтобы добиться этого, определим отдельный класс, реализующий данный интерфейс, и затем добавим в класс LinkedList метод, который будет возвращать экземпляр отдельного класса Enumeration. В следующем примере приведена типичная реализация такого подхода в стиле Java 1.0.

```
import java.util.*;
public class LinkedList {
// Встроенный интерфейс высокого уровня. Тело его опущено...
public interface Linkable {... }

// Заголовок списка.
Linkable head;

// Тела методов опущены.
public void addToHead (Linkable node) {...}
public Linkable removeHead() {...}
```

```
// Данный метод возвращает объект Enumeration для
// данного объекта LinkedList.
public Enumeration enumerate() {
return new LinkedListEnumerator (this);
}
}

// В этом классе определяется тип Enumeration, который
// используется для получения списка элементов в LinkedList.
// Отметим, что каждый объект LinkedListEnuroerator связан с
// определенным объектом LinkedList, который передается конструктору.
class LinkedListEnumerator implements Enumeration {
private LinkedList container;
private LinkedList.Linkable current;
public LinkedListEnumerator (LinkedList 1) {
container = 1;
current = container.head;
}
public boolean hasMoreElements() { return (current != null); }
public Object nextElement() {
if (current == nul) throw
new NoSuchElementException("LinkedList");
Object value = current;
Current = current.getNext();
Return value;
}
}
```

Следует отметить, что конструктору класса LinkedListEnumerator необходимо передавать объект LinkedList в явном виде. Класс LinkedListEnumerator определяется как отдельный класс высокого уровня, тогда как на самом деле элегантнее было бы определить его как составляющую самого класса LinkedList. Это легко реализуется в Java 1.1 через класс-член, как продемонстрировано в следующем примере.

```
import java.util.*;
public class LinkedList
{
// Встроенный интерфейс высокого уровня. Тело опущено…
public interface Linkable {...}
//Заголовок списка.
//Эту переменную можно было бы объявить как private, если бы не
//ошибки компилятора, связанные со встроенными классами.
/* private */ Linkable head;
```

```
// Тела методов опущены.
public void addToHead(Linkable node) {...}
public Linkable removeHead() {...}
//Данный метод возвращает объект Enumeration для данного объекта
//LinkedList. Отметим: никакой из объектов LinkedList не
//передается конструктору в явном виде.
public Enumeration enumerate() {return new Enumerator();}
//Здесь приведена реализация интерфейса Enumeration,
//который определяется как класс-член с модификатором
//видимости private.
private class Enumerator implements Enumeration {
Linkable current;
// Заметьте: конструктор неявно ссылается на переменную head
// включающего класса.
public Enumerator (){ current = head; }
public boolean hasMoreElements() { return(current != null);}
public Object nextElement() {
if (current == null) throw
new NoSuchElementException("LinkedList") ;
Object value = current;
current = current.getNext();
return value;
}
}
}
```

Обратите внимание, как в данном примере класс Enumerator включается в класс LinkedList. Действительно очень удобно определять вспомогательный класс в том месте программы, где он используется включающим классом. Конечно же, запустив приведенный выше пример на компиляцию, можно убедиться в том, что класс-член Enumerator будет откомпилирован в файл LinkedList$Enumerator.class. Хотя в исходном тексте программы один класс является вложенным в другой, этого нельзя сказать об откомпилированном байт-коде программы.

Обратите внимание: конструктору класса-члена Enumerator не передается ни один из экземпляров включающего класса LinkedList. Класс-член может ссылаться на члены включающего его класса неявным образом. Необходимости в каких-либо явных ссылках нет. Также обратите внимание на то, что в классе Enumerator используется переменная *head* включающего класса, хотя эта переменная и объявлена как private. В общем случае классы-члены, равно как локальные и анонимные классы, могут использовать переменные и методы (и даже классы!), объявленные как private, своего включающего класса. Аналогичным образом, включающий класс может использовать закрытые переменные, методы и классы тех классов, которые

встроены в него. Также любые два класса, вложенные в третий класс, имеют доступ к private-членам друг друга.

Контрольные вопросы

1. Что такое встроенные классы?
2. Приведите типы встроенных классов.
3. Для чего используется интерфейсы верхнего уровня?
4. Опишите особенности классов - членов.
5. Почему нельзя создать интерфейсы подобно классам - членам?

Литература:

1. Смирнов Н.И. Java -2. Учебное пособие. М.:«Три Л», 2000. -320 с.
2. Арнольд К. и др. Язык программирования JAVA/ Вильямс. 2001. 624 с.
3. Брюс Эккель. Философия Java . Библиотека программиста. 2003.

ЛЕКЦИЯ 14. ТЕМА: РАБОТА С ФАЙЛАМИ

План: Ввод/Вывод; InputStream; OutputStream; FilelnputStream; FileOutputStream; ByteArraylnputStream;ByteArrayOutputStream; StringBufferlnputStream.

Ввод/Вывод

Обобщенное понятие источника ввода относится к различным способам получения информации: к чтению дискового файла, символов с клавиатуры, либо получению данных из сети. Аналогично, под обобщенным понятием вывода также могут пониматься дисковые файлы, сетевое соединение и т.п. Эти абстракции дают удобную возможность для работы с вводом-выводом (I/O), не требуя при этом, чтобы каждая часть вашего кода понимала разницу между, скажем, клавиатурой и сетью. В Java эта абстракция называется потоком (stream) и реализована в нескольких классах пакета java.io. Ввод инкапсулирован в классе InputStream, вывод — в OutputStream. В Java есть несколько специализаций этих абстрактных классов, учитывающих различия при работе с дисковыми файлами, сетевыми соединениями и даже с буферами в памяти.

InputStream

InputStream — абстрактный класс, задающий используемую в Java модель входных потоков. Все методы этого класса при возникновении ошибки возбуждают исключение IOException. Ниже приведен краткий обзор методов класса InputStream.

- read() возвращает представление очередного доступного символа во входном потоке в виде целого.
- read(byte b[]) пытается прочесть максимум b.length байтов из входного потока в массив b. Возвращает количество байтов, в действительности прочитанных из потока.
- read(byte b[], int off, int len) пытается прочесть максимум len байтов, расположив их в массиве b, начиная с элемента off. Возвращает количество реально прочитанных байтов.
- skip(long n) пытается пропустить во входном потоке n байтов. Возвращает количество пропущенных байтов.
- available() возвращает количество байтов, доступных для чтения в настоящий момент.
- close() закрывает источник ввода. Последующие попытки чтения из этого потока приводят к возбуждению IOException.
- mark(int readlimit) ставит метку в текущей позиции входного потока, которую можно будет использовать до тех пор, пока из потока не будет прочитано readlimit байтов.
- reset() возвращает указатель потока на установленную ранее метку.

- markSupported() возвращает true, если данный поток поддерживает операции mark/reset.

OutputStream

Как и InputStream, OutputStream — абстрактный класс. Он задает модель выходных потоков Java. Все методы этого класса имеют тип void и возбуждают исключение IOException в случае ошибки. Ниже приведен список методов этого класса:

- write(int b) записывает один байт в выходной поток. Обратите внимание — аргумент этого метода имеет тип int, что позволяет вызывать write, передавая ему выражение, при этом не нужно выполнять приведение его типа к byte.
- write(byte b[]) записывает в выходной поток весь указанный массив байтов.
- write(byte b[], int off, int len) записывает в поток часть массива — len байтов, начиная с элемента b[off].
- flush() очищает любые выходные буферы, завершая операцию вывода.
- close() закрывает выходной поток. Последующие попытки записи в этот поток будут возбуждать IOException.

Файловые потоки

FilelnputStream

Класс FileInputStream используется для ввода данных из файлов. В приведенном ниже примере создается два объекта этого класса, использующие один и тот же дисковый файл.

```
InputStream f0 = new FileInputStream("/autoexec.bat");
File f = new File("/autoexec.bat"):
InputStream f1 = new FileInputStream(f);
```

Когда создается объект класса FileInputStream, он одновременно с этим открывается для чтения. FileInputStream замещает шесть методов абстрактного класса InputStream. Попытки применить к объекту этого класса методы mark и reset приводят к возбуждению исключения IOException. В приведенном ниже примере показано, как можно читать одиночные байты, массив байтов и поддиапазон массива байтов. В этом примере также показано, как методом available можно узнать, сколько еще осталось непрочитанных байтов, и как с помощью метода skip можно пропустить те байты, которые вы не хотите читать.

```
import java.io.*;
import java.util.*;
class FileInputStreamS {
public static void main(String args[]) throws Exception {
int size;
InputStream f1 = new FileInputStream("/wwwroot/default.htm");
```

```
size = f1.available();
System.out.println("Total Available Bytes: " + size);
System.out.println("First 1/4 of the file: read()");
for (int i=0; i < size/4; i++) {
System.out.print((char) f1.read());
}
System.out.println("Total Still Available: " + f1.available());
System.out.println("Reading the next 1/8: read(b[])");
byte b[] = new byte[size/8];
if (f1.read(b) != b.length) {
System.err.println("Something bad happened");
}
String tmpstr = new String(b, 0, 0, b.length);
System.out.println(tmpstr);
System.out.println("Still Available: " + f1.available());
System.out.println("Skipping another 1/4: skip()");
f1.skip(size/4);
System.out.println( "Still Available: " + f1.available());
System.out.println("Reading 1/16 into the end of array");
if (f1.read(b, b.length-size/16, size/16) != size/16) {
System.err.println("Something bad happened");
}
System.out.println("Still Available: " + f1.available());
f1.close();
}
}
```

FileOutputStream

У класса FileOutputStream — два таких же конструктора, что и у FileInputStream. Однако, создавать объекты этого класса можно независимо от того, существует файл или нет. При создании нового объекта класс FileOutputStream перед тем, как открыть файл для вывода, сначала создает его.

В очередном нашем примере символы, введенные с клавиатуры, считываются из потока System.in - по одному символу за вызов, до тех пор, пока не заполнится 12-байтовый буфер. После этого создаются три файла. В первый из них, file1.txt, записываются символы из буфера, но не все, а через один — нулевой, второй и так далее. Во второй, file2.txt, записывается весь ввод, попавший в буфер. И наконец в третий файл записывается половина буфера, расположенная в середине, а первая и последняя четверти буфера не выводятся.

```
import java.io.*;
class FileOutputStreamS {
```

```
public static byte getlnput()[] throws Exception {
byte buffer[] = new byte[12];
for (int i=0; i<12; i++) {
buffer[i] = (byte) System.in.read();
}
return buffer;
}
public static void main(String args[]) throws Exception {
byte buf[] = getlnput();
OutputStream f0 = new FileOutputStream("file1.txt");
OutputStream f1 = new FileOutputStream("file2.txt");
OutputStream f2 = new FileOutputStream("file3.txt");
for (int i=0; i < 12; i += 2) {
f0.write(buf[i]);
}
f0.close();
f1.write(buf);
f1.close();
f2.write(buf, 12/4, 12/2);
f2.close();
} }
```

В настоящее время не существует способа открыть FileOutputStream для дозаписи в конец файла. Если вы открываете файл с помощью конструктора FileOutputStream, прежнее содержимое этого файла теряется. Это - явный недостаток реализации Java.

ByteArraylnputStream

ByteArrayInputStream - это реализация входного потока, в котором в качестве источника используется массив типа byte. У этого класса два конструктора, каждый из которых в качестве первого параметра требует байтовый массив. В приведенном ниже примере создаются два объекта этого типа. Эти объекты инициализируются символами латинского алфавита.

```
String tmp = "abcdefghijklmnopqrstuvwxyz";
byte b[] = new byte [tmp.length()];
tmp. getBytes(0, tmp.length(), b, 0);
ByteArrayInputStream input1 = new ByteArrayInputStream(b);
ByteArrayInputStream input2 = new ByteArreyInputStream(b,0,3);
```

ByteArrayOutputStream

У класса ByteArrayOutputStream — два конструктора. Первая форма конструктора создает буфер размером 32 байта. При использовании второй формы создается буфер с размером, заданным параметром конструктора (в приведенном ниже примере — 1024 байта):

```
OutputStream out0 = new ByteArrayOutputStream();
```

```
OutputStream out1 = new ByteArrayOutputStream(1024);
```

В очередном примере объект ByteArrayOutputStream заполняется символами, введенными с клавиатуры, после чего с ним выполняются различные манипуляции.

```
import java.io.*;
import java.util.*;
class ByteArrayOutputStreamS {
public static void main(String args[]) throws Exception {
int i;
ByteArrayOutputStream f0 = new ByteArrayOutputStream(12);
System.out.println("Enter 10 characters and a return");
while (f0.size() != 10) {
f0.write( System.in.read());
}
System.out.println("Buffer as a string");
System.out.println(f0.toString());
System.out.println ("Into array");
byte b[] = f0.toByteArray();
for (i=0; i < b.length; i++) {
System.out.print((char) b[i]);
}
System.out.println();
System.out. println("To an OutputStream()");
OutputStream f2 = new File0utput8tream("test.txt");
f0.writeTo(f2);
System.out.println("Doing a reset");
f0. reset();
System.out.println("Enter 10 characters and a return");
while (f0.size() != 10) {
f0.write (System.in.read());
}
System.out.println("Done.");
} }
```

Заглянув в созданный в этом примере файл test.txt, мы увидим там именно то, что ожидали:

```
C:\> type test.txt
0123456789
```

StringBufferlnputStream

StringBufferInputStream идентичен классу ByteArrayInputStream с тем исключением, что внутренним буфером объекта этого класса является экземпляр String, а не байтовый массив. Кроме того, в Java нет соответствующего ему класса StringBufferedOutputStream. У этого класса есть единственный конструктор:

StringBufferInputStream(String s)

Контрольные вопросы:

1. Что такое поток?
2. Укажите абстрактные классы для работы с вводом.
3. Укажите абстрактные классы для работы с выводом.
4. Укажите классы для чтения из файла.
5. Укажите классы для записи в файл.

Литература:

1. Смирнов Н.И. Java -2. Учебное пособие. М.:«Три Л», 2000. -320 с.
2. Арнольд К. и др. Язык программирования JAVA/ Вильямс. 2001. 624 с.
3. Брюс Эккель. Философия Java . Библиотека программиста. 2003.

ЛЕКЦИЯ 15. ТЕМА: ОСОБЕННОСТИ РАБОТЫ С ФАЙЛАМИ

План: File; Каталоги; Интерфейс FilenameFilter; Фильтруемые потоки; Буферизованные потоки; Класс PrintStream.

File

File — единственный объект в **java.io**, который работает непосредственно с дисковыми файлами. Хотя на использование файлов в апплетах наложены жесткие ограничения, файлы по прежнему остаются основными ресурсами для постоянного хранения и совместного использования информации. Каталог в Java трактуется как обычный файл, но с дополнительным свойством — списком имен файлов, который можно просмотреть с помощью метода list.

ЗАМЕЧАНИЕ

Java правильно обрабатывает разделители имен каталогов в пути, используемые в UNIX и DOS. Если вы используете стиль UNIX — символы '/', то при работе в Windows Java автоматически преобразует их в '\'. Не забудьте, если вы привыкли к разделителям, принятым в DOS, то есть, к '\', то для того, чтобы включить их в строку пути, необходимо их удвоить, аналогично тому, как это сделано в строке “\\java\\COPYRIGHT”.

Для определения стандартных свойств объекта в классе File есть много разных методов. Однако, класс File несимметричен. Есть много методов, позволяющих узнать свойства объекта, но соответствующие функции для изменения этих свойств отсутствуют. В очередном примере используются различные методы, позволяющие получить характеристики файла:

```
import java.io.File;
class FileTest {
static void p(String s) {
System.out.println(s);
}
public static void main(String args[]) {
File f1 = new File("/java/COPYRIGHT");
p("File Name:" + f1 .getName());
p("Path:" + f1.getPath());
p("Abs Path:" + f1.getAbsolutePath());
p("Parent:" + f1.getParent());
p(f1.exists() ? "exists" : "does not exist");
p(f1.canWrite() ? "is writeable" : "is not writeable");
p(f1.canRead() ? "is readable" : "is not readable");
p("is " + (f1.isDirectory() ? " " : "not") + " a directory");
p(f1.isFile() ? "is normal file" : "might be a named pipe");
p(f1.isAbsolute() ? "is absolute" : "is not absolute");
p("File last modified:" + f1. lastModified());
p("File size:" + f1.length() + " Bytes");
```

} }

При запуске этой программы вы получите что-то наподобие вроде:

File Name:COPYRIGHT (имя файла)
Path:/java/COPYRIGHT (путь)
Abs Path:/Java/COPYRIGHT (путь от корневого каталога)
Parent:/java (родительский каталог)
exists (файл существует)
is writeable (разрешена запись)
is readable (разрешено чтение)
is not a directory (не каталог)
is normal file (обычный файл)
is absolute
File last modified:812465204000 (последняя модификация файла)
File size:695 Bytes (размер файла)

Существует также несколько сервисных методов, использование которых ограничено обычными файлами (их нельзя применять к каталогам). Метод renameTo(File dest) переименовывает файл (нельзя переместить файл в другой каталог). Метод delete уничтожает дисковый файл. Этот метод может удалять только обычные файлы, каталог, даже пустой, с его помощью удалить не удастся.

Каталоги

Каталоги — это объекты класса File, в которых содержится список других файлов и каталогов. Если File ссылается на каталог, его метод isDirectory возвращает значение true. В этом случае вы можете вызвать метод list и извлечь содержащиеся в объекте имена файлов и каталогов. В очередном примере показано, как с помощью метода list можно просмотреть содержимое каталога.

```
import java.io.File;
class DirList {
public static void main(String args[]) {
String dirname = "/java"; // имя каталога
File f1 = new File(dirname);
if (f1.isDirectory()) { // является ли f1 каталогом
System.out.println("Directory of ' + dirname);
String s[] = f1.list();
for ( int i=0; i < s.length; i++) {
File f = new File(dirname + "/" + s[i]);
if (f.isDirectory()) { // является ли f каталогом System.out.println(s[i] +
" is a directory"):
} else {
System.out.println(s[i] + " is a file");
} } } else {
System.out.println(dirname + " is not a directory");
```

```
} }
}
```

В процессе работы эта программа вывела содержимое каталога /java моего персонального компьютера в следующем виде:

```
C:\> java DirList
Directory of /java
bin is a directory
COPYRIGHT is a file
README is a file
```

FilenameFilter

Зачастую у вас будет возникать потребность ограничить количество имен файлов, возвращаемых методом list, чтобы получить от него только имена, соответствующие определенному шаблону. Для этого в пакет java.io включен интерфейс FilenameFilter. Объекту, чтобы реализовать этот интерфейс, требуется определить только один метод — accept(), который будет вызываться один раз с каждым новым именем файла. Метод accept должен возвращать true для тех имен, которые надо включать в список, и false для имен, которые следует исключить.

У класса File есть еще два сервисных метода, ориентированных на работу с каталогами. Метод mkdir создает подкаталог. Для создания каталога, путь к которому еще не создан, надо использовать метод mkdirs — он создаст не только указанный каталог, но и все отсутствующие родительские каталоги.

Фильтруемые потоки

При работе системы вывода в среде с параллельными процессами при отсутствии синхронизации могут возникать неожиданные результаты. Причиной этого являются попытки различных подпроцессов одновременно обратиться к одному и тому же потоку. Все конструкторы и методы, имеющиеся в этом классе, идентичны тем, которые есть в классах InputStream и OutputStream, единственное отличие классов фильтруемых потоков в том, что их методы синхронизованы.

Буферизованные потоки

Буферизованные потоки являются расширением классов фильтруемых потоков, в них к потокам ввода-вывода присоединяется буфер в памяти. Этот буфер выполняет две основные функции:

- Он дает возможность исполняющей среде java проделывать за один раз операции ввода-вывода с более чем одним байтом, тем самым повышая производительность среды.
- Поскольку у потока есть буфер, становятся возможными такие операции, как пропуск данных в потоке, установка меток и очистка буфера.

BufferedInputStream

Буферизация ввода-вывода — общепринятый способ оптимизации таких операций. Класс BufferedlnputStream в Java дает возможность “окружить” любой объект InputStream буферизованным потоком, и, тем самым, получить выигрыш в производительности. У этого класса два конструктора. Первый из них

BufferedInputStream(InputStream in)

создает буферизованный поток, используя для него буфер длиной 32 байта. Во втором

BufferedInputStream(InputStream in, int size)

размер буфера для создаваемого потока задается вторым параметром конструктора. В общем случае оптимальный размер буфера зависит от операционной системы, количества доступной оперативной памяти и конфигурации компьютера.

BufferedOutputStream

Вывод в объект BufferedOutputStream идентичен выводу в любой OutputStream с той разницей, что новый подкласс содержит дополнительный метод flush, применяемый для принудительной очистки буфера и физического вывода на внешнее устройство хранящейся в нем информации. Первая форма конструктора этого класса:

BufferedOutputStream(OutputStream out)

создает поток с буфером размером 32 байта. Вторая форма:

BufferedOutputStream(OutputStream out, int size)

позволяет задавать требуемый размер буфера.

PushbacklnputStream

Одно из необычных применений буферизации — реализация операции pushback (вернуть назад). Pushback применяется к InputStream для того, чтобы после прочтения символа вернуть его обратно во входной поток. Однако возможности класса PushbacklnputStream весьма ограничены - любая попытка вернуть в поток более одного символа приведет к немедленному возбуждению исключения IOException. У этого класса — единственный конструктор

PushbackInputStream(InputStream in)

Помимо уже хорошо нам знакомых методов класса InputStream, PushbacklnputStream содержит метод unread(int ch), который возвращает заданный аргументом символ ch во входной поток.

SequencelnputStream

Класс SequencelnputStream поддерживает новую возможность слияния нескольких входных потоков в один. В конструкторе класса SequenceInputStream в качестве параметра используется либо два объекта

InputStream, либо перечисление, содержащее коллекцию объектов InputStream:

SequenceInputStream(Enumeration e)
SequenceInputStream(InputStream s0, InputStream s1)

В процессе работы класс выполняет поступающие запросы, считывая информацию из первого входного потока до тех пор, пока он не закончится, после чего переходит ко второму и т.д.

PrintStream

Класс PrintStream предоставляет все те утилиты форматирования, которые мы использовали в примерах для вывода через файловые дескрипторы пакета System с самого начала книги. Вы уже привыкли писать “System.out.println”, не сильно задумываясь при этом о тех классах, которые занимаются форматированием выводимой информации. У класса PrintStream два конструктора: **PrintStream(OutputStream out)** и **PrintStream(OutputStream out, boolean autoflush)**. Параметр autoflush второго из них указывает, должна ли исполняющая среда Java автоматически выполнять операцию очистки буфера над выходным потоком.

В Java-объектах PrintStream есть методы print и println, “умеющие” работать с любыми типами данных, включая Object. Если в качестве аргумента этих методов используется не один из примитивных типов, то они вызывают метод toString класса Object, после чего выводят полученный результат.

Контрольные вопросы:

1. Какой объект используется для работы с дисковыми файлами?
2. Что такое каталог?
3. Укажите способ осмотра содержимого каталога.
4. Как осуществляется буферизация потока?
5. Что такое буферизированные потоки?

Литература:

1. Смирнов Н.И. Java -2. Учебное пособие. М.:«Три Л», 2000. -320 с.
2. Арнольд К. и др. Язык программирования JAVA/ Вильямс. 2001. 624 с.
3. Брюс Эккель. Философия Java . Библиотека программиста. 2003.

ЛЕКЦИЯ 16. ТЕМА: УПРАВЛЕНИЕ ИСКЛЮЧИТЕЛЬНЫМИ СИТУАЦИЯМИ

План: Исключительные ситуации; Основы; Типы исключений; Неперехваченные исключения; try и catch; Несколько разделов catch; Вложенные операторы try.

Исключительные ситуации

В этой главе обсуждается используемый в Java механизм *обработки исключений*. Исключение в Java — это объект, который описывает исключительное состояние, возникшее в каком-либо участке программного кода. Когда возникает исключительное состояние, создается объект класса Exception. Этот объект пересылается в метод, обрабатывающий данный тип исключительной ситуации. Исключения могут возбуждаться и «вручную» для того, чтобы сообщить о некоторых нештатных ситуациях.

Основы

К механизму обработки исключений в Java имеют отношение 5 ключевых слов: — **try**, **catch**, **throw**, **throws** и **finally**. Схема работы этого механизма следующая. Вы пытаетесь (try) выполнить блок кода, и если при этом возникает ошибка, система возбуждает (throw) исключение, которое в зависимости от его типа вы можете перехватить (catch) или передать умалчиваемому (finally) обработчику.

Ниже приведена общая форма блока обработки исключений.

```
try {
// блок кода }
catch (ТипИсключения1 e) {
// обработчик исключений типа ТипИсключения1 }
catch (ТипИсключения2 e) {
// обработчик исключений типа ТипИсключения2
throw(e) // повторное возбуждение исключения }
finally {
}
```

Типы исключений

В вершине иерархии исключений стоит класс Throwable. Каждый из типов исключений является подклассом класса Throwable. Два непосредственных наследника класса Throwable делят иерархию подклассов исключений на две различные ветви. Один из них — класс Exception — используется для описания исключительных ситуации, которые должны перехватываться программным кодом пользователя. Другая ветвь дерева подклассов Throwable — класс Error, который предназначен для описания

исключительных ситуаций, которые при обычных условиях не должны перехватываться в пользовательской программе.

Неперехваченные исключения

Объекты-исключения автоматически создаются исполняющей средой Java в результате возникновения определенных исключительных состояний. Например, очередная наша программа содержит выражение, при вычислении которого возникает деление на нуль.

```
class Exc0 {
public static void main(string args[]) {
int d = 0;
int a = 42 / d;
} }
```

Вот вывод, полученный при запуске нашего примера.

```
C:\> java Exc0
java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:4)
```

Обратите внимание на тот факт что типом возбужденного исключения был не Exception и не Throwable. Это подкласс класса Exception, а именно: ArithmeticException, поясняющий, какая ошибка возникла при выполнении программы. Вот другая версия того же класса, в которой возникает та же исключительная ситуация, но на этот раз не в программном коде метода main.

```
class Exc1 {
static void subroutine() {
int d = 0;
int a = 10 / d;
}
public static void main(String args[]) {
Exc1.subroutine();
} }
```

Вывод этой программы показывает, как обработчик исключений исполняющей системы Java выводит содержимое всего стека вызовов.

```
C:\> java Exc1
java.lang.ArithmeticException: / by zero
at Exc1.subroutine(Exc1.java:4)
at Exc1.main(Exc1.java:7)
```

try и catch

Для задания блока программного кода, который требуется защитить от исключений, используется ключевое слово try. Сразу же после try-блока помещается блок catch, задающий тип исключения которое вы хотите обрабатывать.

```
class Exc2 {
```

```
public static void main(String args[]) {
try {
int d = 0;
int a = 42 / d;
}
catch (ArithmeticException e) {
System.out.println("division by zero");
}
} }
```

Целью большинства хорошо сконструированных catch-разделов должна быть обработка возникшей исключительной ситуации и приведение переменных программы в некоторое разумное состояние — такое, чтобы программу можно было продолжить так, будто никакой ошибки и не было (в нашем примере выводится предупреждение – division by zero).

Несколько разделов catch

В некоторых случаях один и тот же блок программного кода может возбуждать исключения различных типов. Для того, чтобы обрабатывать подобные ситуации, Java позволяет использовать любое количество catch-разделов для try-блока. Наиболее специализированные классы исключений должны идти первыми, поскольку ни один подкласс не будет достигнут, если поставить его после суперкласса. Следующая программа перехватывает два различных типа исключений, причем за этими двумя специализированными обработчиками следует раздел catch общего назначения, перехватывающий все подклассы класса Throwable.

```
class MultiCatch {
public static void main(String args[]) {
try {
int a = args.length;
System.out.println("a = " + a);
int b = 42 / a;
int c[] = { 1 };
c[42] = 99;
}
catch (ArithmeticException e) {
System.out.println("div by 0: " + e);
}
catch(ArrayIndexOutOfBoundsException e) {
System.out.println("array index oob: " + e);
}
} }
```

Этот пример, запущенный без параметров, вызывает возбуждение исключительной ситуации деления на нуль. Если же мы зададим в командной строке один или несколько параметров, тем самым установив a в

значение больше нуля, наш пример переживет оператор деления, но в следующем операторе будет возбуждено исключение выхода индекса за границы массива ArrayIndexOutOf Bounds. Ниже приведены результаты работы этой программы, запущенной и тем и другим способом.

C:\> java MultiCatch
a = 0
div by 0: java.lang.ArithmeticException: / by zero
C:\> java MultiCatch 1
a = 1
array index oob: java.lang.ArrayIndexOutOfBoundsException: 42

Вложенные операторы try

Операторы try можно вкладывать друг в друга аналогично тому, как можно создавать вложенные области видимости переменных. Если у оператора try низкого уровня нет раздела catch, соответствующего возбужденному исключению, стек будет развернут на одну ступень выше, и в поисках подходящего обработчика будут проверены разделы catch внешнего оператора try. Вот пример, в котором два оператора try вложены друг в друга посредством вызова метода.

```
class MultiNest {
static void procedure() {
try {
int c[] = { 1 };
c[42] = 99;
}
catch(ArrayIndexOutOfBoundsException e) {
System.out.println("array index oob: " + e);
} }
public static void main(String args[]) {
try {
int a = args.length();
System.out.println("a = " + a);
int b = 42 / a;
procedure();
}
catch (ArithmeticException e) {
System.out.println("div by 0: " + e);
}
} }
```

Контрольные вопросы:

1. Что такое исключительная ситуация?
2. Укажите иерархию классов для обработки исключительных ситуаций.
3. Приведите синтаксис записи обработки исключений.
4. Какие операторы, которые используются для реализации исключений?
5. Укажите синтаксис записи функции, генерирующей исключения.

Литература:

1. Смирнов Н.И. Java -2. Учебное пособие. М.:«Три Л», 2000. -320 с.
2. Арнольд К. и др. Язык программирования JAVA/ Вильямс. 2001. 624 с.
3. Брюс Эккель. Философия Java . Библиотека программиста. 2003.

ЛЕКЦИЯ 17.ТЕМА: ОСОБЕННОСТИ УПРАВЛЕНИЯ ИСКЛЮЧИТЕЛЬНЫМИ СИТУАЦИЯМИ

План: throw; throws; finally; Подклассы Exception.

throw

Оператор throw используется для возбуждения исключения «вручную». Для того, чтобы сделать это, нужно иметь объект подкласса класса Throwable, который можно либо получить как параметр оператора catch, либо создать с помощью оператора new. Ниже приведена общая форма оператора throw.

throw ОбъектТипаThrowable;

При достижении этого оператора нормальное выполнение кода немедленно прекращается, так что следующий за ним оператор не выполняется. Ближайший окружающий блок try проверяется на наличие соответствующего возбужденному исключению обработчика catch. Если такой отыщется, управление передается ему. Если нет, проверяется следующий из вложенных операторов try, и так до тех пор пока либо не будет найден подходящий раздел catch, либо обработчик исключений исполняющей системы Java не остановит программу, выведя при этом состояние стека вызовов. Ниже приведен пример, в котором сначала создается объект-исключение, затем оператор throw возбуждает исключительную ситуацию, после чего то же исключение возбуждается повторно — на этот раз уже кодом перехватившего его в первый раз раздела catch.

```
class ThrowDemo {
static void demoproc() {
try {
throw new NullPointerException("demo");
}
catch (NullPointerException e) {
System.out.println("caught inside demoproc");
throw e;
} }
public static void main(String args[]) {
try {
demoproc();
}
catch(NulPointerException e) {
System.out.println("recaught: " + e);
}
} }
```

В этом примере обработка исключения проводится в два приема. Метод main создает контекст для исключения и вызывает demoproc. Метод

demoproc также устанавливает контекст для обработки исключения, создает новый объект класса NullPointerException и с помощью оператора throw возбуждает это исключение. Исключение перехватывается в следующей строке внутри метода demoproc, причем объект-исключение доступен коду обработчика через параметр e. Код обработчика выводит сообщение о том, что возбуждено исключение, а затем снова возбуждает его с помощью оператора throw, в результате чего оно передается обработчику исключений в методе main. Ниже приведен результат, полученный при запуске этого примера.

C:\> java ThrowDemo
caught inside demoproc
recaught: java.lang.NullPointerException: demo

throws

Если метод способен возбуждать исключения, которые он сам не обрабатывает, он должен объявить о таком поведении, чтобы вызывающие методы могли защитить себя от этих исключений. Для задания списка исключений, которые могут возбуждаться методом, используется ключевое слово **throws**. Если метод в явном виде (т.е. с помощью оператора throw) возбуждает исключение соответствующего класса, тип класса исключений должен быть указан в операторе throws в объявлении этого метода. С учетом этого наш прежний синтаксис определения метода должен быть расширен следующим образом:

тип имя_метода(список аргументов) throws список_исключений {}

Ниже приведен пример программы, в которой метод procedure пытается возбудить исключение, не обеспечивая ни программного кода для его перехвата, ни объявления этого исключения в заголовке метода. Такой программный код не будет оттранслирован.

```
class ThrowsDemo1 {
static void procedure() {
System.out.println("inside procedure");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
procedure();
} }
```

Для того, чтобы мы смогли оттранслировать этот пример, нам придется сообщить транслятору, что procedure может возбуждать исключения типа IllegalAccessException и в методе main добавить код для обработки этого типа исключений :

```
class ThrowsDemo {
static void procedure() throws IllegalAccessException {
System.out.println(" inside procedure");
throw new IllegalAccessException("demo");
```

```
}
public static void main(String args[]) {
try {
procedure();
}
catch (IllegalAccessException e) {
System.out.println("caught " + e);
}
} }
```

Ниже приведен результат выполнения этой программы.

C:\> java ThrowsDemo

inside procedure

caught java.lang.IllegalAccessException: demo

finally

Иногда требуется гарантировать, что определенный участок кода будет выполняться независимо от того, какие исключения были возбуждены и перехвачены. Для создания такого участка кода используется ключевое слово finally. Даже в тех случаях, когда в методе нет соответствующего возбужденному исключению раздела catch, блок finally будет выполнен до того, как управление перейдет к операторам, следующим за разделом try. У каждого раздела try должен быть по крайней мере или один раздел catch или блок finally. Блок finally очень удобен для закрытия файлов и освобождения любых других ресурсов, захваченных для временного использования в начале выполнения метода. Ниже приведен пример класса с двумя методами, завершение которых происходит по разным причинам, но в обоих перед выходом выполняется код раздела finally.

```
class FinallyDemo {
static void procA() {
try {
System.out.println("inside procA");
throw new RuntimeException("demo");
}
finally {
System.out.println("procA's finally");
} }
static void procB() {
try {
System.out.println("inside procB");
return;
}
finally {
System.out.println("procB's finally");
} }
```

```
public static void main(String args[]) {
try {
procA();
}
catch (Exception e) {}
procB();
} }
```

В этом примере в методе procA из-за возбуждения исключения происходит преждевременный выход из блока try, но по пути «наружу» выполняется раздел finally. Другой метод procB завершает работу выполнением стоящего в try-блоке оператора return, но и при этом перед выходом из метода выполняется программный код блока finally. Ниже приведен результат, полученный при выполнении этой программы.

```
C:\> java FinallyDemo
inside procA
procA's finally
inside procB
procB's finally
```

Подклассы Exception

Только подклассы класса Throwable могут быть возбуждены или перехвачены. Простые типы — int, char и т.п., а также классы, не являющиеся подклассами Throwable, например, String и Object, использоваться в качестве исключений не могут. Наиболее общий путь для использования исключений — создание своих собственных подклассов класса Exception. Ниже приведена программа, в которой объявлен новый подкласс класса Exception.

```
class MyException extends Exception {
private int detail;
MyException(int a) {
detail = a:
}
public String toString() {
return "MyException[" + detail + "]";
}
}
class ExceptionDemo {
static void compute(int a) throws MyException {
System.out.println("called computer + a + ").");
if (a > 10)
throw new MyException(a);
System.out.println("normal exit.");
}
public static void main(String args[]) {
try {
```

```
compute(1);
compute(20);
}
catch (MyException e) {
System.out.println("caught" + e);
}
} }
```

Этот пример довольно сложен. В нем сделано объявление подкласса MyException класса Exception. У этого подкласса есть специальный конструктор, который записывает в переменную объекта целочисленное значение, и совмещенный метод toString, выводящий значение, хранящееся в объекте-исключении. Класс ExceptionDemo определяет метод compute, который возбуждает исключение типа MyExcepton. Простая логика метода compute возбуждает исключение в том случае, когда значение пара-ветра метода больше 10. Метод main в защищенном блоке вызывает метод compute сначала с допустимым значением, а затем — с недопустимым (больше 10), что позволяет продемонстрировать работу при обоих путях выполнения кода. Ниже приведен результат выполнения программы.

```
C:\> java ExceptionDemo
called compute(1).
normal exit.
called compute(20).
caught MyException[20]
```

Контрольные вопросы:

1. Приведите синтаксис записи генерации исключений?
2. Что происходит при генерации исключительной ситуации?
3. Какой оператор используется для задания списка исключений?
4. Для чего используется блок finally?
5. Укажите подклассы для обработки исключений.

Литература:

1. Смирнов Н.И. Java -2. Учебное пособие. М.:«Три Л», 2000. -320 с.
2. Арнольд К. и др. Язык программирования JAVA/ Вильямс. 2001. 624 с.
3. Брюс Эккель. Философия Java . Библиотека программиста. 2003.

ЛЕКЦИЯ 18. ТЕМА: КЛАССЫ ДЛЯ СТАНДАРНЫХ ТИПОВ

План: Простые оболочки для типов; Number; Double и Float; Бесконечность и NaN; Integer и Long; Character; Boolean; Большие числа.

Простые оболочки для типов

Как вы уже знаете, Java использует встроенные примитивные типы данных, например, int и char ради обеспечения высокой производительности. Эти типы данных не принадлежат к классовой иерархии Java. Они передаются методам по значению, передать их по ссылке невозможно. По этой причине для каждого примитивного типа в Java реализован специальный класс.

Number

Абстрактный класс Number представляет собой интерфейс для работы со всеми стандартными скалярными типами: — long, int, float и double.

У этого класса есть методы доступа к содержимому объекта, которые возвращают (возможно округленное) значение объекта в виде значения каждого из примитивных типов:

• doubleValue() возвращает содержимое объекта в виде значения типа double.

• floatValue() возвращает значение типа float.

• intValue() возвращает значение типа int.

• longValue() возвращает значение типа long.

Double и Float

Double и Float — подклассы класса Number. В дополнение к четырем методам доступа, объявленным в суперклассе, эти классы содержат несколько сервисных функций, которые облегчают работу со значениями double и float. У каждого из классов есть конструкторы, позволяющие инициализировать объекты значениями типов double и float, кроме того, для удобства пользователя, эти объекты можно инициализировать и объектом String, содержащим текстовое представление вещественного числа. Приведенный ниже пример иллюстрирует создание представителей класса Double с помощью обоих конструкторов.

```
class DoubleDemo {
public static void main(String args[]) {
Double d1 = new Double(3.14159);
Double d2 = new Double("314159E-5");
System.out.println(d1 + " = " + d2 + " -> " + d1.equals(d2));
} }
```

Как вы можете видеть из результата работы этой программы, метод equals возвращает значение true, а это означает, что оба использованных в примере конструктора создают идентичные объекты класса Double.

C:\> java DoubleDemo
3.14159 = 3.14159 -> true

Бесконечность и NaN

В спецификации IEEE для чисел с вещественной точкой есть два значения типа double, которые трактуются специальным образом: бесконечность и NaN (Not a Number — неопределенность). В классе Double есть тесты для проверки обоих этих условий, причем в двух формах — в виде методов (статических), которым значение double передается в качестве параметра, и в виде методов, проверяющих число, хранящееся в объекте класса Double.

• islnfinite(d) возвращает true, если абсолютное значение указанного числа типа double бесконечно велико.

• islnfinite() возвращает true, если абсолютное значение числа, хранящегося в данном объекте Double, бесконечно велико.

• isNaN(d) возвращает true, если значение указанного числа типа double неопределено.

• isNaN() возвращает true, если значение числа, хранящегося в данном объекте Double, неопределено.

Очередной наш пример создает два объекта Double, один с бесконечным, другой с неопределенным значением.

```
class InfNaN {
public static void main(String args[]) {
Double d1 = new Double(1/0.);
Double d2 = new Double(0/0.);
System.out.println(d1 + ": " + d1.isInfinite() + ", " + d1.isNaN());
System.out.println(d2 + ": " + d2.isInfinite() + ", " + d2.isNaN());
} }
```

Ниже приведен результат работы этой программы:

```
C:\> java InfNaN
Infinity: true, false
NaN: false, true
```

Integer и Long

Класс Integer — класс-оболочка для чисел типов int, short и byte, а класс Long — соответственно для типа long. Помимо наследуемых методов своего суперкласса Number, классы Integer и Long содержат методы для разбора текстового представления чисел, и наоборот, для представления чисел в виде текстовых строк. Различные варианты этих методов позволяют указывать основание (систему счисления), используемую при преобразовании. Обычно используются двоичная, восьмеричная, десятичная и шестнадцатиричная системы счисления.

- parseInt(String) преобразует текстовое представление целого числа, содержащееся в переменной String, в значение типа int. Если

строка не содержит представления целого числа, записанного в допустимом формате, вы получите исключение NumberFormatException.

- parseInt(String, radix) выполняет ту же работу, что и предыдущий метод, но в отличие от него с помощью второго параметра вы можете указывать основание, отличное от 10.
- toString(int) преобразует переданное в качестве параметра целое число в текстовое представление в десятичной системе.
- toString(int, radix) преобразует переданное в качестве первого параметра целое число в текстовое представление в задаваемой вторым параметром системе счисления.

Character

Character — простой класс-оболочка типа char. У него есть несколько полезных статических методов, с помощью которых можно выполнять над символом различные проверки и преобразования.

- isLowerCase(char ch) возвращает true, если символ-параметр принадлежит нижнему регистру (имеется в виду не просто диапазон a-z, но и символы нижнего регистра в кодировках, отличных от ISO-Latin-1).
- isUpperCase(char ch) делает то же самое в случае символов верхнего регистра.
- isDigit(char ch) и isSpace(char ch) возвращают true для цифр и пробелов, соответственно.
- toLowerCase(char ch) и toupperCase(char ch) выполняют преобразования символов из верхнего в нижний регистр и обратно.

Boolean

Класс Boolean — это очень тонкая оболочка вокруг логических значений, она бывает полезна лишь в тех случаях, когда тип boolean требуется передавать по ссылке, а не по значению.

Большие числа

Если вам недостаточно точности встроенных целочисленных типов и чисел с плавающей точкой, можно обратиться к двум полезным классам в пакете Java .math под названием Biglnteger и BigDecimal. Эти классы предназначены для манипуляций с числами, состоящими из произвольного количества цифр. Классы Biglnteger и BigDecimal реализуют арифметические операции произвольной точности для целых и действительных чисел соответственно.

Для преобразования обычного числа в большое используется статический метод

valueOf:

Biglnteger a = Biglnteger.valueof(100);

К сожалению, к большим числам нельзя применять обычные математические операторы + и *. Вместо этого программист должен использовать методы add и multiply из классов для работы с большими числами.

Biglnteger c = a.add(b); // c = a + b
Biglnteger d = c.multiply(b.add(BigInteger.value.Of(2)));
// d - c * (b + 2)

В листинге 3.6 показана модификация программы для подсчета шансов выиграть в лотерее, приведенной в листинге 3.5. Теперь эта программа может работать с большими числами. Например, если вам предложили сыграть в лотерее, в которой нужно угадать 60 чисел из 490 возможных, то эта программа сообщит вам, что шанс выиграть равен 1 из

716395843461995557415116222540092933411717612789263493493351013459481104668848.

Удачи!

Программа", представленная в листинге 3.5, вычисляла следующий оператор:

lotteryOdds = lottery * (n - i + 1) / i;

При работе с большими числами эквивалентный оператор выглядит так.

lotteryOdds = lotteryOdds.multiply(Biglnteger.valueOf(n-i+1)
.divide{Biglnteger.valueOf(i));

```
import javax.swing.*;
import Java.math.*;
public class BiglntegerTest
{
public static void main(String[] args)
{
String input = JOptionPane.showlnputDialog
("Сколько номеров нужно угадать?");
int k = Integer.parselnt(input);
input = JOptionPane.showlnputDialog
("Чему равен наибольший из возможных номеров?");
int n = Integer.parselnt(input);
Вычисление биномиальных коэффициентов
п * (п - 1) * (п - 2) • * . . . * (п - к + 1)
Biglnteger lotteryOdds = Biglnteger.valueOf(1);
for (int i = 1; i <= k;
lotteryOdds = lotteryOdds
.multiply(Biglnteger.valueOf(n - i + 1))
.divide(Biglnteger.valueOf(i));
System.out.println("Ваш шанс равен 1 из " + lotteryOdds
+ ". Удачи!");
```

System.exit(0) ;

Контрольные вопросы

1. Для чего используются специальные классы для типов?
2. Укажите абстрактный класс, который представляет собой интерфейс для работы со всеми стандартными скалярными типами.
3. Укажите классы для работы с типами.
4. Какие классы используются для работы с большими числами?
5. Укажите методы для сложения и умножения больших чисел.

Литература:

1. Смирнов Н.И. Java -2. Учебное пособие. М.:«Три Л», 2000. -320 с.
2. Арнольд К. и др. Язык программирования JAVA/ Вильямс. 2001. 624 с.
3. Брюс Эккель. Философия Java . Библиотека программиста. 2003.

ЛЕКЦИЯ 19. ТЕМА: СТАНДАРТНЫЕ КЛАССЫ ДЛЯ АБСТРАКТНЫХ ТИПОВ ДАННЫХ

План: Класс вектор; Абстрактный класс словарь; Класс для хэш таблицы.

Vector

Vector — это способный увеличивать число своих элементов массив ссылок на объекты. Внутри себя Vector реализует стратегию динамического расширения, позволяющую минимизировать неиспользуемую память и количество операций по выделению памяти. Объекты можно либо записывать в конец объекта Vector с помощью метода addElement, либо вставлять в указанную индексом позицию методом insertElementAt. Вы можете также записать в Vector массив объектов, для этого нужно воспользоваться методом copyInto. После того, как в Vector записана коллекция объектов, можно найти в ней индивидуальные элементы с помощью методов Contains, indexOf и lastIndexOf. Кроме того методы elementAt, firstElement и lastElement позволяют извлекать объекты из нужного положения в объекте Vector.

Stack

Stack — подкласс класса Vector, который реализует простой механизм типа “первым вошел — первым вышел" (FIFO). В дополнение к стандартным методам своего родительского класса, Stack предлагает метод push для помещения элемента в вершину стека и pop для извлечения из него верхнего элемента. С помощью метода peek вы можете получить верхний элемент, не удаляя его из стека. Метод empty служит для проверки стека на наличие элементов — он возвращает true, если стек пуст. Метод search ищет заданный элемент в стеке, возвращая количество операция pop, которые требуются для того чтобы перевести искомый элемент в вершину стека. Если заданный элемент в стеке отсутствует, этот метод возвращает -1.

Ниже приведен пример программы, которая создает стек, заносит в него несколько объектов типа Integer, а затем извлекает их.

```
import java.util.Stack;
import java.util.EmptyStackException;
class StackDemo {
static void showpush(Stack st, int a) {
st.push(new Integer(a));
System.out.println("push(" + a + ")");
System.out.println("stack: " + st);
}
```

```
static void showpop(Stack st) {
System.out.print("pop -> ");
Integer a = (Integer) st.pop();
System.out.println(a);
System.out.println("stack: " + st);
}
public static void main(String args[]) {
Stack st = new Stack();
System.out.println("stack: " + st);
showpush(st, 42);
showpush(st, 66);
showpush(st, 99);
showpop(st);
showpop(st);
showpop(st);
try {
showpop(st);
}
catch (EmptyStackException e) {
System.out.println("empty stack");
} }
}
```

Ниже приведен результат, полученный при запуске этой программы. Обратите внимание на то, что обработчик исключений реагирует на попытку извлечь данные из пустого стека. Благодаря этому мы можем аккуратно обрабатывать ошибки такого рода.

```
C:\> java StackDemo
stack: []
push(42)
stack: [42]
push(66)
stack: [42, 66]
push(99)
stack: [42, 66, 99]
```

pop -> 99

stack: [42, 66]

pop -> 66

stack: [42]

pop -> 42

stack: []

pop -> empty stack

Dictionary

Dictionary (словарь) — абстрактный класс, представляющий собой хранилище информации типа “ключ-значение”. Ключ — это имя, по которому осуществляется доступ к значению. Имея ключ и значение, вы можете записать их в словарь методом put(key, value). Для получения значения по заданному ключу служит метод get(key). И ключи, и значения можно получить в форме перечисления (объект Enumeration) методами keys и elements. Метод size возвращает количество пар “ключ-значение”, записанных в словаре, метод isEmpty возвращает true, если словарь пуст. Для удаления ключа и связанного с ним значения предусмотрен метод remove(key).

HashTable

HashTable — это подкласс Dictionary, являющийся конкретной реализацией словаря. Представителя класса HashTable можно использовать для хранения произвольных объектов, причем для индексации в этой коллекции также годятся любые объекты. Наиболее часто HashTable используется для хранения значений объектов, ключами которых служат строки (то есть объекты типа String). В очередном нашем примере в HashTable хранится информация об этой книге.

```
import java.util.Dictionary;
import java.util.Hashtable;
class HTDemo {
public static void main(String args[]) {
Hashtable ht = new Hashtable();
ht.put("title", "The Java Handbook");
ht.put("author", "Patrick Naugnton");
ht.put("email", "naughton@starwave.com");
ht.put(“age", new Integer(30));
show(ht);
```

```
}
static void show(Dictionary d) {
System.out.println("Title: " + d.get("title"));
System.out.println("Author: " + d.get("author"));
System.out.println("Email: " + d.get("email"));
System.out.println("Age: " + d.get("age"));
} }
```

Результат работы этого примера иллюстрирует тот факт, что метод show, параметром которого является абстрактный тип Dictionary, может извлечь все значения, которые мы занесли в ht внутри метода main.

```
C:\> java HTDemo
Title: The Java Handbook
Author: Patrick Naughton
Email: naughton@starwave.com
Age: 30
```

Properties

Properties — подкласс HashTable, в который для удобства использования добавлено несколько методов, позволяющих получать значения, которые, возможно, не определены в таблице. В методе getProperty вместе с именем можно указывать значение по умолчанию:

```
getPгoperty("имя","значение_по_умолчанию");
```

При этом, если в таблице свойство “имя” отсутствует, метод вернет “значение_по_умолчанию”. Кроме того, при создании нового объекта этого класса конструктору в качестве параметра можно передать другой объект Properties, при этом его содержимое будет использоваться в качестве значений по умолчанию для свойств нового объекта. Объект Properties в любой момент можно записать либо считать из потока — объекта Stream. Ниже приведен пример, в котором создаются и впоследствии считываются некоторые свойства:

```
import java.util.Properties;
class PropDemo {
static Properties prop = new Properties();
public static void main(String args[]) {
prop.put("Title", "put title here");
prop.put("Author", "put name here");
prop.put("isbn", "isbn not set");
Properties book = new Properties(prop);
```

book.put("Title", "The Java Handbook");

book.put("Author", "Patrick Naughton");

System.out.println("Title: " +

book.getProperty("Title"));

System.out.println("Author: " +

book.getProperty("Author"));

System.out.println("isbn: " +

book.getProperty("isbn"));

System.out.println("ean: " +

book.getProperty("ean", "???"));

} }

Здесь мы создали объект prop класса Properties, содержащий три значения по умолчанию для полей Title, Author и isbn. После этого мы создали еще один объект Properties с именем book, в который мы поместили реальные значения для полей Title и Author. В следующих трех строках примера мы вывели результат, возвращенный методом getProperty для всех трех имеющихся ключей. В четвертом вызове getProperty стоял несуществующий ключ “ean”. Поскольку этот ключ отсутствовал в объекте book и в объекте по умолчанию prop, метод getProperty выдал нам указанное в его вызове значение по умолчанию, то есть “???”:

C:\> java PropDemo

Title: The Java Handbook

Author: Patrick Naughton

isbn: isbn not set

ean: ???

Контрольные вопросы

1. Для чего используются класс вектор?
2. Укажите методы класса вектор.
3. Какой механизм использует стек?
4. Укажите абстрактный класс представляющий словарь.
5. Укажите подклассы класса словарь.

Литература:

1. Смирнов Н.И. Java -2. Учебное пособие. М.:«Три Л», 2000. -320 с.
2. Арнольд К. и др. Язык программирования JAVA/ Вильямс. 2001. 624 с.
3. Брюс Эккель. Философия Java . Библиотека программиста. 2003.

ОГЛАВЛЕНИЕ

№ т.	Название темы	Стр.
1.	Основные конструкции.	3
2.	Операторы.	9
3.	Массивы.	16
4.	Строки.	22
5.	Особенности использования строк.	28
6.	Классы и объекты.	33
7.	Особенности использования классов и объектов.	38
8.	Наследование в классах.	44
9.	Особенности использования наследования.	50
10.	Интерфейсы.	55
11.	Пакеты.	59
12.	Встроенные классы.	64
13.	Работа с файлами.	70
14.	Особенности работы с файлами.	76
15.	Управление исключительными ситуациями.	81
16.	Особенности управления исключительными ситуациями.	86
17.	Классы для стандартных типов.	91
18.	Стандартные классы для абстрактных типов данных	96