МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего профессионального образования
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

О.Б. Фофанов

ИНФОРМАТИКА И ПРОГРАММИРОВАНИЕ

Часть 1

Рекомендовано
Редакционно-издательским советом Томского политехнического университета в качестве учебного пособия для студентов бакалавриата, обучающихся по направлению «Программная инженерия»

Издательство
Томского политехнического университета
2014

УДК 004+004.4(075.8)
ББК 32.81е73
Ф817

Фофанов О.Б.

Ф817 Информатика и программирование. Ч. 1: учебное пособие/О.Б. Фофанов; Томский политехнический университет.– Томск: Изд-во Томского политехнического университета, 2013. – 140 с.

ISBN 978-5-4387-0369-3

В пособии кратко изложены базовые конструкции универсального языка программирования Java, рассмотрена реализация в Java принципов объектно-ориентированного проектирования и программирования – инкапсуляции, полиморфизма и наследования. Подробно рассмотрена структура классов и методов, реализация абстрактных классов и интерфейсов, а также описаны основные конструкции и механизмы обработки исключительных ситуаций.

Предназначено для студентов бакалавриата, обучающихся по направлениям «Программная инженерия», «Информатика и вычислительная техника» и «Прикладная информатика».

УДК 004+004.4(075.8)
ББК 32.81е73

Рецензенты

Доктор технических наук, доцент
заведующий кафедрой БИС ТУСУРа
Р.В. Мещеряков

Кандидат технических наук, доцент ТУСУРа
Н.Ю. Хабибулина

ISBN 978-5-4387-0369-3

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Данное учебное пособие ориентировано на формирование у студентов навыков и знаний в области программирования на языке Java, который рассматривается в данном пособии как универсальный язык программирования высокого уровня, а также на понимание и усвоение в ходе изучения парадигм объектно-ориентированного проектирования и программирования.

Официальный день рождения технологии Java – 23 мая 1995г. и первоначально она предназначалась для программирования бытовых электронных устройств, таких как телефоны. Из-за сходства характеристик Java и C, кое кто склонен считать Java просто "Internet версией C++". Однако это серьезное заблуждение. Языку Java присущи значительные практические и концептуальные отличия. Хотя и верно, что C++ оказал влияние на характеристики языка Java, последний не является усовершенствованной версией C++. Например, Java не обладает совместимостью с C++ ни по восходящей, ни по нисходящей линии. Конечно, сходство с языком C++ значительно, и в программе Java программист C++ будет чувствовать себя как дома. Вместе с тем, Java не предназначен служить заменой C++. Java был предназначен для решения одного набора проблем, а C++ для решения другого. Еще длительное время оба эти языка неизбежно будут сосуществовать.

Изменение информационной и технической среды, которое обусловило потребность в языке, подобном Java , была потребность в не зависящих от платформы программах, предназначенных для распространения в Internet. Однако Java изменяет также подход к написанию программ. В частности, Java углубил и усовершенствовал объектно-ориентированный подход, использованный в C++, добавил в него поддержку многопоточной обработки и предоставил библиотеку, которая упростила доступ к Internet. Однако столь поразительный успех Java обусловлен не теми или иными его отдельными особенностями, а их совокупностью как языка в целом. Он явился прекрасным ответом на потребности в то время лишь зарождающейся среды в высшей степени распределенных компьютерных систем. Для программирования Internet-программ Java стал тем, чем язык C был для системного программирования: революционной силой, которая изменила мир.

Начиная с 1995г. в среднем в год появлялись 2 новых версии Java (первая – Java 1.0), базовой на долгое время стала версия Java 2 (1.2-1.5).

Новейшая (на момент подготовки этого издания) версия Java получила название SE 7. С выходом Java SE 6 компания Sun решила в очередной раз изменить название платформы Java . Во-первых, цифра 2 в названии была опущена. Таким образом, теперь платформа называется Java SE, а официльное название продукта Java Platfonn, Standard Edition 7 (Платформа Java, стандартная версия 7). Внутренним номером разработки этой версии является 1.7.

Версия Java SE 7 (релиз версии состоялся 28 июля 2011 года) построена на основе предыдущих версий и добавляет к ней ряд дальнейших усовершенствований, они должны значительно расширить выразительность языка (например в работе с коллекциями) и привнести некоторую долю функционального программирования в Java . Она расширяет библиотеки API, добавляет несколько новых пакетов и предоставляет ряд усовершенствований времени выполнения

В новых версиях Java Enterprise и Standard Edition не будет некоторых обещанных возможностей. Однако, как заявляют в Oracle, обновления все же заслуживают внимания благодаря большому числу улучшений. Java EE 7 планируется выпустить во втором квартале 2013 года, а Java SE 8 – до его окончания, сообщили представители Oracle на конференции Java One. В корпоративной редакции не будет улучшений, рассчитанных на применение Java как облачной платформы в виде сервиса. Эти особенности появятся позднее, примерно в 2015 году. Но как обещают в Oracle, EE 7 станет проще для использования программистами: благодаря поддержке внедрения зависимостей можно будет исключить повторяющийся код. Появится также поддержка протокола WebSocket для связи с HTML5-приложениями и интерфейса программирования Java API for RESTful Web Services 2.0.

В Java SE 8 не будет средств модульности, реализуемых в рамках проекта Jigsaw, но в JDK 8 будет возможность программирования на Java Script (проект Nashorn). Кроме того, добавлены особенности из проекта Lambda по поддержке многоядерных процессоров. В Oracle также рассказали о проекте Easel по созданию HTML5-инструментария для NetBeans и о совместной работе с AMD над возможностью использования в JDK вычислительных функций графических чипов.

Данное пособие не описывает полностья язык программирования Java и предназначено для самостоятельного изучения ядра языка, начиная с лексической структуры языка и заканчивая обработкой исключительных ситуаций. Пособие ориентировано на студентов знакомых с основами программрования хотя бы на уровне языка Pascal

или Basic. В дисциплине Информатика и программирование для направления 230700 Программная инженерия (второй – третий семестры) подробно рассматриваются более сложные элементы технологии Java – коллекции, работа с потоками, разработка графического интерфейса и аплетов, введение в Java Beans.

1. ЛЕКСИКА ОСНОВНЫЕ ОПЕРАЦИИ ЯЗЫКА JAVA

1.1. Кодировка

Технология Java , как платформа, изначально спроектированная для Глобальной сети Internet, должна быть многоязыковой, а значит, обычный набор символов ASCII (American Standard Code for Information Interchange, Американский стандартный код обмена информацией), включающий в себя лишь латинский алфавит, цифры и простейшие специальные знаки (скобки, знаки препинания, арифметические операции и т.д.), недостаточен. Поэтому для записи текста программы применяется более универсальная кодировка Unicode.

Как известно, Unicode представляет символы кодом из 2 байт, описывая, таким образом, 65535 символов. Это позволяет поддерживать практически все языки мира. Первые 128 символов совпадают с набором ASCII. Однако понятно, что требуется некоторое специальное обозначение, чтобы иметь возможность задавать в программе любой символ Unicode, ведь никакая клавиатура не позволяет вводить более 65 тысяч различных знаков.

. Официальный web-сайт стандарта, где можно получить дополнительную информацию,– http://www.unicode.org/.

Компилятор, анализируя программу, сразу разделяет ее на:

- пробелы (white spaces);
- комментарии (comments);
- основные лексемы (tokens).

1.2. Комментарии

Комментарии не влияют на результирующий бинарный код и используются только для ввода пояснений к программе.

В Java комментарии бывают двух видов:

- строчные
- блочные

Строчные комментарии начинаются с ASCII-символов // и длятся до конца текущей строки. Как правило, они используются для пояснения именно этой строки, например:

int y=1970; // год рождения

Блочные комментарии располагаются между ASCII-символами /* и */, могут занимать произвольное количество строк.

Блочный комментарий не обязательно должен располагаться на нескольких строках, он может даже находиться в середине оператора:

float s = 2*Math.PI/*getRadius()*/;

// Закомментировано для отладки

Комментарии не могут находиться в символьных и строковых литералах, идентификаторах.

Комментарии не могут быть вложенными.

Любые комментарии полностью удаляются из программы во время компиляции, поэтому их можно использовать неограниченно, не опасаясь, что это повлияет на бинарный код.

Кроме этого, существует особый вид блочного комментария – комментарий разработчика. Он применяется для автоматического создания документации кода. В стандартную поставку JDK, начиная с версии 1.0, входит специальная утилита Java doc.

1.3. Лексемы

Ниже перечислены все виды лексем в Java:

- идентификаторы (identifiers);
- ключевые слова (key words);
- литералы (literals);
- разделители (separators);
- операторы (operators).

1.3.1. Идентификаторы

Идентификаторы – это имена, которые даются различным конструкциям языка для упрощения доступа к ним. Имена имеют пакеты, классы, интерфейсы, поля, методы, аргументы и локальные переменные (все эти понятия подробно рассматриваются в следующих разделах). Идентификаторы можно записывать символами Unicode, то есть на любом удобном языке. Длина имени не ограничена.

Идентификатор состоит из букв и цифр. Имя не может начинаться с цифры. Java -буквы, используемые в идентификаторах, включают в себя ASCII-символы A-Z (\u0041-\u005a), a-z (\u0061-\u007a), а также

знаки подчеркивания _ (ASCII underscore, \u005f) и доллара $ (\u0024). Знак доллара используется только при автоматической генерации кода (чтобы исключить случайное совпадение имен), либо при использовании каких-либо старых библиотек, в которых допускались имена с этим символом. Java -цифры включают в себя обычные ASCII-цифры 0-9 (\u0030-\u0039).

Для идентификаторов не допускаются совпадения с зарезервированными словами (это ключевые слова, булевские литералы true и false и null-литерал null). Конечно, если 2 идентификатора включают в себя разные буквы, которые одинаково выглядят (например, латинская и русская буквы A), то они считаются различными.

Примеры идентификаторов:

Character, a, b, c, D, x1, x2, Math, sqrt, x,
y, i, s, PI, getRadius, circle, getAbs,
calculate, condition, getWidth, getHeight,
Java , lang, String

Также допустимыми являются идентификаторы:

Компьютер,ももも,鷲鷲鷲,COLOR_RED,
aVeryLongNameOfTheMethod

1.3.2. Ключевые слова

Ключевые слова – это зарезервированные слова, состоящие из ASCII-символов и выполняющие различные задачи языка. Вот их полный список (48 слов):

abstract	double	int	strictfp
boolean	else	interface	super
break	extends	long	switch
byte	final	native	synchronized
case	finally	new	this
catch	float	package	throw
char	for	private	throws
class	goto	protected	transient
const	if	public	try
continue	implements	return	void
default	import	short	volatile
do	instanceof	static	while

Ключевые слова goto и const зарезервированы, но не используются. Это сделано для того, чтобы компилятор мог правильно отреагировать на их использование в других языках.

1.3.3. Литералы

Литералы позволяют задать в программе значения для числовых, символьных и строковых выражений, а также null-литералов [4]. Всего в Java определены следующие виды литералов:

- целочисленный (integer);
- вещественный (floating-point);
- булевский (boolean);
- символьный (character);
- строковый (string);
- null-литерал (null-literal).

Рассмотрим их по отдельности.

Целочисленные литералы

Целочисленные литералы позволяют задавать целочисленные значения в десятеричном, восьмеричном и шестнадцатеричном виде. Десятеричный формат традиционен и ничем не отличается от правил, принятых в других языках. Значения в восьмеричном виде начинаются с нуля, и, конечно, использование цифр 8 и 9 запрещено. Запись шестнадцатеричных чисел начинается с 0x или 0X (цифра 0 и латинская ASCII-буква X в произвольном регистре). Таким образом, ноль можно записать тремя различными способами:

0
00
0x0

Как обычно, для записи цифр 10-15 в шестнадцатеричном формате используются буквы A, B, C, D, E, F, заглавные или прописные. Примеры таких литералов:

0xaBcDeF, 0xCafe, 0xDEC

Логические литералы

Логические литералы имеют два возможных значения – true и false. Эти два зарезервированных слова не являются ключевыми, но также не могут использоваться в качестве идентификатора.

Символьные литералы

Символьные литералы описывают один символ из набора Unicode, заключенный в одиночные кавычки, или апострофы (ASCII-символ single quote, \u0027). Например:

'a' // латинская буква a

' ' // пробел

'K' // греческая буква каппа

Также допускается специальная запись для описания символа через его код (см. тему "Кодировка"). Примеры:

'\u0041' // латинская буква A

'\u0410' // русская буква A

'\u0391' // греческая буква A

Символьный литерал должен содержать строго один символ, или специальную последовательность, начинающуюся с \. Для записи специальных символов (неотображаемых и служебных, таких как ", ', \) используются следующие обозначения:

\b \u0008 backspace BS – забой

\t \u0009 horizontal tab HT – табуляция

\n \u000a linefeed LF – конец строки

\f \u000c form feed FF – конец страницы

\r \u000d carriage return CR –
возврат каретки

\" \u0022 double quote " – двойная кавычка

\' \u0027 single quote ' – одинарная кавычка

\\ \u005c backslash \ – обратная косая черта

\восьмеричный код от \u0000 до \u00ff символа
в восьмеричном формате.

Строковые литералы

Строковые литералы состоят из набора символов и записываются в двойных кавычках. Длина может быть нулевой или сколь угодно большой. Любой символ может быть представлен специальной последовательностью, начинающейся с \ (см. "Символьные литералы").

"" // литерал нулевой длины

"\"" //литерал, состоящий из одного символа "

"Простой текст" //литерал длины 13

Строковый литерал нельзя разбивать на несколько строк в тексте программы. Если требуется текстовое значение, состоящее из нескольких строк, то необходимо воспользоваться специальными символами \n и/или \r. Если же текст просто слишком длинный, чтобы уместиться на одной строке кода, можно использовать оператор конкатенации строк +.

Каждый строковый литерал является экземпляром класса String. Это определяет некоторые необычные свойства строковых литералов, которые будут рассмотрены в следующем разделе.

Null-литерал

Null-литерал может принимать всего одно значение: null. Это литерал ссылочного типа, причем эта ссылка никуда не ссылается, объект отсутствует. Его можно применять к ссылкам любого объектного типа данных. Типы данных подробно рассматриваются в следующемразделе.

Разделители

Разделители – это специальные символы, которые используются в служебных целях языка. Назначение каждого из них будет рассмотрено по ходу изложения курса. Вот их полный список:

( ) [ ] { } ; . ,

1.3.4. Операторы

Операторы используются в различных операциях – арифметических, логических, битовых, операциях сравнения и присваивания. Следующие 37 лексем (все состоят только из ASCII-символов) являются операторами языка Java :

= > < ! ~ ? :

== <= >= != && || ++ --

+ - *= / & | ^ % << >> >>>

+= -= * /= &= |= ^= %= <<= >>= >>>=

Большинство из них вполне очевидны и хорошо известны из других языков программирования, однако некоторые нюансы в работе с операторами в Java все же присутствуют, поэтому в конце раздела приводятся краткие комментарии к ним.

Пример программы

В заключение для примера приведем простейшую программу (традиционное Hello, world!), а затем классифицируем и подсчитаем используемые лексемы:

```
public class Demo {
/**
 * Основной метод, с которого начинается
 * выполнение любой Java  программы.
 */
 public static void main (String args[])
 {
 System.out.println("Hello, world!");
```

```
 }
}
```

Итак, в приведенной программе есть один комментарий разработчика, 7 идентификаторов, 5 ключевых слов, 1 строковый литерал, 13 разделителей и ни одного оператора. Этот текст можно сохранить в файле Demo.Java , скомпилировать и запустить. Результатом работы будет, как очевидно:

Hello, world!

Работа с операторами

Рассмотрим некоторые детали использования операторов в Java . Здесь будут описаны подробности, относящиеся к работе самих операторов.

Операторы присваивания и сравнения

Во-первых, конечно же, различаются оператор присваивания = и оператор сравнения ==.

```
x = 1; // присваиваем переменной x значение 1
x == 1 // сравниваем значение переменной x с
 // единицей
```

Оператор сравнения всегда возвращает булевское значение true или false. Оператор присваивания возвращает значение правого операнда.

Арифметические операции

Наряду с четырьмя обычными арифметическими операциями +, -, *, /, существует оператор получения остатка от деления %, который может быть применен как к целочисленным аргументам, так и к вещественным.

Работа с целочисленными аргументами подчиняется простым правилам. Если делится значение a на значение b, то выражение (a/b)*b+(a%b) должно в точности равняться a. Здесь, конечно, оператор деления целых чисел / всегда возвращает целое число. Например:

9/5 возвращает 1

9/(-5) возвращает -1

(-9)/5 возвращает -1

(-9)/(-5) возвращает 1

Остаток может быть положительным, только если делимое было положительным. Соответственно, остаток может быть отрицательным только в случае отрицательного делимого.

9%5 возвращает 4

9%(-5) возвращает 4

(-9)%5 возвращает -4

(-9)%(-5) возвращает -4

Унарные операторы инкрементации ++ и декрементации --, как обычно, можно использовать как справа, так и слева.

```
int x=1;
int y=++x;
```

В этом примере оператор ++ стоит перед переменной x, это означает, что сначала произойдет инкрементация, а затем значение x будет использовано для инициализации y. В результате после выполнения этих строк значения x и y будут равны 2.

```
int x=1;
int y=x++;
```

А в этом примере сначала значение x будет использовано для инициализации y, и лишь затем произойдет инкрементация. В результате значение x будет равно 2, а y будет равно 1.

Логические операции

Логические операторы "и" и "или" (& и |) можно использовать в двух вариантах. Это связано с тем, что для каждого оператора возможны случаи, когда значение первого операнда сразу определяет значение всего логического выражения. Если вторым операндом является значение некоторой функции, то появляется выбор – вызывать ее или нет, причем это решение может сказаться как на скорости, так и на функциональности программы.

Первый вариант операторов (&, |) всегда вычисляет оба операнда, второй же – (&&, ||) не будет продолжать вычисления, если значение выражения уже очевидно.

Логический оператор отрицания "не" записывается как ! и, конечно, имеет только один вариант использования. Этот оператор меняет булевское значение на противоположное.

```
int x=1;
x>0 // выражение истинно
!(x>0) // выражение ложно
```

Оператор с условием ? : состоит из трех частей – условия и двух выражений. Сначала вычисляется условие (булевское выражение), а на основании результата значение всего оператора определяется первым выражением в случае получения истины и вторым – если условие ложно. Например, так можно вычислить модуль числа x:

```
x>0 ? x : -x
```

Битовые операции

Прежде чем переходить к битовым операциям, необходимо уточнить, каким именно образом целые числа представляются в

двоичном виде. Конечно, для неотрицательных величин это практически очевидно:

0 0
1 1
2 10
3 11
4 100
5 101

и так далее. Однако как представляются отрицательные числа? Во-первых, вводят понятие знакового бита. Первый бит начинает отвечать за знак, а именно 0 означает положительное число, 1 – отрицательное. Но не следует думать, что остальные биты остаются неизменными.

Рассмотрим операторы побитового сдвига. В Java есть один оператор сдвига влево и два варианта сдвига вправо. Такое различие связано с наличием знакового бита.

При сдвиге влево оператором << все биты числа смещаются на указанное количество позиций влево, причем освободившиеся справа позиции заполняются нулями. Эта операция аналогична умножению на 2^n и действует вполне предсказуемо, как при положительных, так и при отрицательных аргументах.

Рассмотрим примеры применения операторов сдвига для значений типа int, т.е. 32-битных чисел. Пусть положительным аргументом будет число 20, а отрицательным -21.

Как видно из примера, неожиданности возникают тогда, когда значащие биты начинают занимать первую позицию и влиять на знак результата.

При сдвиге вправо все биты аргумента смещаются на указанное количество позиций, соответственно, вправо. Возникает вопрос – каким значением заполнять освобождающиеся позиции слева, в том числе и отвечающую за знак. Есть два варианта. Оператор >> использует для заполнения этих позиций значение знакового бита, то есть результат всегда имеет тот же знак, что и начальное значение. Второй оператор >>> заполняет их нулями, то есть результат всегда положительный.

Очевидно, что для положительного аргумента операторы >> и >>> работают совершенно одинаково. Дальнейший сдвиг на большее количество позиций будет также давать нулевой результат.

Очевидно – эти операции аналогичны делению на 2^n. Причем, если для положительных аргументов с ростом n результат закономерно стремится к 0, то для отрицательных предельным значением является -1.

2. ТИПЫ ДАННЫХ В JAVA

Прежде всего, важно уяснить, что Java строго типизированный язык[2,4]. Действительно, в определенной степени безопасность и надежность Java программ обусловлена именно этим обстоятельством. Давайте разберемся, что это означает. Во-первых, каждая переменная обладает типом, каждое выражение имеет тип, и каждый тип строго определен. Во-вторых, все присваивания, как явные, так и посредством передачи параметров в вызовах методов, проверяются на соответствие типов. В Java отсутствуют, какие либо средства автоматического приведения или преобразования конфликтующих типов, как это имеет место в некоторых языках. Компилятор Java проверяет все выражения и параметры на предмет совместимости типов. Любые несоответствия типов являются ошибками, которые должны быть исправлены до завершения компиляции класса.

2.1. Примитивные типы

Java определяет восемь элементарных типов данных: byte, short, int, long, char, float, double и boolean.

Часто элементарные типы называют также простыми или примитивными типами. Элементарные типы можно разделить на четыре группы.

. ***Целочисленные.*** Эта группа включает в себя типы byte, short, int и long, которые представляют точные целые числа со знаком.

Числа с плавающей точкой. Эта группа включает в себя типы float и double, которые представляют числа, определенные с точностью до определенного десятичного знака.

Символы. Эта группа включает в себя тип char, которая представляет символы символьного набора, такие как буквы и цифры.

Булевские значения. Эта группа включает в себя тип boolean специальный тип, предназначенный для представления значений типа истинно/ложно.

Эти типы можно использовать в том виде, как они определены, или же для создания собственных типов классов. Таким образом, они служат основой для всех других типов данных, которые могут быть созданы.

Элементарные типы представляют одиночные значения, а не сложные объекты. Хотя во всех других отношениях Java полностью объектно-ориентированный язык, элементарные типы данных таковыми не являются. Они аналогичны простым типам, которые можно встретить в большинстве других не объектно-ориентированных языков.

Эта особенность обусловлена стремлением обеспечить максимальную эффективность. Превращение элементарных типов в объекты привело бы к слишком большому снижению производительности.

Элементарные типы определены так, чтобы они обладали явной областью допустимых значений и математически строгим поведением. Языки вроде C и C++ допускают варьирование размеров целочисленных переменных в зависимости от требований среды выполнения. Однако java отличается в этом отношении. В связи с требованием переносимости, предъявляемым к java-программам, все типы данных обладают строго определенной областью допустимых значений. Например, независимо от конкретной платформы, значения типа int всегда являются 32битными. Это позволяет создавать программы, которые гарантированно будут выполняться в любой машинной архитектуре без cпеци альною пepeнoca. Хотя в некоторых средах строгое указание размера целых чисел может приводить к незначительному снижению производительности, оно абсолютно необходимо для обеспечения переносимости программ.

Рассмотрим каждый и типов данных.

2.1.1. Целочисленные значения

Java определяет четыре целочисленных типа: byte, short, int и long. Все эти типы представляют значения со знаком положительные и отрицательные. Java не поддерживает только положительные целочисленные значения без знака. Многие другие языки программирования поддерживают целочисленные значения как со знаком, так и без знака. Однако разработчики Java посчитали целочисленные значения без знака не нужными.

В частности, они решили, что концепция значений без знака использовалась, в основном, для указания поведения старшего бита, который определяет знак целочисленного значения. Как будет показано ниже, в Java управление значением старшего бита осуществляется иначе посредством применения специальной операции "сдвига вправо без учета знака". Тем самым потребность в целочисленном типе без знака была исключена.

Размер целочисленного типа представляет не занимаемый объем памяти, а скорее поведение, определяемое им для переменных и выражений этого типа. Среда времени выполнения Java может использовать любой размер, до тех пор, пока типы ведут себя

объявленным образом. Размер и область допустимых значений показаныв табл. 2.1

Таблица 2.1. Целочисленные типы данных.

Название типа	Длина (байты)	Область значений
byte	1	-128 .. 127
short	2	-32.768 .. 32.767
int	4	-2.147.483.648 .. 2.147.483.647
long	8	-9.223.372.036.854.775.808 .. 9.223.372.036.854.775.807 (примерно 10^{19})
char	2	'\u0000' .. '\uffff', или 0 .. 65.535

byte

Наименьший по размеру целочисленный тип. Это 8- битный тип со знаком с областью допустимых значений от 128 до 127. Переменные типа byte особенно полезны при работе с потоком данных, поступающих из сети или файла. Они полезны также при манипулировании необработанными двоичными данными, которые могут не быть непосредственно совместимыми с другими встроенными типами Java .

Для объявления переменных типа byte служит ключевое слово byte. Например, byte b, c;

short

short – 16-битный тип со знаком. Он имеет область допустимых значений от 32768 до 32767. Вероятно, этот тип используется в Java наименее часто. Ниже приведено Несколько примеров объявления переменных типа short:

short s;

short t;

int

Наиболее часто используемым целочисленным типом является int. Это 32 битный тип со знаком, который имеет область допустимых значений от 2147483648 до 2147483647. Кроме других применений переменные типа int часто применяются для управления циклами и индексирования массивов. Хотя на первый взгляд может показаться, что использование типов byte или short эффективнее использования типа int в ситуациях, когда не требуется более широкий допустимый диапазон значений, предоставляемый последним, в действительности это не всегда так. Это обусловлено тем, что при указании значений типа byte и

short в выражениях их тип повышается до int при вычислении выражения. (Повышение типа описано в этой разделе позже.) Поэтому часто тип int наиболее подходит для работы с целочисленными значениями.

long

long – 64-битный тип со знаком, удобный в тех ситуациях, когда длина типа int недостаточна для хранения требуемого значения. Область допустимых значений типа long достаточно велика. Это делает его удобным для работы с большими целыми числами.

Например, ниже приведен пример программы, которая вычисляет количество километров, проходимых лучом света за указанное число дней.

```
// Вычисление расстояния, проходимого светом,
// с применением переменных типа long.
c1ass Light {
 public static void main(String args[]) {
  int 1ightspeed;
  long days;
  long seconds;
  long distance;
// приблизительная скорость света в км за секунду
//1ightspeed
 //300000;
  days = 1000; // указание количества дней
  seconds = days * 24 * 60 * 60; // преобразование в секунды
  distance = 1ightspeed * seconds; // вычисление расстояния
  System.out.prinт("За " + days);
  System.out.print (" дней свет пройдет около ");
  System.out.print1n (distance + "км.");
 }
}
```

Эта программа генерирует следующий вывод:

За 1000 дней свет пройдет около 25920000000000 км. Очевидно,что результат не поместился бы в переменной типа int.

2.1.2. Типы с плавающей точкой

Числа с плавающей точкой, называемые также действительными числами, используются при вычислении выражений, которые требуют получение результата с точностью до определенного десятичного знака. Например, такие вычисления, как вычисление квадратного корня или трансцендентных функций вроде синуса или косинуса[4], приводят к результату, который требует применения типа с плавающей точкой. В Java реализован стандартный (в соответствии с IEEE754) набор типов и операций с плавающей точкой. Существуют два вида типов с плавающей точкой: float и double, которые соответственно, представляют числа одинарной и двойной точности. Их размер и области допустимых значений описаны в табл. 2.2.

Таблица 2.2. Вещественные типы данных.

Название типа	Длина (байты)	Область значений
float	4	3.40282347e+38f ; 1.40239846e-45f
double	8	1.79769313486231570e+308 9406565841246544e-324

float

Тип float определяет значение одинарной точности, которое при хранении занимает 32 бита. В некоторых процессорах обработка значений одинарной точности выполняется быстрее и требует в два раза меньше памяти, чем обработка значений двойной точности, но в тех случаях, когда значения либо очень велики, либо очень малы, точность вычислений оказывается недостаточной. Переменные типа float удобны в тех случаях, когда требуется дробная часть значения без особой точности. Например, значение типа float может быть удобно для представления денежных сумм в рублях и копейках. Ниже приведен пример объявлений переменных типа float:

float hightemp, lowtemp;

double

Двойная точность, как следует из ключевого слова double (двойная), требует использования 64 битов для хранения значений. В действительности в некоторых современных процессорах, которые оптимизированы для выполнения математических вычислений с высокой скоростью, обработка значений двойной точности осуществляется быстрее, чем обработка значений одинарной точности. Все трансцендентные математические функции, такие как sin (), cos () и sqrt (), возвращают значения типа double. Применение типа double наиболее рационально, когда требуется сохранение точности множества

последовательных вычислений или манипулирование большими числами.

Ниже приведен пример короткой программы, в которой переменные типа double используются для вычисления площади Круга.

```
// Вычисление площади круга.
class Area {
 public static void main{String args[])
 double pi, r, a;
 r = 10.8; // радиус окружности
pi = 3.1416; // pi, приблизительное значение
a = pi * r * r; // вычисление площади
System.out.println("Площадь Круга составляет" + a);
 }
}
```

2.1.3. Символы

В Java для хранения символов используется тип данных char. Однако программистам на C/C++ следует помнить, что тип char в Java не эквивалентен типу char в C или C++. В C/C++ char это целочисленный тип, имеющий ширину 8 битов. В Java это не так. Вместо этого в нем для представления символов используется Unicode. Unicode определяет международный набор символов, который может представлять все символы, присутствующие во всех известных языках. Он представляет собой унифицированный набор десятков наборов символов, таких как латиница, греческий алфавит, арабский алфавит, кириллица, иврит, японские и тайские иероглифы и множество других. Поэтому для хранения этих символов требуется 16 битов. Таким образом, в Java тип char является 16 битным. Диапазон допустимых значений этого типа от 0 до 65536. Не существует отрицательных значений типа char. Стандартный набор символов, известный как ASCII, содержит значения от 0 до 127, а расширенный 8-битный набор символов, ISO Latin1 значения от 0 до 255. Поскольку язык Java предназначен для обеспечения возможности создания программ, применимых во всем мире, использование кодировки Unicode для представления символов вполне обосновано[2,3]. Конечно, применение Unicode несколько неэффективно для таких языков, как английский, немецкий, испанский или французский, для представления символов которых вполне достаточно 8 битов. Но это та цена, которую приходится платить за переносимость программ во всемирном

масштабе. Использование переменных типа char демонстрирует следующая программа:

```
// Демонстрация использования типа данных char.
class CharDemo {
 public static void main(String args[]){
 char ch1, ch2;
 chl = 88; // код переменной X
 ch2 = 'У';
 System.out.print("chl и ch2:”);
 System.out.println(chl + " " + ch2);
 }
}
```

Эта программа отображает следующий вывод:

```
chl и ch2: X У
```

Обратите внимание, что переменной chl присвоено значение 88, являющееся значением ASCII (и Unicode), которое соответствует букве x. Как уже было сказано, набор символов ASCII занимает первые 127 значений набора символов Unicode. Поэтому, все "старые трюки", применяемые при работе с символами в других языках, будут работать и в среде Java. Хотя тип char был разработан для хранения символов Unicode, его можно считать также целочисленным типом, пригодным для выполнения арифметических операций.

Например, он позволяет выполнять сложение символов или уменьшать значение символьной переменной. Рассмотрим следующую программу:

```
// Символьные переменные ведут себя подобно целочисленным
//значениям.
class CharDemo2 {
 public static void main(String args[]) {
 char chl;
 ch = ‘X’,
System.out.println ("chl содержит" + chl);
chl++; // увеличение значения chl на единицу
System.out.println ("chl теперь" + chl);
 }
}
```

Эта программа генерирует следующий вывод:

```
chl содержит X
chl теперь Y
```

Вначале присваивается переменной ch1 значение X. Затем она увеличивает значение переменной ch1 на единицу. В результате,

хранящееся в переменной значение становится буквой Y– следующим символом в последовательности ASCII (и Unicode).

2.1.4. Булевские значения

Java содержит элементарный тип, названный boolean, который предназначен для хранения логических значений. Переменные этого типа могут принимать только одно из двух возможных значений: true (истинно) или false (ложно). Этот тип возвращается всеми операциями сравнения, подобными [4] a < b. Тип boolean обязателен для использования также в условных выражениях, которые управляют такими управляющими операторами, как if и for.

Следующая программа служит примером использования типа boolean:

```
// Демострация использования значений типа boolean.
class BoolTest {
 public static void main(String args[]) {
  boolean b;
  b = false;
  System.out.println ("b равна" + b);
  b = true;
  System.out.println ("b равна" + b);
// значение типа boolean может управлять оператором if
  if (b) System.out.println ("Это выполняется.");
  b = false;
  if (b) System.out.println ("Это не выполняется.");
// результат выполнения операции сравнения
//значение типа boolean
System.out.println ("10 > 9 равно" + (10 > 9));
 }
}
```

Эта программа генерирует следующий вывод:

```
b равна false
b равна true
Это выполняется.
10 > 9 равно true
```

В приведенной программе особый интерес представляют три момента. Во-первых, как видно при выводе значения типа boolean методом println () на экране отображается строка "true" или "false". Во-вторых ,самого по себе значения переменной типа boolean достаточно

для управления оператором if. Вовсе не обязательно записывать оператор if так, как показано ниже:

if (b = true) ...

В-третьих, результат выполнения операции сравнения вроде < – значение типа boolean. Именно поэтому выражение 10 > 9 приводит к отображению строки "true".

Более того, выражение 10 > 9 должно быть заключено в дополнительный набор круглых скобок, поскольку операция + обладает более высоким приоритетом, чем операция >.

2.2. Преобразование и приведение типов

Те, кто уже обладает определенным опытом программирования, знают, что достаточно часто программисты присваивают переменной одного типа значение другого[3,5,7]. Если оба эти типа совместимы, Java выполнит преобразование автоматически. Например, всегда можно присвоить значение типа int переменной типа long. Однако не все типы совместимы и, следовательно, не все преобразования типов безоговорочно разрешены. Например, не существует никакого определенного автоматического преобразования double в byte.

К счастью, преобразования между несовместимыми типами выполнять всетаки можно. Для этого необходимо использовать приведение, которое выполняет явное преобразование несовместимых типов. Рассмотрим автоматическое преобразование и приведение типов.

2.2.1. Автоматическое преобраэование типов в Java

При присваивании типа данных переменной другого типа автоматическое преобразования типа выполняется в случае удовлетворения следующих двух условий:

- оба типы совместимы;
- длина целевого типа больше длины исходного типа.

При соблюдении этих двух условий выполняется преобразование с расширением. Например, тип int всегда достаточно велик, чтобы хранить все допустимые значения типа byte, поэтому никакие операторы явного приведения типа не требуются. С точки зрения преобразования с расширением числовые типы, среди которых целочисленный и с плавающей точкой, совместимы друг с другом. Однако не существует автоматических преобразований числовых типов

в char или boolean. Типы char и boolean также не совместимы и между собой.

Как уже говорилось ранее ,Java выполняет автоматическое преобразование типов при сохранении целочисленной константы в переменных типа byte, short, long или char.

2.2.2. Приведение несовместимых типов

Хотя автоматическое преобразование типов удобно, оно не в состоянии удовлетворить все потребности. Например, что делать, если нужно присвоить значение типа int переменной типа byte? Это преобразование не будет выполняться автоматически, поскольку тип byte меньше типа int. Иногда этот вид преобразования называют преобразованием с сужением, поскольку значение явно сужается, чтобы оно могло уместиться в целевом типе. Чтобы выполнить преобразование между двумя несовместимыми типами, необходимо использовать приведение типов. Приведение это всего лишь явное преобразование типов. Общая форма преобразования имеет вид:

(целевой тип) значение

Здесь целевой тип определяет тип, к которому нужно преобразовать указанное значение. Например, следующий фрагмент кода приводит тип int к типу byte. Если значение целочисленного типа больше допустимого диапазона типа byte, оно будет уменьшено до результата деления по модулю (остатка от целочисленного деления) на диапазон типа byte.

```
int a;
byte b;
// ...
b = (byte) a;
```

При присваивании значения с плавающей точкой переменной целочисленного типа выполняется другой вид преобразования типа: усечение. Как вы знаете, целые числа не содержат дробной части. Таким образом, когда значение с плавающей точкой присваивается переменной целочисленного типа, дробная часть отбрасывается. Например, в случае присваивания значения 1,23 целочисленной переменной результирующим значением будет просто 1. Дробная часть 0,23 усекается. Конечно, если размер целочисленной части слишком велик, чтобы уместиться в целевом целочисленном типе, значение будет уменьшено до результата деления по модулю на диапазон целевого типа.

Следующая программа демонстрирует ряд преобразований типа, которые требуют приведения:

```
// Демонстрация приведения типов.
c1ass Conversion {
 public static void main(String args[])
  byte b;
  int i = 257;
  double d = 323.142;
  System.out.priпtln ("\nПреобразование int в byte.");
  b = (byte) i;
  System.out.println ("i и b " + i + " " + b);
  System.out.println ("\nПреобразование double в int.");
 i = (int) d;
  System.out.println ("d и i " + d +" " + i);
  System.out.priпtln ("\nПреобразоаание double в byte.");
  b = (byte) d;
  System.out.print1n ("d и b " + d + " " + b);
 }
}
```

Эта программа генерирует следующий вывод:

```
Преобразование int в byte.
i и b 257 1
Преобразование double в int.
d и i 323.142 323
Преобразование double в byte.
d и b 323.142 67
```

Рассмотрим каждое из этих преобразований. Когда значение 257 приводится к типу byte, результатом будет остаток от деления 257 на 256 (диапазон допустимых значений типа byte), который в данном случае равен 1. Когда значение переменной d преобразуется в тип int, его дробная часть отбрасывается. Когда значение переменной d преобразуется в тип byte, его дробная часть отбрасывается, и значение уменьшается до результата деления по модулю на 256, который в данном случае равен 67.

2.2.3 Автоматическое повышение типа в выражениях

Кроме оцераций присваивания определенное' преобразование типов может выпоняться также в выражениях. Для примера рассмотрим следующую ситуацию. Иногда в выражениях в целях обеспечения необходимой точности промежуточное значение может выходить за

пределы допустимого диапазона любого из операндов. Например, расмотрим следующее выражение:

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

Результат вычисления промежуточного члена a * b вполне может выйти за пределы диапазона допустимых значений его операндов типа byte. Для решения подобных проблем при вычислении выражений Java автоматически повышает тип каждого операнда byte или short до int. То есть вычисление промежуточного выражения a*b выполняется с применением целочисленных значений, а не байтов. Поэтому результат промежуточного выражения 50 * 40, равный 2000, оказывается допустимым, несмотря на то, что и для a и для b задан тип byte.

Хотя автоматическое повышение типа очень удобно, оно может приводить к досадным ошибкам во время компиляции. Например, следующий внешне абсолютно корректный код приводит к возникновению проблемы:

```
byte b = 50;
b = (b * 2); // Ошибка! Значение типа int не может быть присвоено
// переменной типа byte!
```

Код предпринимает попытку повторного сохранения произведения 50 * 2 – недопустимого значения типа byte в переменной типа byte. Однако, поскольку во время вычисления выражения тип операндов был автоматически повышен до int, тип результата также был повышен до int. Таким образом, теперь результат выражения имеет тип int, который не может быть присвоен переменной типа byte без приведения типа.

Сказанное справедливо даже тогда, когда, как в данном конкретном случае, значение, которое должно быть присвоено, умещается в переменной целевого типа.

В тех случаях, когда последствия переполнения понятны, следует использовать явное приведение типов вроде:

```
byte b= 50;
b = (byte) (b * 2);
```

которое приводит к правильному значению, равному 100.

Правила повышения типа

В Java определено несколько правил повышения типа, применяемых к выражениям.

Эти правила следующие [8]: во-первых, тип всех значений byte, short и char повышается до int, как было описано в предыдущем разделе. Во-вторых, если один операнд имеет тип long, тип всего выражения

повышается до long. Если один операнд имеет тип float, тип всего выражения повышается до float. Если любой из операндов имеет тип double, типом результата будет double.

Следующая программа демонстрирует повышение типа значения одного из операндов к типу второго в каждой операции с двумя операндами:

```
c1ass Promote {
public static void main(String args[]) {
 byte b = 42;
 char c = 'a';
 short s = 1024;
 int i = 50000;
 float f = 5.67f;
 double d = .1234;
 double resu1t = (f * b) + (i / c) - (d * s);
 System.out.print1n((f * b) + " + " + (i / c) + "-" + (d * s));
 System.out.print1n("resu1t = " + resu1t);
 }
}
```

Давайте подробнее рассмотрим повышение типа, выполняемое в следующей строке программы:

```
double resu1t = (f * b) + (i / c) - (d * s);
```

В первом промежуточном выражении, f*b, тип переменной b повышается до float, и типом результата вычисления этого промежуточного выражения также является float. В следующем промежуточном выражении i/c тип c повышается до int и результат вычисления этого выражения – in t. Затем в выражении d * s тип значения s повышается до double, и все промежуточное выражение получает тип double. И, наконец, выполняются операции с этими тремя промежуточными значениями типов float, int и double. Результат сложения значений типов float и int имеет тип float. Затем тип значения разности результирующего значения типа float и последнего значения типа double повышается до double, который и становится окончательным типом результата выражения.

2.3. Ссылочные типы

Выражение ссылочного типа имеет значение либо null, либо ссылку, указывающую на некоторый объект в виртуальной памяти JVM.

2.3.1. Объекты и правила работы с ними

Объект (object) – это экземпляр некоторого класса, или экземпляр массива. Массивы будут подробно рассматриваться в соответствующем разделе. Класс – это описание объектов одинаковой структуры, и если в программе такой класс используется, то описание присутствует в единственном экземпляре[2-8]. Объектов этого класса может не быть вовсе, а может быть создано сколь угодно много.

Объекты всегда создаются с использованием ключевого слова new, причем одно слово new порождает строго один объект (или вовсе ни одного, если происходит ошибка). После ключевого слова указывается имя класса, от которого мы собираемся породить объект. Создание объекта всегда происходит через вызов одного из конструкторов класса (их может быть несколько), поэтому в заключение ставятся скобки, в которых перечислены значения аргументов, передаваемых выбранному конструктору. В приведенных выше примерах, когда создавались объекты типа Point, выражение new Point (3,5) означало обращение к конструктору класса Point, у которого есть два аргумента типа int. Кстати, обязательное объявление такого конструктора в упрощенном объявлении класса отсутствовало. Объявление классов рассматривается в следующих разделах, однако приведем правильное определение Point:

```
class Point {
  int x, y;

 /**
 * Конструктор принимает 2 аргумента,
 * которыми инициализирует поля объекта.
 */
  Point (int newx, int newy) {
  x=newx;
  y=newy;
  }
}
```

Если конструктор отработал успешно, то выражение new возвращает ссылку на созданный объект. Как только обнаруживается, что ссылок больше нет, такой объект предназначается для уничтожения сборщиком мусора (garbage collector). Восстановить ссылку на такой "потерянный" объект невозможно.

```
Point p=new Point(1,2);
 // Создали объект, получили на него ссылку
```

```
Point p1=p;
 // теперь есть 2 ссылки на точку (1,2)
p=new Point(3,4);
 // осталась одна ссылка на точку (1,2)
p1=null;
```

Ссылок на объект-точку (1,2) больше нет, доступ к нему утерян, и он вскоре будет уничтожен сборщиком мусора.

Любой объект порождается только с применением ключевого слова new. Единственное исключение – экземпляры класса String. Записывая любой строковый литерал, мы автоматически порождаем объект этого класса. Оператор конкатенации +, результатом которого является строка, также неявно порождает объекты без использования ключевого слова new.

Рассмотрим пример:

```
"abc"+"def"
```

При выполнении этого выражения будет создано три объекта класса String. Два объекта порождаются строковыми литералами, третий будет представлять результат конкатенации.

Над ссылочными значениями можно производить следующие операции:

- обращение к полям и методам объекта
- оператор instanceof (возвращает булевское значение)
- операции сравнения == и != (возвращают булевское значение)
- оператор приведения типов
- оператор с условием ? :
- оператор конкатенации со строкой +

Обращение к полям и методам объекта можно назвать основной операцией над ссылочными величинами. Осуществляется она с помощью символа “.” (точка). Примеры ее применения будут приводиться.

Используя оператор instanceof, можно узнать, от какого класса произошел объект. Этот оператор имеет два аргумента. Слева указывается ссылка на объект, а справа – имя типа, на совместимость с которым проверяется объект.

2.3.2. Класс Object

В Java множественное наследование отсутствует (оно реализуется через интерфейсы). Каждый класс может иметь только одного родителя. Таким образом, мы можем проследить цепочку наследования от любого класса, поднимаясь все выше. Существует класс, на котором такая цепочка всегда заканчивается, это класс Object. Именно от него наследуются все классы, в объявлении которых явно не указан другой родительский класс. А значит, любой класс напрямую, или через своих родителей, является наследником Object. Отсюда следует, что методы этого класса есть у любого объекта (поля в Object отсутствуют), а потому они представляют особенный интерес.

Рассмотрим основные из них.

getClass()

Этот метод возвращает объект класса Class, который описывает класс, от которого был порожден этот объект. Класс Class будет рассмотрен ниже. У него есть метод getName(), возвращающий имя класса:

```
String s = "abc";
Class cl=s.getClass();
System.out.println (cl.getName());
```

Результатом будет строка:

```
Java .lang.String
```

В отличие от оператора instanceof, метод getClass() всегда возвращает точно тот класс, от которого был порожден объект.

equals()

Этот метод имеет один аргумент типа Object и возвращает boolean. Как уже говорилось, equals() служит для сравнения объектов по значению, а не по ссылке. Сравнивается состояние объекта, у которого вызывается этот метод, с передаваемым аргументом.

```
Point p1=new Point (2,3);
Point p2=new Point (2,3);
System.out.println (p1.equals(p2));
```

Результатом будет true.

Поскольку сам Object не имеет полей, а значит, и состояния, в этом классе метод equals возвращает результат сравнения по ссылке. Однако при написании нового класса можно переопределить этот метод и описать правильный алгоритм сравнения по значению (что и сделано в большинстве стандартных классов).

hashCode()

Данный метод возвращает значение int. Цель hashCode() – представить любой объект уникальным целым числом. В классе Object этот метод реализован на уровне JVM. Сама виртуальная машина генерирует число хеш-кодов, основываясь на расположении объекта в памяти.

toString()

Этот метод позволяет получить текстовое описание любого объекта. Создавая новый класс, данный метод можно переопределить и возвращать более подробное описание. Для класса Object и его наследников, не переопределивших toString(), метод возвращает следующее выражение:

getClass().getName()+"@"+hashCode()

Метод getName() класса Class уже приводился в пример, а хэш-код еще дополнительно обрабатывается специальной функцией для представления в шестнадцатеричном формате.

Например:

System.out.println (new Object());

Результатом будет:

Java .lang.Object@92d342

В результате этот метод позволяет по текстовому описанию понять, от какого класса был порожден объект и, благодаря хеш-коду, различать разные объекты, созданные от одного класса.

Именно этот метод вызывается при конвертации объекта в текст, когда он передается в качестве аргумента оператору конкатенации строк.

finalize()

Данный метод вызывается при уничтожении объекта автоматическим сборщиком мусора (garbage collector). В классе Object он ничего не делает, однако в классе-наследнике позволяет описать все действия, необходимые для корректного удаления объекта, такие как закрытие соединений с БД, сетевых соединений, снятие блокировок на файлы и т.д. В обычном режиме напрямую этот метод вызывать не нужно, он отработает автоматически. Если необходимо, можно обратиться к нему явным образом.

2.3.3. Класс String

Как уже указывалось, класс String занимает в Java особое положение. Экземпляры только этого класса можно создавать без использования ключевого слова new. Каждый строковый литерал

порождает экземпляр String, и это единственный литерал (кроме null), имеющий объектный тип.

Затем значение любого типа может быть приведено к строке с помощью оператора конкатенации строк, который был рассмотрен для каждого типа, как примитивного, так и объектного.

Еще одним важным свойством данного класса является неизменяемость. Это означает, что, породив объект, содержащий некое значение-строку, мы уже не можем изменить данное значение – необходимо создать новый объект.

```
String s="a";
s="b";
```

Во второй строке переменная сменила свое значение, но только создав новый объект класса String.

Поскольку каждый строковый литерал порождает новый объект, что есть очень ресурсоемкая операция в Java , зачастую компилятор стремится оптимизировать эту работу.

Во-первых, если используется несколько литералов с одинаковым значением, для них будет создан один и тот же объект.

```
String s1 = "abc";
String s2 = "abc";
String s3 = "a"+"bc";
System.out.println (s1==s2);
System.out.println (s1==s3);
```

Результатом будет:

```
true
true
```

2.3.4 Класс Class

Наконец, последний класс, который будет рассмотрен в этом разделе.

Класс Class[3,4] является метаклассом для всех классов Java . Когда JVM загружает файл .class, который описывает некоторый тип, в памяти создается объект класса Class, который будет хранить это описание.

Например, если в программе есть строка

```
Point p=new Point(1,2);
```

то это означает, что в системе созданы следующие объекты:

1. объект типа Point, описывающий точку (1,2);
2. объект класса Class, описывающий класс Point;

3. объект класса Class, описывающий класс Object. Поскольку класс Point наследуется от Object, его описание также необходимо;
4. объект класса Class, описывающий класс Class. Это обычный Java -класс, который должен быть загружен по общим правилам.

Одно из применений класса Class уже было рассмотрено – использование метода getClass() класса Object. Если продолжить последний пример с точкой:

```
Class cl=p.getClass();
 // это объект №2 из списка
Class cl2=cl.getClass();
 // это объект №4 из списка
Class cl3=cl2.getClass();
 // опять объект №4
```

Выражение cl2==cl3 верно.

3.ПРОГРАММИРОВАНИЕ ВЫРАЖЕНИЙ

Из констант и переменных, операций над ними, вызовов методов и скобок составляются *выражения* (expressions). Разумеется, все элементы выражения должны быть совместимы, нельзя написать, например, 2 + true. При вычислении выражения выполняются четыре правила:

1. Операции одного приоритета вычисляются слева направо: x + y + z вычисляется как (x + y) + z. Исключение: операции присваивания вычисляются справа налево: x = y = z вычисляется как x = (y = z).

2. Левый операнд вычисляется раньше правого.

3. Операнды полностью вычисляются перед выполнением операции.

4. Перед выполнением составной операции присваивания значение левой части сохраняется для использования в правой части.

Следующие примеры показывает особенности применения первых трех правил. Пусть

int a = 3, b = 5;

Тогда результатом выражения b + (b = 3) будет число 8; но результатом выражения (b = 3) + b будет число 6. Выражение b += (b = 3) даст в результате 8, потому что вычисляется как первое из приведенных выше выражений.

Приоритет операций

Операции перечислены в порядке убывания приоритета. Операции на одной строке имеют одинаковый приоритет[4,7].

1. Постфиксные операции ++ и —.
2. Префиксные операции ++ и —, дополнение ~ и отрицание !.
3. Приведение типа (тип).
4. Умножение *, деление / и взятие остатка %.
5. Сложение + и вычитание -.
6. Сдвиги <<, >>, >>>.
7. Сравнения >, <, >=, <=.
8. Сравнения ==, !=.
9. Побитовая конъюнкция &.
10. Побитовое исключающее ИЛИ ^.
11. Побитовая дизъюнкция | .
12. Конъюнкция &&.
13. Дизъюнкция | | .
14. Условная операция ?: .
15. Присваивания =, +=, -=, *=, /=, %=, &=, ^=, |=, <<, >>, >>>.

Здесь перечислены не все операции языка Java , список будет дополняться по мере изучения новых операций.

3.1.Класс Math

При кодировании выражений используются всевозможные математические функции, которые находятся в статическом классе встроенного пакета Java .lang.
Класс Math состоит из набора статических методов, производящих наиболее популярные математические вычисления, и двух констант, имеющих особое значение в математике, – это число Пи и основание натурального логарифма. Часто этот класс еще называют классом-утилитой (Utility class). Так как все методы класса статические, нет необходимости создавать экземпляр данного класса, потому он и не имеет открытого конструктора. Нельзя также и наследоваться от этого класса, так как он объявлен с модификатором final. Итак, константы определены следующим образом:

public static final double Matht.PI – задает число π ("пи");

public static final double Matht.E – основание натурального логарифма.

В таблице 1 приведены все методы класса и дано их краткое описание.

Таблица 3.1Методы класса Math и их краткое описание.		
Возвращаемое значение	**Имя метода и параметры**	**Описание**
…	abs(… a)	абсолютное значение (модуль) для типов double, float, int, long
double	acos(double a)	арккосинус
double	asin(double a)	арксинус
double	atan(double a)	арктангенс
double	ceil(double a)	наименьшее целое число, большее a
double	floor(double a)	целое число, меньшее a
double	IEEEremainder (double a, double b)	остаток по стандарту IEEE 754
double	sin(double a)	синус (здесь и далее: аргумент должен быть в радианах)
double	cos(double a)	косинус
double	tan(double a)	тангенс
double	exp(double a)	e в степени a
double	log(double a)	натуральный логарифм a
…	max(… a, … b)	большее из двух чисел (для типов double, float, long, int)
…	min(… a, … b)	меньшее из двух чисел (для типов double, float, long, int)
double	pow(double a, double b)	a в степени b
double	random()	случайное число от 0.0 до 1.0
double	rint(double a)	значение int, ближайшее к a
…	round(… a)	значение long для double (int для float), ближайшее к a
double	sqrt(double a)	квадратный корень числа a
double	toDegrees(double a)	преобразование из радианов в градусы
double	toRadians(double a)	преобразование из градусов в радианы

-

3.2.Класс 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.

4.БАЗОВЫЕ ИНСТРУКЦИИ

Любой алгоритм, предназначенный для выполнения на компьютере, можно разработать, используя только линейные вычисления, разветвления и циклы. Записать его можно в разных формах: в виде блок-схемы, на псевдокоде, на обычном языке, как мы записываем кулинарные рецепты, или как-нибудь еще "алгоритмы". Любой язык программирования должен иметь средства записи алгоритмов. Они называются инструкциями или операторами (statements) языка. Минимальный набор инструкций должен содержать инструкции для записи линейных вычислений, условный оператор для записи разветвлении и оператор цикла.

Обычно состав инструкций языка программирования шире: для удобства записи алгоритмов в язык включаются несколько операторов цикла, оператор варианта, операторы перехода, операторы описания объектов.

Набор инструкций языка Java включает [4,7]:

- операторы описания переменных и других объектов (они были рассмотрены выше);
- операторы-выражения;
- операторы присваивания;
- условный оператор if;
- три оператора цикла while, do-while, for;
- оператор варианта switch;
- операторы перехода break, continue и return;
- блок {};

- пустой оператор – просто точка с запятой.

Последовательность выполнения инструкций может быть непрерывной, а может и прерываться (при возникновении определенных условий). Выполнение инструкций может быть прервано, если в потоке вычислений будут обнаружены инструкции break, continue, return.

4.1. Блоки и локальные переменные

Блок – это последовательность инструкций, объявлений локальных классов или локальных переменных, заключенных в скобки. Область видимости локальных переменных и классов ограничена блоком, в котором они определены.

Инструкции в блоке выполняются слева направо, сверху вниз. Если все операторы (выражения) в блоке выполняются нормально, то и весь блок выполняется нормально. Если какой-либо оператор (выражение) завершается ненормально, то и весь блок завершается ненормально.

Нельзя объявлять несколько локальных переменных с одинаковыми именами в пределах видимости блока. Приведенный ниже код вызовет ошибку времени компиляции.

```
public class Test {
 public Test() { }
 public static void main(String[] args) {
  Test t = new Test();
  int x;
  lbl: { int x = 0; System.out.println("x = " + x); }
 }
}
```

В то же время не следует забывать, что локальные переменные перекрывают видимость переменных-членов. Так, следующий пример отработает нормально.

```
public class Test {
 static int x = 5;
 public Test() { }
 public static void main(String[] args) {
  Test t = new Test();
  int x = 1; System.out.println("x = " + x);
 }
}
```

На консоль будет выведено x = 1.

4.2. Пустой оператор

Точка с запятой (;) является пустым оператором. Данная конструкция вполне применима там, где не предполагается выполнение никаких действий. Преждевременное завершение пустого оператора невозможно.

4.3. Метки

Любая инструкция, или блок, может иметь метку. Метка это идентификатор с двоеточием. Метку можно указывать в качестве параметра для инструкций break и continue. Область видимости метки ограничивается инструкцией, или блоком, к которому она относится.

В случае, если имеется несколько вложенных блоков и операторов, допускается обращение из внутренних блоков к меткам, относящимся к внешним.

Этот пример является вполне корректным:

```
public class Test {
 static int x = 5;
 static {
 }
 public Test() {
 }
 public static void main(String[] args) {
  Test t = new Test();
  int L2 = 0;
  Test: for(int i = 0; i< 10;i++) {
  test: for(int j = 0; j< 10;j++) {
    if( i*j > 50) break Test;
  }
  }
 }
 private void test() {
 ;
 }
}
```

В этом же примере можно увидеть, что метки используют пространство имен, отличное от пространства имен переменных, методов и классов.

Традиционно использование меток не рекомендуется, особенно в объектно-ориентированных языках, поскольку серьезно усложняет понимание порядка выполнения кода, а значит, и его тестирование и отладку. Для Java этот запрет можно считать не столь строгим, поскольку самый опасный оператор goto отсутствует. В некоторых ситуациях (как в рассмотренном примере с вложенными циклами) использование меток вполне оправданно, но, конечно, их применение следует ограничивать лишь самыми необходимыми случаями.

4.4. Условный оператор

Условный оператор (if-then-else statement) в языке Java записывается так:

if (логВыр) оператор1 else оператор2

и действует следующим образом. Сначала вычисляется логическое выражение *логвыр.* Если результат true, то выполняется *оператор1* и на этом действие условного оператора завершается, *оператор2* не работает, далее будет выполняться следующий за if оператор. Если результат false, то выполняется *оператор2,* при этом оператор1 вообще не выполняется.

Условный оператор может быть сокращенным (if-then statement):

if (логВыр) оператор1

и в случае false не выполняется ничего.

Синтаксис языка не позволяет записывать несколько инструкций ни в ветви then, ни в ветви else. При необходимости составляется блок инструкций в фигурных скобках. Соглашения "Code Conventions" рекомендуют всегда использовать фигурные скобки и размещать оператор на нескольких строках с отступами, как в следующем примере:

```
if (a < x) {
x = a + b; } else {
x = a - b;
}
```

Это облегчает добавление инструкций в каждую ветвь при изменении алгоритма.

Очень часто одним из операторов является снова условный оператор, например:

```
if (n == 0}{
sign = 0;
} else if (n < 0){
sign = -1;
```

```
} else {
sign = 1;
}
```
При этом может возникнуть такая ситуация ("dangling else"):
```
int ind = 5, x = 100;
if (ind >= 10) if (ind <= 20) x = 0; else x = 1;
```
Сохранит переменная x значение 100 или станет равной 1? Здесь необходимо волевое решение, и общее для большинства языков, в. том числе и Java ,. правило таково: ветвь else относится к ближайшему слева условию if, не имеющему своей ветви else. Поэтому в нашем примере переменная x останется равной 0.
Изменить этот порядок можно с помощью блока:
```
if (ind > 10) {if (ind < 20) x = 0; else x = 1;}
```
Вообще не стоит увлекаться сложными вложенными условными операторами. Проверки условий занимают много времени. По возможности лучше использовать логические операции, например, в нашем примере можно написать
```
if (ind >= 10 && ind <= 20) x = 0; else x = 1;
```

4.5. Оператор switch

Переключатель удобно использовать в случае необходимости множественного выбора. Выбор осуществляется на основе целочисленного значения.
Структура оператора:
```
 switch(int value) {
  case const 1:
 выражение или блок
  case const 2:
 выражение или блок
case const n:
 выражение или блок
default:
 выражение или блок
 }
```

Причем, фраза default не является обязательной.

В качестве параметра switch может использоваться переменная типа byte, short, int, char или выражение. Выражение должно в конечном итоге возвращать параметр одного из указанных ранее типов. В

инструкции case не могут применяться значения примитивного типа long и ссылочных типов Long, String, Integer, Byte и т.д.

При выполнении инструкции switch производится последовательное сравнение значения x с константами, указанными после case, и в случае совпадения выполняется выражение следующего за этим условием. Если выражение выполнено нормально и нет преждевременного его завершения, то производится сравнение для последующих case. Если же выражение, следующее за case, завершилось ненормально, то будет прекращено выполнение всего оператора switch.

Если не выполнен ни одна инструкция case, то выполнится оператор default, если он имеется в данном switch. Если оператора default нет и ни одно из условий case не выполнено, то ни одно из выражений switch также выполнено не будет.

Следует обратить внимание, что, в отличие от многозвенного if-else, если какое-либо условие case выполнено, то выполнение switch не прекратится, а будут проверяться следующие за ним условия. Если этого необходимо избежать, то после кода следующего за оператором case используется инструкция break, прерывающая дальнейшее выполнение инструкции switch.

После оператора case должен следовать литерал, который может быть интерпретирован как 32-битовое целое значение. Здесь не могут применяться выражения и переменные, если они не являются final static.

Рассмотрим пример:

```
int x = 2;
switch(x) {
case 1:
case 2:
System.out.println("Равно 1 или 2");
break;
case 3:
case 4:
System.out.println("Равно 3 или 4");
break;
default:
System.out.println(
"Значение не определено");
}
```

В данном случае на консоль будет выведен результат "Равно 1 или 2". Если же убрать операторы break, то будут выведены все три строки.Вот такая конструкция вызовет ошибку времени компиляции.

```
int x = 5;
switch(x) {
case y: // только константы!
:
break;
}
```

В инструкции switch не может быть двух case с одинаковыми значениями.

Т.е. следующая конструкция недопустима.

```
switch(x) {
case 1:
System.out.println("One");
break;
case 1:
System.out.println("Two");
break;
case 3:
System.out.println("Tree or other value");
}
```

Также в конструкции switch может быть применен только один оператор default.

4.6. Управление циклами

В программах часто приходится повторять некоторые действия. Та часть программы, которая повторяет инструкцию или группу инструкций, называется циклом (loop). Инструкции, подлежащие повторению в цикле, называются телом (body) цикла. Каждое повторение тела цикла называется итерацией (iteration) цикла.

При разработке цикла необходимо определить, какое действие будет выполняться в теле цикла, а также механизм для принятия решения о том, когда должно прекратиться повторение тела цикла.

В языке Java имеется три основных конструкции управления циклами:

- цикл while;
- цикл do;
- цикл for.

4.6.1. Цикл while

Основная форма цикла while может быть представлена так:

while(логическое выражение) <повторяющееся выражение, или блок>;

В данной языковой конструкции повторяющееся выражение, или блок будет исполняться до тех пор, пока логическое выражение будет иметь истинное значение. Этот многократно исполняемый блок называют телом цикла. Предположим следующий контекст оператора цикла

<оператор1>;
while (логич.выражение)
<оператор2>;
<оператор3>;

тогда блок-схема оператора можно изобразить следующим образом:

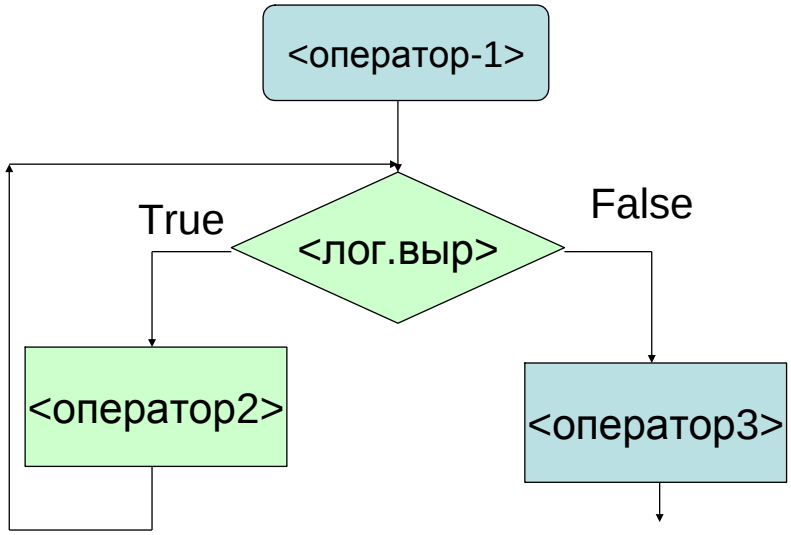

Рис.4.6.1.

Операторы continue и break могут изменять нормальное исполнение тела цикла. Так, если в теле цикла встретился инструкция continue, то операторы, следующие за ним, будут пропущены и выполнение цикла начнется сначала. Если continue используется с меткой и метка принадлежит к данному while, то выполнение его будет аналогичным. Если метка не относится к данному while, его выполнение будет прекращено и управление будет передано на оператор, или блок, к которому относится метка.

Если встретится инструкция break, то выполнение цикла будет прекращено.

Если выполнение блока было прекращено по какой-то другой причине (возникла исключительная ситуация), то выполнение всего цикла будет прекращено по той же причине.

Рассмотрим несколько примеров:

```
public class Test {
  static int x = 5;
  public Test() {
  }
  public static void main(String[] args) {
    Test t = new Test();
    int x = 0;
    while(x < 5) {
       x++;
       if(x % 2 == 0) continue;
       System.out.print (" " + x);
    }
  }
}
```

На консоль будет выведено

```
1 3 5
```

т.е. вывод на печать всех четных чисел будет пропущен.

Типичный вариант использования выражения while():

```
int i = 0;
while( i++ < 5) {
System.out.println("Counter is " + i);
}
```

Следует помнить, что цикл while() будет выполнен только в том случае, если на момент начала его выполнения логическое выражение будет истинным. Таким образом, при выполнении программы может иметь место ситуация, когда цикл while() не будет выполнен ни разу.

```
boolean b = false;
while(b) {
System.out.println("Executed");
}
```

В данном случае строка System.out.println("Executed"); выполнена не будет.

4.6.2. Цикл do

Основная форма цикла do имеет следующий вид:

do

<повторяющееся выражение или блок>;

while(логическое выражение)

Цикл do будет выполняться до тех пор, пока логическое выражение будет истинным. В отличие от цикла while, этот цикл будет выполнен, как минимум, один раз.

Пусть контекст оператора do будет следующим:

do

<оператор-1>;

<оператор-2>;

…

<оператор-n>;

while (логическое выражение);

На рис.2 изображена блок-схема выполнения этого оператора.

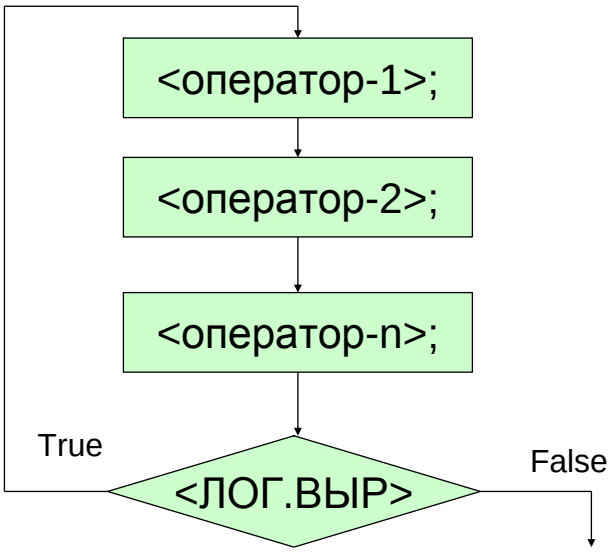

Рис.4.6.2.

Типичная конструкция цикла do:

```
int counter = 0;
do {
counter ++;
System.out.println("Counter is "
+ counter);
} while(counter > 5);
```

В остальном выполнение цикла do аналогично выполнению цикла while, включая использование инструкций break и continue.

4.6.3. Цикл for

Довольно часто бывает необходимо изменять значение какой-либо переменной в заданном диапазоне и выполнять повторяющуюся последовательность инструкций с использованием этой переменной. Для выполнения такой последовательности действий как нельзя лучше подходит конструкция цикла for.

Основная форма цикла for выглядит следующим образом:

for(выражение инициализации; условие; выражение обновления)
 <повторяющееся выражение или блок>;

Блок-схема этого цикла приведена на рис.3.

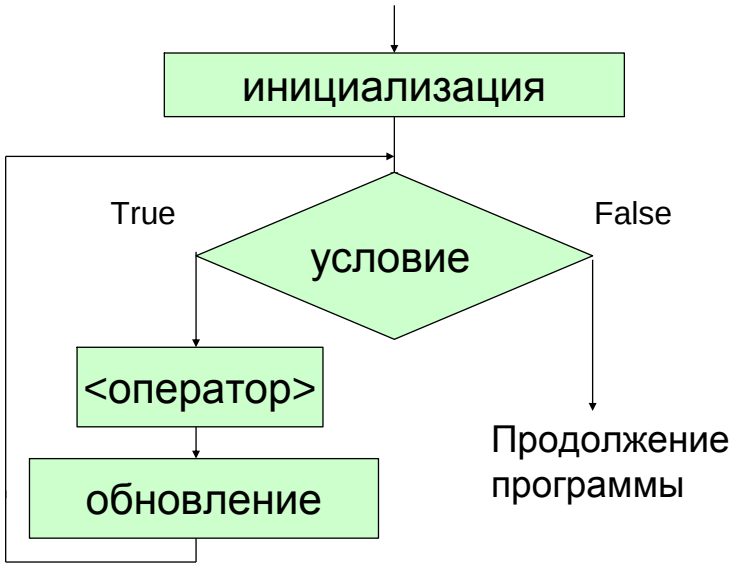

Рис.4.6.3.

Ключевыми элементами данной языковой конструкции являются предложения, заключенные в круглые скобки и разделенные точкой с запятой.

Выражение инициализации выполняется до начала выполнения тела цикла. Чаще всего используется как некое стартовое условие (инициализация, или объявление переменной).

Условие должно быть логическим выражением и трактуется точно так же, как логическое выражение в цикле while(). Тело цикла выполняется до тех пор, пока логическое выражение истинно. Как и в

случае с циклом while(), тело цикла может не исполниться ни разу. Это происходит, если логическое выражение принимает значение "ложь" до начала выполнения цикла.

Выражение обновления выполняется сразу после исполнения тела цикла и до того, как проверено условие продолжения выполнения цикла. Обычно здесь используется выражение инкрементации, но может быть применено и любое другое выражение.

Пример использования цикла for():

```
...
for(counter=0;counter<10;counter++) {
 System.out.println ("Counter is " + counter);
}
```

В данном примере предполагается, что переменная counter была объявлена ранее. Цикл будет выполнен 10 раз и будут напечатаны значения счетчика от 0 до 9. Разрешается определять переменную прямо в предложении:

```
for (int cnt = 0;cnt < 10; cnt++) {
 System.out.println("Counter is " + cnt);
}
```

Результат выполнения этой конструкции будет аналогичен предыдущему. Однако нужно обратить внимание, что область видимости переменной cnt будет ограничена телом цикла.

Любая часть конструкции for() может быть опущена. В вырожденном случае мы получим инструкцию for с пустыми значениями

```
for(;;) {...
}
```

В данном случае цикл будет выполняться бесконечно. Эта конструкция аналогична конструкции while(true){}. Условия, в которых она может быть применена, мы рассмотрим позже.

Возможно также расширенное использование синтаксиса инструкции for(). Предложение и выражение могут состоять из нескольких частей, разделенных запятыми

```
for(i = 0, j = 0; i<5;i++, j+=2) {
...
}
```

Использование такой конструкции вполне правомерно.

Возможны случаи, когда внутри тела цикла необходимо повторять некоторую последовательность инструкций, т. е. организовать внутренний цикл. Такая структура получила название цикла в цикле или вложенных циклов. Глубина вложения циклов (то есть количество

вложенных друг в друга циклов) может быть, в принципе, неограниченной. При использовании такой структуры для экономии машинного времени необходимо выносить из внутреннего цикла во внешний все инструкции, которые не зависят от параметра внутреннего цикла.

Естественно, области действия вложенных циклов не должны пересекаться, по крайней мере, все циклы во вложенной конструкции могут завершаться в одной точке. Любой из циклов может быть завершен досрочно (например, по оператору break).

```
while...{
....
 for ...{

 do ...{
 .... break;
 }
  }
 }
```

Общее правило выполнения вложенных циклов можно сформулировать следующим образом: для каждой итерации внешнего цикла полностью прорабатывает внутренний цикл.

В циклах довольно часто используют инструкции continue и break.

4.6.4. Оператор continue

Оператор continue может использоваться только в циклах while, do, for. Если в потоке вычислений встречается инструкция continue, то выполнение текущей последовательности операторов (выражений) должно быть прекращено и управление будет передано на начало блока, содержащего этот оператор. В очередном фрагменте кода инструкция continue позволяет обойти деление на нуль:

```
for (int i = 0; i < N; i++){
 if (i == j) continue;
 s += 1.0 / (i - j);
}
```

4.6.5. Оператор break

Инструкция break используется в операторах цикла и операторе switch для немедленного выхода из этих конструкций. Оператор break <метка> применяется внутри помеченных

операторов цикла, оператора варианта или помеченного блока для немедленного выхода за эти операторы. Следующая схема поясняет эту конструкцию.

```
ml: { // Внешний блок
m2: { // Вложенный блок – второй уровень
m3: { // Третий уровень вложенности...
if (что-то случилось) break m2;
// Если true, то здесь ничего не выполняется
} //конец блока m3
// Здесь тоже ничего не выполняется
} //конец блока m2
// Сюда передается управление
}
```

Поначалу сбивает с толку то обстоятельство, что метка ставится перед блоком или оператором, а управление передается за этот блок или оператор. Поэтому не стоит увлекаться оператором break с меткой.

5. КЛАССЫ И ОБЪЕКТЫ

Со времени формулирования этой концепции в 1974 году (Liskov B., Zilles S. Programming with abstract data types) абстрактные типы данных играют важную роль в теории программирования. Концепция АТД является, наряду с объектно-ориентированным подходом, наиболее популярной в настоящее время методологией для создания программ. В процессе декомпозиции программы на составляющие компоненты доступ к ним организуется посредством так называемого, кластера операций, который представляет собой конечный список операций, которые могут быть использованы для модификации данных, предоставляемых данным компонентом.

Отличительной особенностью АТД как механизма абстракции является тот факт, что функциональность компонента программы, описываемая кластером операций, может быть реализована различными способами. Различные реализации абстрактных типов данных взаимозаменяемы благодаря механизму абстракции АТД, позволяющему скрыть детали реализации с помощью набора предопределенных операций.

Концепция абстрактных типов данных хорошо описывается с помощью математической теории алгебраических систем. Алгебраическую систему или, проще говоря, алгебру (абстрактную алгебру), неформально можно определить как множество с набором

операций, действующих на элементах данного множества. Операции реализуются как функции от одного или более параметров, действующие на элементах данного множества (для операции с одним аргументом) или на декартовых произведениях множества (для операций с несколькими аргументами). Описание операций, включающее в себя описание типов аргументов и возвращаемых значений, называется сигнатурой алгебраической системы. Сигнатуры, очевидно, представляют математическую модель абстрактного типа данных. Это обстоятельство дает возможность описывать программные сущности, заданные посредством АТД, как алгебраические системы.

Типы данных впервые были описаны Д. Кнутом в его книге «Искусство программирования» [1]. В главе 2, «Информационные структуры», Кнут описывает так называемые структуры данных, определяемые как способы организации данных внутри программы. Кнут описывает такие типы данных, как списки, деревья, стеки, очереди, деки и т.д. Рассмотрим, например, как Кнут описывает тип данных «стек». Кроме собственно описания самой структуры данных, Кнут описывает «алгоритмы обработки» этой структуры с помощью словаря специальных терминов [1, стр. 281]. Для стека этот словарь содержит термины: push (втолкнуть), pop (вытолкнуть) и top (верхний элемент стека).

Таким образом, типы данных описываются Кнутом с помощью специального языка, задающего определенную терминологию и толкование этой терминологии. Эта особенность описания была замечена Стивеном Жиллем и, таким образом, явилась одним из побудительных мотивов для осознания важности концепции АТД.

В 1972 году была напечатана работа Дэвида Парнаса (David Parnas), в которой впервые был сформулирован принцип разделения программы на модули. Модули – это компоненты программы, которые имеют два важных свойства:

- модули скрывают в себе детали реализации;
- модули могут быть повторно использованы в различных частях программы.

Парнас представлял модули как абстрактные машины, хранящие внутри себя состояния и позволяющие изменять это состояние посредством определенного набора операций. Эта концепция является базовой, как для концепции абстрактных типов данных, так и для объектно-ориентированного программирования.

Понятие абстрактного типа данных впервые в явном виде было сформулировано в совместной работе Стивена Жиля и Барбары Лисков (1974). В разделе «Смысл понятия абстракции» авторы обсуждают,

каким образом понятие абстракции может быть применимо к программному коду. Абстракция – это способ отвлечься от неважных деталей и, таким образом, выбрать наиболее важные признаки для рассмотрения. В процессе создания программы разработчик строит программную модель решаемой задачи. В процессе построения программной модели разработчик оперирует элементами этой модели. Программный код структурируется соответствующим образом. Для выделения программных сущностей в коде программы естественно использовать механизм абстракции. В работе Жиля и Лисков рассматривался механизм т.н. поведенческой абстракции или, в терминологии авторов, функциональной абстракции.

Функциональная абстракция подразумевает выделение набора операций (функций) для работы с элементами программной модели. Таким образом, сущности программной модели представляются с помощью набора операций. Так осуществляется поведенческая абстракция сущности в программном коде. Сами авторы использовали термин «operational cluster», т.е. набор операций, и назвали такой набор операций абстрактным типом данных.

Что такое объектно-ориентированное программирование? Если абстрагироваться от программирования, и не вдаваться глубоко в философские концепции понятия объекта, то можно просто рассмотреть объекты на реальных примерах: вы сами, ваш телевизор, ваша шариковая ручка, ваш телефон, ваш бумажник и т.д. – все это примеры реальных объектов. У каждого такого реального объекта есть два рода характеристик: состояние (или набор свойств, state) и поведение (behaviour)[8]. Например, ваша шариковая ручка может иметь следующий набор свойств – название (изготовитель), цвет, модель, размер, тип стержня (хотя стержень можно рассматривать как отдельный объект), толщина рисуемой линии, цена, дата изготовления, объем чернил, автоматическая или обыкновенная и т.д. Поведение шариковой ручки можно ограничить парой действий – писать, и, в случае автоматической ручки, открываться и закрываться. Как можно заметить, для описания свойств используются существительные и прилагательные, а для поведения – глаголы.

Программные объекты (software objects) моделируются также как и реальные, и могут иметь свойства и поведение. Состояние (набор свойств) объекта хранится в переменных (variables) объекта (их часто называют полями объекта), а поведение (варианты поведения) определяется через методы (methods, функции) объекта.

Наряду с понятием объекта в объектно-ориентированном программировании концептуальным является понятие класса.

Рассмотрим это понятие на примере той же шариковой ручки. Можно сказать, что наша конкретная ручка принадлежит классу шариковых ручек, и класс, таким образом, объединяет в себя все множество шариковых ручек. И если мы говорим, что что-то принадлежит классу шариковых ручек, то это «что-то» является шариковой ручкой. А что значит - являться шариковой ручкой? Это значит иметь что-то общее (или похожее) со всеми шариковыми ручками – главное, способность писать, ну, и, например, иметь какой-то размер и цвет. И наша конкретная ручка является конкретным представителем класса шариковых ручек, имея конкретные размеры и цвет. Таким образом, объект (наша ручка) является конкретным представителем своего класса (шариковых ручек). У класса могут быть подклассы – например, подкласс автоматических шариковых ручек, а у каждого подкласса - свои подклассы, например – подкласс автоматических шариковых ручек с кнопкой-переключателем на конце ручки и подкласс автоматических шариковых ручек с фонариком – такая структура классов называется иерархической. В нашем случае наверху иерархии находится класс шариковых ручек. Можно ввести более общий класс, например, класс пишущих ручек, для которого класс шариковых ручек является всего лишь подклассом, а в качестве другого подкласса можно принять класс перьевых пишущих ручек. Если объект является представителем какого-то класса в нижней части иерархии, то, естественно, он является и представителем класса, который стоит выше на этой же ветви дерева иерархии.

В программировании понятия объекта и класса весьма схожи с представленными выше примерами. Каждый объект является конкретным представителем (instance) своего класса, и каждый объект должен обладать тем набором свойств (полей) и уметь выполнять действия (иметь те методы), которые предписаны представителям его класса. Объекты одного класса могут отличаться друг от друга только значениями своих полей (правда, эти различия могут привести и к различному поведению объектов, но это уже следствие).

Программные объекты используются в качестве моделей реальных объектов, причем реальные объекты могут быть как конкретными «вещественными» (например, та же ручка), так и представлять невещественные понятия – например, событие определенного рода. Моделирование объектов – процесс весьма творческий, и, как правило, включает в себя как выбор наиболее важных свойств и возможных действий реального объекта, так и определение функциональности каждого метода в соответствии с тем, какое реальное действие он должен моделировать.

Язык Java является «более» объектно-ориентированным по сравнению с C++. В Java все является объектом. Подчеркнем основные черты объектно-ориентированного программирования[4,9]:

1. Все является объектом. Объект – своего рода переменная, которой можно посылать сообщения с запросами совершить какие-то операции над собой.

2. Программа – это группа объектов, говорящих друг другу, что делать посредством сообщений. Сообщение можно представить как вызов функции (метода), принадлежащей определенному объекту.

3. Каждый объект имеет собственную память, состоящую из других объектов. Новый объект создается с помощью «укладывания» в него новых объектов.

4. У каждого объекта есть тип. Каждый объект является экземпляром класса, здесь «класс» является аналогом слова «тип».

5. Все объекты определенного типа могут получать одинаковые сообщения. Т.е. это значит, что можно писать код для геометрических фигур, и быть уверенным, что он подойдет для всего, что можно подогнать под понятие геометрической фигуры. Например, задавшись целью определить модель поведения геометрической фигуры, в качестве действий, которые каждая фигура должна «уметь» выполнять, можно определить следующие: фигура должна уметь нарисовать себя, стереть, двигаться на экране, менять свой цвет. И тогда к любому объекту, имеющему тип «фигура», будь то маленький черный квадрат или большой зеленый треугольник, можно обратиться с запросом (сообщением) изменить свой цвет или стереть себя. Конечно, данная модель весьма условна, и служит лишь для демонстрации одного из принципов ООП.

Объекты, идентичные во всем, кроме внутреннего состояния во время работы программы, группируются в «классы объектов». Создание абстрактных типов данных является фундаментальным понятием во всем ООП. В большинстве объектных языках для обозначения новых типов используется слово «класс», поэтому мы тоже будем подразумевать под словом «класс» – понятие «тип» и наоборот.

Как только определен новый класс, программист может создавать сколько угодно объектов этого класса (или, как их еще называют, экземпляров класса) и манипулировать ими так, как будто они представляют собой элементы решаемой задачи.

В Java нет указателей, и когда программист определяет идентификатор для объекта, он тем самым определяет ссылку на него (хотя в Java понятие ссылки отличается от ссылки в C++).

При определении ссылки желательно присоединить ее к новому объекту. В основном это делается с использованием ключевого слова new.

Все объекты Java размещаются в так называемой «куче» (heap). «Куча» – это резерв памяти общего назначения, располагающегося в оперативной памяти. Преимущество «кучи» состоит в том, что компилятору не нужно знать, сколько конкретно выделить памяти на «куче» и как долго существуют находящиеся там объекты. Когда программисту нужно создать объект – он пишет код с использованием слова new, и память выделяется на куче во время выполнения программы. Такой способ выделения памяти является более гибким по сравнению со стеком, но занимает больше времени.

Ссылки на объекты и переменные так называемых примитивных типов располагаются в стеке. Кроме того, в Java есть поддержка native методов C и C++, а значит регистры и стек также доступны опосредованно через такие методы.

5.1. Создание классов. Поля и методы классов

Для объявления класса используется ключевое слово class, за которым следует имя нового класса:

```
class AClass {/* тело класса */}
```

Это предложение вводит новый тип AClass, теперь можно создавать объект этого типа, используя new:

```
AClass ac = new AClass();
```

При создании классов программист может определить два типа элементов класса: данные-члены (поля) и функции-члены (методы).

Данные-члены – это объекты любого типа, с которыми можно общаться посредством ссылки на них, или переменные примитивных типов.

```
class DataOnly {
int i;
float f;
String s;
}
```

Приведенный в примере класс ничего не делает и не может делать, т.к. в нем не определены методы, но его полям можно присвоить значения, и синтаксис в этом случае не отличается от синтаксиса C++ - для обращения к члену класса пишется имя ссылки на объект данного класса, затем точка, затем имя члена:

```
DataOnly d = new DataOnly();
```

d.i = 7;
d.f =4.23f;
d.s = “hello”;

Если явно не проинициализировать примитивные члены-данные, то в Java им гарантированно присваиваются значению по умолчанию.

Примитивный тип (Значения по умолчанию)
boolean - false
char - ‘\u000’ (null)
byte - (byte) 0
short - (short) 0
int - 0
long - 0L
float - 0.0f
double - 0.0d

Отметим, что значение по умолчанию гарантируется только для переменной, являющейся членом класса (потому что могут быть еще локальные переменные для методов и областей видимости), хотя для всех переменных в программе хорошим стилем считается их явная инициализация.

Методы в Java определяют сообщения, которые может принимать объект. Основные части метода - имя, параметры, возвращаемый тип и тело:

```
returnType methodName( /* Argument list */ ) {
/* Method body */
}
```

Имя метода и его список параметров уникальным образом идентифицируют метод. Методы в Java создаются только как части класса. Вызов метода делается только для объекта (или для класса в случае статического метода), и этот объект должен обладать возможностью вызова этого метода. Синтаксис для вызова метода: имяОбъекта.имяМетода(параметр1, параметр2, параметр3,…);

Пример определения (реализации) метода:

```
int toSquare (int i) {
 return i*i ;
}
```

В примере метод возвращает целое значение, равное квадрату числа i. i является параметром метода. Слово return требуется для выполнения двух действий: 1) оно означает «выйти из метода»; 2) если метод возвращает какое-нибудь значение, то это значение помещается сразу за оператором return. Если метод не должен возвращать ничего, то надо в качестве возвращаемого типа указывать void, в этом случае

использование оператора return необязательно – выход из метода произойдет после завершения последнего оператора в теле метода.

Следует отметить, что даже в таком простом примере не удалось обойтись без подводных камней: диапазон значений примитивных типов ограничен, и представленный пример вычислений квадрата целого числа не является безопасным – для больших значений i при вычислении квадрата возможен выход за диапазон значений типа int, и, как результат, получение неверного результата.

Декларирование класса позволяет с помощью спецификаторов указать свойства класса: в приведенном примере присутствует только спецификатор доступа public, но могут быть и другие (final, abstract, extends, implements) – о них поговорим чуть позже.

Тело класса содержит код, необходимый для функциональности объектов данного класса: конструкторы для инициализации новых объектов, декларирование переменных – полей класса, обеспечивающих хранение информации о состоянии объекта, определение методов, реализующих поведение класса (в случае static-методов) и его объектов.

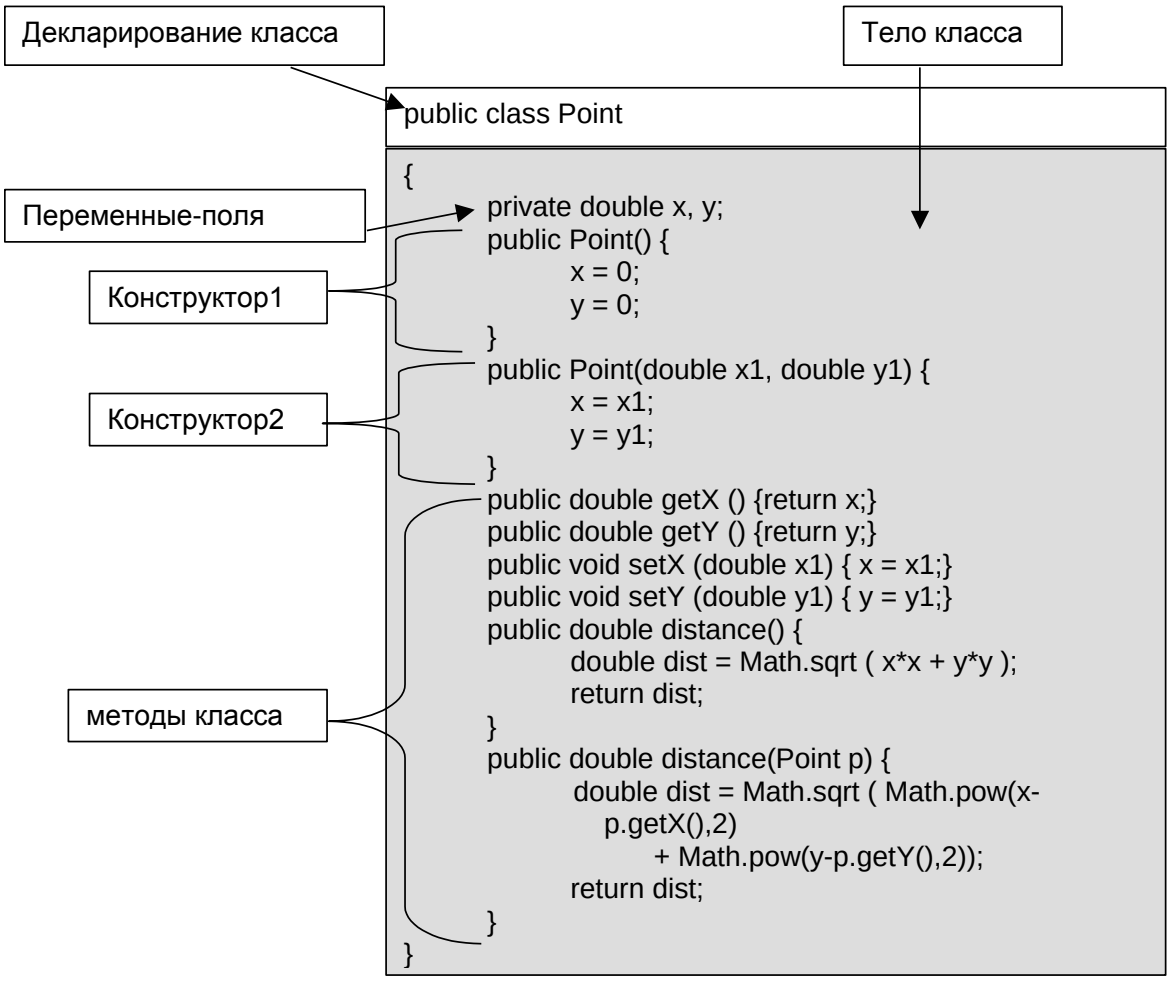

Рис.4.

Параметры методов в Java могут быть любого типа, и их может быть любое количество. В Java аргументы передаются в метод только по значению. Это может показаться программистам, привыкших в другим языкам программирования, странным и неудобным, но не все так плохо – такой подход более соответствует принципам объектно-ориентированного программирования (более, чем подход с передачей аргументов по ссылке). Как правило, метод, если и должен менять какие-то значения, то это значения полей класса, а доступ к полям осуществляется не с помощью механизма параметров, а непосредственно – поля являются глобальными переменными класса.

Структура класса в общем случае показано на рисунке 4[10].

5.2. Массивы Java

Массив – это пронумерованная последовательность объектов или примитивных данных, состоящая из элементов одного типа и хранимая под одним именем (Рис.5.1).

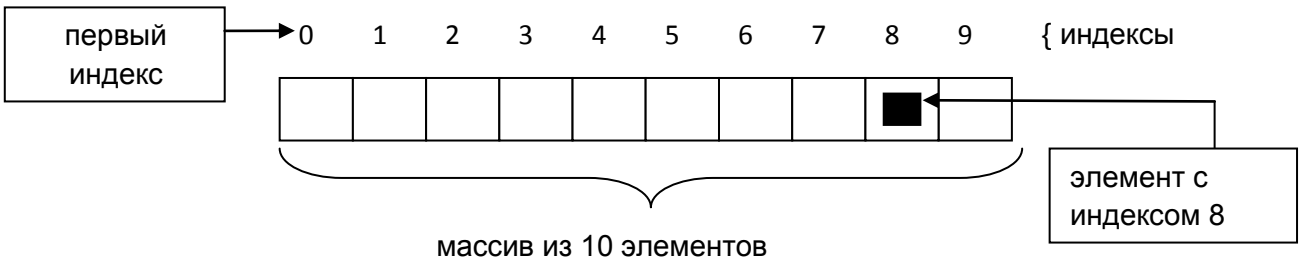

Рис.5.1

Элемент массива – это одно из значений, хранящихся внутри массива, доступ к которому осуществляется по его индексу. Первый индекс (индекс первого элемента массива) всегда имеет значение 0. Тип элементов, которые предполагается хранить в массиве, задается в точке определения массива.

Массивы в языке Java относятся к ссылочным типам и описываются своеобразно, но характерно для ссылочных типов. Описание производится в три этапа.

Первый этап – объявление (declaration). На этом этапе определяется только переменная типа ссылка (reference) на массив, содержащая тип массива. Для этого записывается имя типа элементов массива, квадратными скобками указывается, что объявляется ссылка на массив,

а не простая переменная, и перечисляются имена переменных типа ссылка, например,

double[] a, b;

Здесь определены две переменные – ссылки а и ь на массивы типа double. Можно поставить квадратные скобки и непосредственно после имени. Это удобно делать среди определений обычных переменных:

int I = 0, ar[], k = -1;

Здесь определены две переменные целого типа i и k, и объявлена ссылка на целочисленный массив аг.

Второй этап – определение (installation). На этом этапе указывается количество элементов массива, называемое его длиной, выделяется место для массива в оперативной памяти, переменная-ссылка получает адрес массива. Все эти действия производятся еще одной операцией языка Java – операцией new тип, выделяющей участок в оперативной памяти для объекта указанного в операции типа и возвращающей в качестве результата адрес этого участка. Например,

a = new double[5];

b = new double[100];

ar = new int[50];

Индексы массивов всегда начинаются с 0. Массив а состоит из пяти переменных a[0], a[1], , a[4]. Элемента a[5] в массиве нет. Индексы можно задавать любыми целочисленными выражениями, кроме типа long, например, a[i+j], a[i%5], a[++i]. Исполняющая система Java следит за тем, чтобы значения этих выражений не выходили за границы длины массива.

Третий этап – инициализация (initialization). На этом этапе элементы массива получают начальные значения. Например,

a[0] = 0.01; a[1] = -3.4; a[2] = 2:.89; a[3] = 4.5; a[4] = -6.7;

for (int i = 0; i < 100; i++) b[i] = 1.0 /i;

for (int i = 0; i < 50; i++) ar[i] = 2 * i + 1;

Первые два этапа можно совместить:

doublet] a = new double[5], b = new double[100];

int i = 0, ar[] = new int[50], k = -1;

Можно сразу задать и начальные значения, записав их в фигурных скобках через запятую в виде констант или константных выражений. При этом даже необязательно указывать количество элементов массива, оно будет равно количеству начальных значений;

double[] a = {0.01, -3.4, 2.89, 4.5, -6.7};

Можно совместить второй и третий этап:

a = new double t[] {0.1, 0.2, -0.3, 0.45, -0.02};

Можно даже создать безымянный массив, сразу же используя результат операции new, например, так:

System.out.println (new char[] {'H', 'e', '1', '1', 'o'});

Ссылка на массив не является частью описанного массива, ее можно перебросить на другой массив того же типа операцией присваивания. Например, после присваивания a = b обе ссылки a и b указывают на один и тот же массив из 100 вещественных переменных типа double и содержат один и тот же адрес.

Ссылка может присвоить "пустое" значение null, не указывающее ни на какой адрес оперативной памяти:

ar = null;

После этого массив, на который указывала данная ссылка, теряется, если на него не было других ссылок.

Кроме простой операции присваивания, со ссылками можно производить еще только сравнения на равенство, например, a = b, и неравенство, a != b. При этом сопоставляются адреса, содержащиеся в ссылках, мы можем узнать, не ссылаются ли они на один и тот же массив.

Массивы в Java всегда определяются динамически, хотя ссылки на них задаются статически.

Кроме ссылки на массив, для каждого массива автоматически определяется целая константа с одним и тем же именем length. Она равна длине массива. Для каждого массива имя этой константы уточняется именем массива через точку. Так, после наших определений, константа a.length равна 5, константа b.length равна 100, а ar.length равна 50.

Последний элемент массива a можно записать так: a [a. length - 1], предпоследний – a [a. length - 2] и т. д. Элементы массива обычно перебираются в цикле вида:

```
double aMin = a[0], aMax = aMin;
for (int i = 1; i < a.length; i++){
if <a[i] < aMin) aMin = a[i]; //нахождение минимального элемента
//массива
if (a[i] > aMax) aMax = a[i]; //нахождение максимального элемента
//массива
}
double range = aMax – aMin;
```

Здесь вычисляется range - диапазон значений массива.

Элементы массива – это обыкновенные переменные своего типа, с ними можно производить все операции, допустимые для этого типа:

(a[2] + a[4]) / a[0] и т. д.

5.3. Обработка строк

Как и в других языках программирования, в Java строка это последовательность символов. Но в отличие от многих других языков, в которых строки реализованы как массивы символов, в Java строки реализованы в виде объектов типа String, реализация строк в виде встроенных объектов позволяет Java обеспечить полный комплект средств, делающих управления строками удобным. Например, Java предоставляет методы для сравнения двух строк, поиска подстрок, объединения двух строк и изменения регистра символов в строке. Кроме того объекты String могут быть сконструированы множеством разных способов, что позволяет легко получать строки, когда они требуются.

Что несколько неожиданно, так это тот факт, что когда вы создаете объект типа String, то мы создаем строку, которая не может быть изменена. То есть, как только объект String создан, мы не можем изменить символы, образующие строку. На первый взгляд это может показаться серьезным ограничением. Однако на самом деле это не так уж важно. Вы можете осуществлять любые операции над строками. Особенность в том, что всякий раз, когда вам нужна измененная версия существующей строки, создается новый объект String, включающий все модификации. Оригинальная строка остается неизменной. Этот подход используется потому, что фиксированная, неизменная строка может быть реализована более эффективно, нежели изменяемая. Для тех случаев, когда нужны модифицируемые строки, Java предлагает два выбора: StringBuffer и StringBuilder.

Оба содержат строки, которые могут быть изменены после того, как созданы.

Классы String, StringBuffer и StringBui1der определены в пакете java.lang, поэтому они доступны всем программистам автоматически. Все они объявлены с модификатором fina1, что означает, что ни от одного из них нельзя порождать подклассы. Это позволяет осуществить некоторую оптимизацию, которая повышает производительность общих операций со строками. Все три класса реализуют интерфейс CharSequence. И последнее: когда говорится о том, что строки в объектах типа String неизменяемы, это означает, что содержимое экземпляра String не может быть изменено после его создания. Однако переменная, объявлная как ссылка на String в любой момент может быть переназначена так, чтоб указыватьа другой объект String.

5.3.1. Класс String

Перед работой со строкой ее следует создать. Это можно сделать разными способами.

Самый простой способ создать строку – это организовать ссылку типа string на строку-константу:

String s1 = "Это строка.";

Если константа длинная, можно записать ее в нескольких строках текстового редактора, связывая их операцией сцепления:

String s2 = "Это длинная строка, " +

"записанная в двух строках исходного текста";

Замечание

Не забывайте разницу между пустой строкой string s = "" , не содержащей ни одного символа, и пустой ссылкой string s = null, не указывающей ни на какую строку и не являющейся объектом.

Самый правильный способ создать объект с точки зрения ООП – это вызвать его конструктор в операции new. Класс String предоставляет вам девять конструкторов[4,9]:

string() – создается объект с пустой строкой;

string (String str) – из одного объекта создается другой, поэтому этот конструктор используется редко;

string (StringBuffer str) – преобразованная копия объекта класса BufferString;

string(byte[] byteArray) – объект создается из массива байтов byteArray;

String (char [] charArray) – объект создается из массива charArray символов Unicode;

String (byte [] byteArray, int offset, int count) – объект создается из части массива байтов byteArray, начинающейся с индекса offset и содержащей count байтов;

String (char [] charArray, int offset, int count) – то же, но массив состоит из символов Unicode;

String(byte[] byteArray, String encoding) – символы, записанные в массиве байтов, задаются в Unicode-строке, с учетом кодировки encoding ;

String(byte[] byteArray, int offset, int count, String encoding) – то же самое, но только для части массива.

При неправильном заданий индексов offset , count или кодировки encoding возникает исключительная ситуация.

Конструкторы, использующие массив байтов byteArray , предназначены для создания Unicode-строки из массива байтовых

ASCII-кодировок символов. Такая ситуация возникает при чтении ASCII-файлов, извлечении информации из базы данных или при передаче информации по сети.

Если же на компьютере сделаны местные установки, как говорят на жаргоне "установлена локаль" (locale) (в MS Windows это выполняется утилитой Regional Options в окне Control Panel), то компилятор, прочитав эти установки, создаст символы Unicode, соответствующие местной кодовой странице. В русифицированном варианте MS Windows это обычно кодовая страница CP1251.

Если исходный массив с кириллическим ASCII-текстом был в кодировке CP1251, то строка Java будет создана правильно. Кириллица попадет в свой диапазон '\u0400'—'\u04FF' кодировки Unicode.

Но у кириллицы есть еще, по меньшей мере, четыре кодировки.

В MS-DOS применяется кодировка CP866. В UNIX обычно применяется кодировка KOI8-R. На компьютерах Apple Macintosh используется кодировка MacCyrillic. Есть еще и международная кодировка кириллицы ISO8859-5;

Еще один способ создать строку – это использовать два статических метода

copyValueOf(char[] charArray) и copyValueOf(char[] charArray, int offset, int length).

Они создают строку по заданному массиву символов и возвращают ее в качестве результата своей работы. Например, после выполнения следующего фрагмента программы

```
chart] c = ('С', 'и', 'м', 'в', 'о' , 'л', 'ь', 'н', 'ы', 'й'};
String s1 = String.copyValueOf(c);
String s2 = String.copyValueOf(c, 3, 7);
```

получим в объекте s1 строку " Символьный ", а в объекте s2 – строку " вольный ".

Рассмотрим основные методы работы со строками.

Сцепление строк

Со строками можно производить операцию сцепления строк (concatenation), обозначаемую знаком плюс +. Эта операция создает новую строку, просто составленную из состыкованных первой и второй строк, как показано в начале данной главы. Ее можно применять и к константам, и к переменным.

Например:

```
String attention = "Внимание: ";
String s = attention + "неизвестный символ";
```

Вторая операция – присваивание += – применяется к переменным в левой части:

attention += s;

Поскольку операция + перегружена (переопределена) со сложения чисел на сцепление строк, встает вопрос о приоритете этих операций. У сцепления строк приоритет выше, чем у сложения, поэтому, записав "2" + 2 + 2 , получим строку " 222 ". Но, записав 2 + 2 + "2" , получим строку "42", поскольку действия выполняются слева направо. Если же запишем "2" + (2 + 2) , то получим "24".

Как узнать длину строки

Для того чтобы узнать длину строки, т. е. количество символов в ней, надо обратиться к методу length() :

```
String s = "Write once, run anywhere.";
int len = s.length{);
```

или еще проще

```
int len = "Write once, run anywhere.".length();
```

поскольку строка-константа – полноценный объект класса string . Заметьте, что строка – это не массив, у нее нет поля length.

Как выбрать символы из строки

Выбрать символ с индексом ind (индекс первого символа равен нулю) можно методом

charAt(int ind).

Если индекс ind отрицателен или не меньше чем длина строки, возникает исключительная ситуация. Например, после определения

char ch = s.charAt(3);

переменная ch будет иметь значение 't'.

Все символы строки в виде массива символов можно получить методом toCharArray(), возвращающим массив символов.

Если же надо включить в массив символов dst , начиная с индекса ind массива подстроку от индекса begin включительно до индекса end исключительно, то используйте метод

getChars(int begin, int end, char[] dst, int ind) типа void .

В массив будет записано end - begin символов, которые займут элементы массива, начиная с индекса ind до индекса ind + (end - begin) - 1 .

Если надо получить массив байтов, содержащий все символы строки в байтовой кодировке ASCII, то используйте метод getBytes().

Этот метод при переводе символов из Unicode в ASCII использует локальную кодовую таблицу.

Если же надо получить массив байтов не в локальной кодировке, а в какой-то другой, используйте метод getBytes(String encoding).

Как выбрать подстроку

Метод

substring (int begin, int end)

выделяет подстроку от символа с индексом begin включительно до символа с индексом end исключительно. Длина подстроки будет равна end - begin .

Метод

substring (int begin)

выделяет подстроку от индекса begin включительно до конца строки.

Если индексы отрицательны, индекс end больше длины строки или begin больше чем end, то возникает исключительная ситуация.

Например, после выполнения

String s = "Write once, run anywhere.";

String sub1 = s.substring (6, 10);

String sub2 = s.substring (16);

получим в строке sub1 значение " once ", а в sub2 – значение " anywhere ".

Как сравнить строки

Операция сравнения == сопоставляет только ссылки на строки. Она выясняет, указывают ли ссылки на одну и ту же строку. Например, для строк

String s1 = "Какая-то строка";

String s2 = "Другая строка";

сравнение s1 == s2 дает в результате false .

Значение true получится, только если обе ссылки указывают на одну и ту же строку, например, после присваивания si = s2 .

Интересно, что если мы определим s2 так:

String s2 == "Какая-то строка";

то сравнение s1 == s2 даст в результате true , потому что компилятор создаст только один экземпляр константы "Какая-то строка" и направит на него все ссылки.

Вы, разумеется, хотите сравнивать не ссылки, а содержимое строк. Для этого есть несколько методов.

Логический метод

equals (object obj) ,

переопределенный из класса object , возвращает true , если аргумент obj не равен null , является объектом класса string , и строка, содержащаяся в нем, полностью идентична данной строке вплоть до

совпадения регистра букв. В остальных случаях возвращается значение false .

Логический метод

equalsIgnoreCase (object obj)

работает так же, но одинаковые буквы, записанные в разных регистрах, считаются совпадающими.

Например, s2.equals ("другая строка") даст в результате false , а s2.equalsIgnoreCase ("другая строка") возвратит true .

Метод

compareTo (string str)

возвращает целое число типа int , вычисленное по следующим правилам:

Сравниваются символы данной строки this и строки str с одинаковым индексом, пока не встретятся различные символы с индексом, допустим k , или пока одна из строк не закончится.

В первом случае возвращается значение this.charAt(k) - str.charAt(k), т. е. разность кодировок Unicode первйх несовпадающих символов.

Во втором случае возвращается значение this.length() - str.length() , т. е. разность длин строк.

Если строки совпадают, возвращается 0.

Если значение str равно null , возникает исключительная ситуация.

Нуль возвращается в той же ситуации, в которой метод equals() возвращает true .

Метод

compareToignoreCase (string str)

производит сравнение без учета регистра букв, точнее говоря, выполняется метод

this.toUpperCase ().toLowerCase ().compareTo (

str.toUpperCase ().toLowerCase ());

Еще один метод–

compareTo (Object obj)

создает исключительную ситуацию, если obj не является строкой. В остальном он работает как метод compareTo(String str).

Эти методы не учитывают алфавитное расположение символов в локальной кодировке.

Русские буквы расположены в Unicode по алфавиту, за исключением одной буквы. Заглавная буква Ё расположена перед всеми кириллическими буквами, ее код '\ u040l ', а строчная буква е – после всех русских букв, ее код '\ u0451 '.

Сравнить подстроку данной строки this с подстрокой той же длины len другой строки str можно логическим методом

regionMatches (int ind1, String str, int ind2, int len)

Здесь ind1 – индекс начала подстроки данной строки this, ind2 – индекс начала подстроки другой строки str .

Этот метод различает символы, записанные в разных регистрах. Если надо сравнивать подстроки без учета регистров букв, то используйте логический метод:

regionMatches (boolean flag, int ind1, String str, int ind2, int len)

Если первый параметр flag равен true , то регистр букв при сравнении подстрок не учитывается, если false – учитывается.

Как найти символ в строке

Поиск всегда ведется с учетом регистра букв.

Первое появление символа ch в данной строке this можно отследить методом

indexOf (int ch) ,

возвращающим индекс этого символа в строке или -1 , если символа ch в строке this нет.

Например,

"Молоко", indexOf ('o') выдаст в результате 1 .

Конечно, этот метод выполняет в цикле последовательные сравнения this.charAt(k++> == ch , пока не получит значение true .

Второе и следующие появления символа ch в данной строке this можно отследить методом

index (int ch, int ind) .

Этот метод начинает поиск символа ch с индекса ind . Если ind < 0, то поиск идет с начала строки, если ind больше длины строки, то символ не ищется, т. е. возвращается -1.

Например,

"молоко".indexof ('o', indexof ('o') + 1) даст в результате 3.

Последнее появление символа ch в данной строке this отслеживает метод lastIndexof (int ch). Он просматривает строку в обратном порядке. Если символ ch не найден, возвращается.-1.

Например, "Молоко".lastindexof ('o') даст в результате 5.

Предпоследнее и предыдущие появления символа ch в данной строке this можно отследить методом

lastIndexof (int ch, int ind) ,

который просматривает строку в обратном порядке, начиная с индекса ind .

Если ind больше длины строки, то поиск идёт от конца строки, если ind < 0, то возвращается-1.

Как найти подстроку

Поиск всегда ведется с учетом регистра букв.

Первое вхождение подстроки sub в данную строку this отыскивает метод

indexof (String sub).

Он возвращает индекс первого символа первого вхождения подстроки sub в строку или -1, если подстрока sub не входит в строку this .

Например, " Раскраска ".indexof ("рас") даст в результате 4.

Если вы хотите начать поиск не с начала строки, а с какого-то индекса ind , используйте метод

indexOf (String sub, int ind).

Если ind < 0 , то поиск идет с начала строки, если ind больше длины строки, то символ не ищется, т. е. возвращается -1.

Последнее вхождение подстроки sub в данную строку this можно отыскать методом

lastindexof (string sub),

возвращающим индекс первого символа последнего вхождения подстроки sub в строку this или (-1), если подстрока sub не входит в строку this .

Последнее вхождение подстроки sub не во всю строку this , а только в ее начало до индекса ind можно отыскать методом

lastIndexof (String stf, int ind).

Если ind больше длины строки, то .поиск идет от конца строки, если ind < o , то возвращается -1.

Для того чтобы проверить, не начинается ли данная строка this с подстроки sub , используйте логический метод

startsWith (string sub) ,

возвращающий true , если данная строка this начинается с подстроки sub , или совпадает с ней, или подстрока sub пуста.

Можно проверить и появление подстроки sub в данной строке this , начиная с некоторого индекса ind логическим методом

startsWith (String sub, int ind).

Если индекс ind отрицателен или больше длины строки, возвращается false.

Для того чтобы проверить, не заканчивается ли данная строка this подстрокой sub , используйте логический метод

endsWitht (String sub) .

Учтите, что он возвращает true , если подстрока sub совпадает со всей строкой или подстрока sub пуста.

Например, if (fileName.endsWith(". Java ")) отследит имена файлов с исходными текстами Java .

Перечисленные выше методы создают исключительную ситуацию, если

sub == null.

Если вы хотите осуществить поиск, не учитывающий регистр букв, измените предварительно регистр всех символов строки.

Как изменить регистр букв

Метод

toLowerCase ()

возвращает новую строку, в которой все буквы переведены в нижний регистр, т. е. сделаны строчными.

Метод

toUpperCase ()

возвращает новую строку, в которой все буквы переведены в верхний регистр, т. е. сделаны прописными.

При этом используется локальная кодовая таблица по умолчанию. Если нужна другая локаль, то применяются методы toLowerCase(Locale l oc) и toUpperCase (Locale loc).

Как заменить отдельный символ

Метод

replace (int old, int new)

возвращает новую строку, в которой все вхождения символа old заменены символом new . Если символа old в строке нет, то возвращается ссылка на исходную строку.

Например, после выполнения

" Рука в руку сует хлеб" , replace ('y', 'e')

получим строку

" Река в реке сеет хлеб".

Регистр букв при замене учитывается.

Как убрать пробелы в начале и конце строки

Метод

trim ()

возвращает новую строку, в которой удалены начальные и конечные символы с кодами, не превышающими '\u0020 '. Например,

String pause = " Hmmm ";

pause.trim () возвращает строку "Hmmm"

Как преобразовать данные другого типа в строку

В языке Java принято соглашение – каждый класс отвечает за преобразование других типов в тип этого класса и должен содержать нужные для этого методы.

Класс string содержит восемь статических методов

valueof (type elem)

преобразования в строку примитивных типов boolean, char, int, long, float, double , массива char[], и просто объекта типа object.

Девятый метод

valueof (char[] ch, int offset, int len)

преобразует в строку подмассив массива ch , начинающийся с индекса offset и имеющий len элементов.

Кроме того, в каждом классе есть метод

toString () ,

переопределенный или просто унаследованный от класса Object . Он преобразует объекты класса в строку. Фактически, метод valueOf() вызывает метод tostring () соответствующего класса. Поэтому результат преобразования зависит от того, как реализован метод toString ().

Еще один простой способ – сцепить значение elem какого-либо типа с пустой строкой: "" + elem. При этом неявно вызывается метод elem. toString ().

5.3.2. Класс StringBuffer

Объекты класса StringBuffer – это строки переменной длины. Только что созданный объект имеет буфер определенной емкости (capacity), по умолчанию достаточной для хранения 16 символов. Емкость можно задать в конструкторе объекта.

Как только буфер начинает переполняться, его емкость автоматически увеличивается, чтобы вместить новые символы.

В любое время емкость буфера можно увеличить, обратившись к методу ensureCapacity(int minCapacity)

Этот метод изменит емкость, только если minCapacity будет больше длины хранящейся в объекте строки. Емкость будет увеличена по следующему правилу. Пусть емкость буфера равна N. Тогда новая емкость будет равна

Max(2 * N + 2, minCapacity)

Таким образом, емкость буфера нельзя увеличить менее чем вдвое.

Методом

setLength (int newLength)

можно установить любую длину строки.

Если она окажется больше текущей длины, то дополнительные символы будут равны ' \uOOOO' . Если она будет меньше текущей длины, то строка будет обрезана, последние символы потеряются,

точнее, будут заменены символом '\uOOOO' . Емкость при этом не изменится.

Количество символов в строке можно узнать, как и для объекта класса String , методом length () , а емкость – методом capacity ().

Создать объект класса stringBuffer можно только конструкторами.

Конструкторы

В классе stringBuffer три конструктора:

stringBuffer () – создает пустой объект с емкостью 16 символов;

stringBuffer (int capacity) – создает пустой объект заданной емкости capacity ;

StringBuffer (String str) – создает объект емкостью str.length () + 16, содержащий строку str .

Как добавить подстроку

В классе stringBuffer есть десять методов append (), добавляющих подстроку в конец строки. Они не создают новый экземпляр строки, а возвращают ссылку на ту же самую, но измененную строку.

Основной метод append (string str) присоединяет строку str в конец данной строки. Если ссылка str == null, то добавляется строка "null".

Шесть методов append (type elem) добавляют примитивные типы boolean, char, int, long, float, double, преобразованные в строку.

Два метода присоединяют к строке массив str и подмассив sub символов,

преобразованные в строку:

append (char [] str)

append (char [] , sub, int offset, int len).

Десятый метод добавляет просто объект append (Object obj). Перед этим объект obj преобразуется в строку своим методом Tostring ().

Метод insert идентичен методу append в том смысле, что для каждого возможного типа данных существует своя совмещенная версия этого метода.

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. Например:

StringBuffer sb = new StringBuffer ("hello world !");

sb.insert (6,"there ");

При выполнении метода insert существующие символы сдвигаются, чтобы освободить место для вставки новых символов.

Как удалить подстроку

Метод

delete (int begin, int end)

удаляет из строки символы, начиная с индекса begin включительно до индекса end исключительно, если end больше длины строки, то до конца строки.

Например, после выполнения

String s = new StringBuffer ("Это небольшая строка");

s.delete (4, 6).toString();

получим s == "Это большая строка".

Если begin отрицательно, больше длины строки или больше end , возникает исключительная ситуация.

Если begin == end, удаление не происходит.

Как удалить символ

Метод

DeleteCharAt (int ind)

удаляет символ с указанным индексом ind . Длина строки уменьшается на единицу.

Если индекс ind отрицателен или больше длины строки, возникает исключительная ситуация.

Как заменить подстроку

Метод

replace (int begin, int end, String str)

удаляет символы из строки, начиная с индекса begin включительно до индекса end исключительно, если end больше длины строки, то до конца строки, и вставляет вместо них строку str.

Если begin отрицательно, больше длины строки или больше end , возникает исключительная ситуация.

Разумеется, метод replace () – это последовательное выполнение методов

delete () и insert ().

Как перевернуть строку

Метод

reverse()

меняет порядок расположения символов в строке на обратный порядок.

Например, после выполнения

String s = new StringBuffer("Это небольшая строка");

s.reverse().toString();

получим s == "акортс яашьлобен отЭ".

В Java 2SE 5 в дополнение к существующим богатым возможностям обработки строк появился новый строковый класс. Этот новый класс называется StringBui1der. Он идентичен StringBuffer за исключением одного важного отличия: он не синхронизирован, что означает, что он не является безопасным в отношении потоков. Выгода от применения StringBuilder связана с более высокой производительностью. Однако в случае разработки многопоточных программ рекомендуется использовать StringBuffer, а не StringBui1der.

6. ВВЕДЕНИЕ В ООП

6.1. Принципы объектно-ориентированного программирования

Все языки объектно-ориентированного программирования предоставляют механизмы, которые облегчают реализацию объектно-ориентированной модели. Этими механизмами являются инкапсуляция, наследование и полиморфизм [7,8,12]. Рассмотрим эти концепции.

Инкапсуляция

Инкапсуляция механизм, который связывает код и данные, которыми он манипулирует, защищая оба эти компонента от внешнего вмешательства и злоупотреблений. Один из возможных способов представления инкапсуляции представление в виде защитной оболочки, которая предохраняет код и данные от произвольного доступа со стороны другого кода, находящегося снаружи оболочки. Доступ к коду и данным, находящимся внутри оболочки, строго контролируются тщательно определенным интерфейсом. Чтобы провести аналогию с реальным миром, рассмотрим автоматическую коробку передач автомобиля. Она инкапсулирует сотни бит информации об автомобиле, такой как степень ускорения, крутизна поверхности, по которой совершается движение и положение рычага переключения скоростей. Пользователь (водитель) может влиять на эту сложную инкапсуляцию только одним методом: перемещая рычаг переключения скоростей. На коробку передач нельзя влиять, например, посредством индикатора поворота или дворников.

Таким образом, рычаг переключения скоростей строго определенный (а в действительности единственный) интерфейс к

коробке передач. Более того, происходящее внутри коробки передач, не влияет на объекты, находящиеся вне ее. Например, переключение передач не включает фары! Поскольку функция автоматического переключения передач инкапсулирована, десятки изготовителей автомобилей могут реализовать ее каким угодно способом. Однако с точки зрения водителя все эти коробки передач работают одинаково.

Аналогичную идею можно при менять к программированию. Сила инкапсулированного кода в том, что все знают, как к нему можно получить доступ, и, следовательно, могут его использовать независимо от нюансов реализации и не опасаясь неожиданных побочных эффектов. В языке Java основой инкапсуляции является класс. Хотя подробнее мы рассмотрим

классы в последующих разделах, сейчас полезно ознакомиться со следующим кратким описанием. Класс определяет структуру и поведение (данные и код), которые будут совместно использоваться набором объектов. Каждый объект данного класса содержит структуру и поведение, которые определены классом, как если бы объект был "отлит" в форме класса. Поэтому иногда объекты называют экземплярами 'класса. Таким образом, класс это логическая конструкция, а объект имеет физическое воплощение.

При создании класса определяют код и данные, которые образуют этот класс. Совокупность этих элементов называют членами класса. В частности, определенные классом данные называют переменными членами или переменными экземпляра. Код, который выполняет действия по отношению к данным, называют переменными методами или просто методами. (То, что программисты на Java называют методом, программисты на C/C++ называют функциями.) В правильно написанных Java программах методы определяют способы использования переменных членов. Это означает, что поведение и интерфейс класса определяются методами, которые выполняют действия по отношению к данным его экземпляра. Поскольку назначение класса инкапсуляция сложной структуры программы, существуют механизмы сокрытия сложной структуры реализации внутри класса. Каждый метод или переменная в классе может быть помечена как закрытая или общедоступная. Общедоступный интерфейс класса представляет все, что должны или могут знать внешние пользователи класса. Закрытые методы и данные могут быть доступны только для кода, который является членом данного класса. Следовательно, любой другой код, не являющийся членом класса, не может получать доступ к приватному методу или переменной. Поскольку приватные члены класса доступны другим частям

программы только посредством общедоступных методов класса, можно быть уверенным в невозможности выполнения неправомерных действий. Конечно, это означает, что общедоступный интерфейс должен быть тщательно спроектирован, открывая не слишком много нюансов внутренней работы класса

Наследование

Наследование процесс, посредством которого один объект получает свойства другого. объекта. Это важно, поскольку он поддерживает концепцию иерархической классификации. Как уже отмечалось, большинство сведений становится доступным для понимания посредством иерархических (т.е. нисходящих) классификаций. Без использования иерархий каждый объект должен был бы явно определять все свои характеристики. Однако благодаря наследованию объект должен определять только те из них, которые делают его уникальным внутри класса. Объект может наследовать общие атрибуты от своего родительского объекта. Таким образом, механизм наследования обеспечивает возможность того, чтобы один объект был особым экземпляром более общего случая.

Наследование связано также с инкапсуляцией. Если данный класс инкапсулирует определенные атрибуты, то любой его подкласс будет иметь эти же атрибуты плюс любые дополнительные атрибуты, являющиеся составной частью его специализации. Эта ключевая концепция делает возможным возрастание сложности объектно-ориентированных программ в линейной, а не геометрической прогрессии. Новый подкласс наследует все атрибуты всех своих родительских классов. Поэтому он не содержит непредсказуемых взаимодействий с большей частью остального кода системы.

Полиморфизм

Полиморфизм (от греческого слова, означающего "много форм") свойство, которое позволяет использовать один и тот же интерфейс для общего класса действий. Конкретное действие определяется конкретным характером ситуации. В более общем виде концепцию полиморфизма часто выражают фразой "один интерфейс, несколько методов". Это означает, что можно спроектировать общий интерфейс для группы связанных между собой действий. Этот подход позволяет уменьшить сложность программы, поскольку один и тот же интерфейс используется для указания общего класса действий. Выбор же конкретного действия (т.е. метода), применимого к каждой ситуации задача компилятора. Программисту не нужно осуществлять этот выбор вручную. Необходимо лишь помнить об общем интерфейсе и применять его.

Совместное использование полиморфизма, инкапсуляции и наследования

При правильном совместном использовании полиморфизма, инкапсуляции и наследования они создают среду программирования, которая поддерживает разработку более устойчивых и масштабируемых программ, чем в случае применения модели, ориентированной на процессы[8]. Тщательно спроектированная иерархия классов основа многократного использования кода, на разработку и тестирование которого были затрачены время и усилия. Инкапсуляция позволяет возвращаться к ранее созданным реализациям, не разрушая код, зависящий от общедоступного интерфейса применяемых в приложении классов. Полиморфизм позволяет создавать понятный, чувствительный, удобочитаемый и устойчивый код. Пример с автомобилем достаточно полно иллюстрирует возможности объектно-ориентированного проектирования. Садясь за руль различных типов (подклассов) автомобилей, все водители используют наследование. Независимо от Того, является ли автомобиль школьным автобусом, легковым мерседесом, порше или семейным микроавтобусом, все водители могут более менее легко найти и пользоваться рулем, тормозами и педалью акселератора. Немного помучившись с рычагом переключения передач, большинство людей может даже оценить различия между ручной и автоматической коробками передач. Это становится возможным благодаря получению четкого представления об общем родительском классе этих объектов системе передач.

В процессе использования автомобилей люди постоянно взаимодействуют с их инкапсулированными характеристиками. Педали тормоза и газа скрывают невероятную сложность соответствующих объектов за настолько простым интерфейсом, что управлять этими объектами можно простым нажатием на педаль. Конкретная реализация двигателя, тип тормозов и размер шин не оказывают никакого влияния на способ взаимодействия с определением класса педалей.

Последний атрибут, полиморфизм, четко отражает способность компаний изготовителей автомобилей предлагать широкое множество вариантов, по сути, одного и того же средства передвижения. Например, на автомобиле могут быть установлены система тормозов с защитой от блокировки или традиционные тормоза, рулевая система с гидроусилителем или с реечной передачей и 4, 6 или 8-цилиндровые двигатели. В любом случае нужно будет жать на педаль тормоза, чтобы остановиться, вращать руль, чтобы повернуть, и жать педаль акселератора, чтобы автомобиль двигался. Один и тот же интерфейс может применяться для управления множеством различных реализаций.

Именно совместное применение инкапсуляции, наследования и полиморфизма позволяет преобразовать отдельные детали в объект, который мы называем автомобилем. Сказанное применимо также к компьютерным программам. За счет применения объектно-ориентированных принципов различные части сложной программы могут быть собраны воедино, образуя надежную, пригодную для обслуживания программу. Как было отмечено в начале этого раздела, каждая Java программа является объектно-ориентированной. Или, точнее, каждая Java программа использует инкапсуляцию, наследование и полиморфизм. Хотя на первый взгляд может показаться, что не все эти свойства используются в приведенных в последующих разделах коротких примерах, тем не менее, они в них присутствуют. Как вы вскоре убедитесь, многие функции, предоставляемые языком Java , являются составной частью его встроенных библиотек классов, в которых интенсивно применяются инкапсуляция, наследование и полиморфизм.

6.2. Абстрактные классы

Абстрактные классы представляют собой исключительно полезную концепцию объектно-ориентированного программирования. С их помощью можно объявлять классы, реализованные лишь частично, поручив полную реализацию расширенным классам (потомкам).

Абстракция оказывается полезной, когда некоторое поведение характерно для большинства или всех объектов данного класса, но некоторые аспекты имеют смысл лишь для ограниченного круга объектов, не составляющих суперкласса. В Java такие классы объявляются с ключевым словом abstract, и каждый метод, не реализованный в классе, также объявляется absract.

Такие классы нужны только в качестве общего интерфейса для своих производных классов. Компилятор запрещает создание объектов абстрактного класса. Для декларирования класса как абстрактного нужно в его определении перед словом class поместить ключевое слово abstract:

abstract class AnAbstractClassm{ /*…*/}

Абстрактный класс может содержать абстрактные методы – методы, у которых есть лишь определение и отсутствует тело. Реализация таких методов должна определяться в производных классах такого абстрактного класса. Для объявления метода абстрактным также используется ключевое слово abstract:

public abstract void abAbstractMethod();

В абстрактном классе может не быть абстрактных методов. Но если в классе есть хоть один абстрактный метод, то класс должен быть объявлен абстрактным, в противном случае компилятор выдаст ошибку. При наследовании от абстрактного класса в производном классе нужно определять реализацию всех абстрактных методов, иначе и производный класс окажется абстрактным (и компилятор вынудит и в его объявлении использовать слово abstract).

Например:

```
//Абстрактный класс “Фигура”
abstract public class Shape {
// Цвет фигуры
int Color;
// Начальная точка фигуры
Coordinates StartPoint;
// Нарисовать фигуру
abstract public void Draw();
}
// Конкретный класс "Круг“
class Circle extends Shape {
//Нарисовать круг
public void Draw() {
// Здесь определяется метод, который рисует круг
}
}
```

6.3. Интерфейсы

Определение интерфейса во многом подобно определению класса. Общая форма интерфейса имеет следующий вид:

доступ interface *имя {*

возвращаемый_тип имя_методаl (список_параметров) ;
возвращаемый_тип имя_метода2 (список_параметров) ;
тип имя_конечной_переменнойl = значение;
тип имя_конечной_ переменной2 = значение;
// ...
Возвращаемый_тип имя_ методаN (список_параметров) ;
тип имя_конечной_переменнойN = значение;
}

Если. определение не содержит никакого спецификатора доступа, используется доступ по умолчанию, и интерфейс доступен только другим членам того пакета, в котором он объявлен. Если интерфейс

объявлен как public, он может быть использован любым другим кодом. В этом случае интерфейс должен быть единственным общедоступным интерфейсом, объявленным в файле, и имя файла должно совпадать с именем интерфейса. *Имя* – имя интерфейса, которым может быть любой допустимый идентификатор. Обратите внимание, что объявляемые методы не содержат тел. Их объявления завершаются списком параметров, за которым следует символ точки с запятой. По сути, они представляют собой абстрактные методы. Ни один из указанных внутри интерфейса методов не может обладать никакой заданной по умолчанию реализацией. Каждый класс, который включает в себя интерфейс, должен реализовать все его методы.

Переменные могут быть объявлены внутри объявлений интерфейсов. Они неявно объявляются как final и static, т.е. реализующий класс не может их изменять. Кроме того, они должны быть также инициализированы. Все методы и переменные неявно объявляются как public.

Ниже приведен пример определения интерфейса. В нем объявляется простой интерфейс, который содержит один метод callback () , принимающий единственный целочисленный параметр.

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

Реализация интерфейсов

Как только интерфейс определен, его может реализовать один или более классов. Чтобы реализовать интерфейс, в определение класса потребуется включить конструкцию implements, а затем создать методы, определенные интерфейсом. Общая форма класса,

который содержит выражение implements, имеет следующий вид:

```
доступ class имя_класса [extends суперкласс]
[implements интерфейс1 [,интерфейс2...]]{
// тело_ класса
}
```

Если класс реализует более одного интерфейса, имена интерфейсов разделяются запятыми. Если класс реализует два интерфейса, которые объявляют один и тот же метод, то один и тот же метод будет использоваться клиентами любого интерфейса. Методы, которые реализуют интерфейс, должны быть объявлены как public. Кроме того, сигнатура типа реализующего метода должна в точности совпадать с сигнатурой типа, указанной в определении interface.

Рассморим небольшой пример класса, который реализует приведенный ранее интерфейс Callback.

```
class Client implements Callback {
//Реализует интерфейс Callback
 public void callback (int p) {
  System.out.println("Метод callback, вызванный со значением" +
 p);
 }
}
```

Обратите внимание, что метод callback() объявлен с использованием спецификатора доступа public.

Переменные можно объявлять как объектные ссылки, которые используют тип интерфейса, а не тип класса. Посредством такой переменной можно ссылаться на любой экземпляр любого класса, реализующего объявленный интерфейс. При вызове метода с помощью одной из таких ссылок выбор нужной версии будет производиться в зависимости от конкретного экземпляра интерфейса, на который выполняется ссылка. Это одна из главных особенностей интерфейсов. Поиск выполняемого метода осуществляется динамически во время выполнения, что позволяет создавать классы позже, чем код, который вызывает методы по отношению к этим классам. Диспетчеризация кода может выполняться посредством интерфейса без необходимости наличия каких-либо сведений о "вызывающем". Если класс содержит интерфейс, но не полностью реализует определенные им методы, он должен быть объявлен как abstract (абстрактный). С помощью интерфейсов в Java реализуется множественное наследование, поскольку класс может “наследовать” несколько интерфейсов.

6.4. Классы-оболочки

Для большинства примитивных данных в языке Java существуют классы, представляющие значения данного типа. Эти классы-оболочки (wrapper classes) обладают двумя основными возможностями[3].

Первая - в них находятся методы и переменные, относящиеся к типу (например, методы строковых преобразований и константы для границ диапазонов - Integer.MAX_VALUE, Float.MIN_VALUE и т.д.).

Вторая - создание объектов, содержащих значения определенного примитивного типа, для универсальных типов, умеющих работать только со ссылками на Object. Например, объекты класса Hashtable могут содержать только ссылки на объекты, а не на примитивные типы. Чтобы использовать int в качестве ключа или элемента в объекте

Hashtable, необходимо создать объект Integer, содержащий нужное значение:

Integer keyObj = new Integer (key);

Integer n = Integer (101);

Перечислим методы, которые имеются в каждом из классов - оболочек.

- Конструктор, который получает значение соответствующего примитивного типа и создает объект класса.
- Конструктор, который определяет исходное значение объекта по единственному параметру типа String.
- Метод toString, который возвращает строковое представление объекта.
- Метод typeValue, который возвращает значение примитивного типа - Character.charValue, Boolean.booleanValue и т.д.(например int i = n.intValue (); // n - имя объекта класса Integer
- Метод equals, который определяет, равны ли между собой объекты, относящиеся к одному классу

Рассмотрим основные классы-оболочки.

Класс Boolean

Это очень небольшой класс, предназначенный главным образом для того, чтобы передавать логические значения в методы по ссылке.

Конструктор Boolean (String s) создает объект, содержащий значение true , если строка s равна " true " в любом сочетании регистров букв, и значение false – для любой другой строки.

Логический метод booieanvalue() возвращает логическое значение, хранящееся в объекте.

Класс Character

В этом классе собраны статические константы и методы для работы с отдельными символами.

Статический метод

digit(char ch, in radix)

переводит цифру ch системы счисления с основанием radix в ее числовое значение типа int .

Статический метод

forDigit(int digit, int radix)

производит обратное преобразование целого числа digit в соответствующую цифру (тип char) в системе счисления с основанием radix.

Основание системы счисления должно находиться в диапазоне от Character.MIN_RADIX до Character.MAX_RADIX.

Метод tostring() переводит символ, содержащийся в классе, в строку с тем же символом.

Статические методы toLowerCase() , touppercase(), toTitieCase() возвращают символ, содержащийся в классе, в указанном регистре. Последний из этих методов предназначен для правильного перевода в верхний регистр четырех кодов Unicode, не выражающихся одним символом.

Числовые классы

В каждом из шести числовых классов-оболочек есть статические методы преобразования строки символов типа string лредставляющей число, в соответствующий примитивный тип: Byte.parseByte(), Double.parseDouble(), Float.parseFloat(), Integer.parselnt(), Long.parseLong(), Short.parseShort() . Исходная строка типа string , как всегда в статических методах, задается как аргумент метода. Эти методы полезны при вводе данных в поля ввода, обработке параметров командной строки, т. е. всюду, где числа представляются строками цифр со знаками плюс или минус и десятичной точкой.

В каждом из этих классов есть статические константы MAX_VALUE и MIN_VALUE , показывающие диапазон числовых значений соответствующих примитивных типов. В классах Double и Float есть еще константы POSITIVE_INFINITY, NEGATIVE_INFINITY, NaN и логические методы проверки isNan() , isInfinite().

6.5. Права доступа

Во многих языках существуют права доступа, которые ограничивают возможность использования, например, переменной в классе. Два крайних вида прав доступа: это модификатор public, когда поле доступно из любой точки программы, и private, когда поле может использоваться только внутри того класса, в котором оно объявлено.

Функциональность класса необходимо разделять на открытый интерфейс, описывающий действия, которые будут использовать внешние типы, и на внутреннюю реализацию, которая применяется только внутри самого класса. Внешний интерфейс в дальнейшем модифицировать невозможно, или очень сложно, для больших систем, поэтому его требуется продумывать особенно тщательно. Детали внутренней реализации могут быть изменены на любом этапе, если они не меняют логику работы всего класса. Благодаря такому подходу реализуется одна из базовых характеристик объектной модели –

инкапсуляция, и обеспечивается важное преимущество технологии ООП – модульность.

Таким образом, модификаторы доступа вводятся не для защиты типа от внешнего пользователя, а, напротив, для защиты, или избавления, пользователя от излишних зависимостей от деталей внутренней реализации. Что же касается неправильного применения класса, то его создателям нужно стремиться к тому, чтобы класс был прост в применении, тогда таких проблем не возникнет, ведь программист не станет намеренно писать код, который порождает ошибки в его программе.

В Java используется 4 уровня доступа:

public

private

protected

если не указан ни один из этих трех типов, то уровень доступа определяется по умолчанию (default), иногда его называют пакетным.

Прежде чем рассматривать модификаторы доступа дадим определение пакета.

Современное объектно-ориентированное программирование, прежде всего, ориентировано на использование так называемых библиотек. Библиотеки – это набор готовых программ или законченных фрагментов кода (в Java они представлены в виде классов и интерфейсов), реализующих определенную функциональность. Когда у программиста возникает потребность в функциональности, которая уже реализована в каком-то из классов библиотеки, программист может воспользоваться этой функциональностью, соответствующим образом выполнив вызов метода необходимого класса. Проблема лишь в том, что заранее неизвестно – реализована ли уже требуемая функциональность и если реализована – то в каком классе, и решить эту проблему может помочь только хорошо организованная документация библиотеки. Библиотека в этом случае выполняет функцию организации хранения классов и доступа к ним. В Java эта организация достигается путем использования пакетов. Пакеты позволяют управлять «пространством имен» классов, решая проблему пересечения имен классов.

Пакет (package) – коллекция связанных между собой классов и интерфейсов, обеспечивающая контроль доступа к классам и их элементам и контроль пространства имен. Пакет представляет собой элемент библиотеки – своеобразную «полку», содержимым которой вместо книг являются классы и интерфейсы. Связанность классов и

интерфейсов одного пакета определяется только нюансами доступа к классам и пространства имен.

Итак, пакет в Java – это не более чем коллекция сгруппированных вместе классов, которой присвоено некоторое имя. Все классы пакета размещаются в отдельных файлах, причем имя каждого файла совпадает с именем содержащегося в нем класса. Единственная новая деталь, отличающая обычный файл от пакетного, состоит в том , что первая строка каждого файла пакета должна иметь следующий вид:

package *Имя_пакета;*

например,

package mystuff.utilities;

Классы пакета хранятся в некотором каталоге, пакету присваивается имя, которое можно использовать затем в программах или классах. Любое приложение или определение класса может использовать все классы пакета, поместив соответствующий оператор import в начало файла, содержащего это приложение или это определение класса:

import mystuff.utilities;

Если классы хранятся в каталоге \myJava stuff\lib\math\stat, имя пакета будет иметь следующий вид

myJava stuff.lib.math.stat

Сборка файлов пакета в отдельном каталоге решает две другие проблемы: создания уникальных имен пакета, и поиска тех классов, которые могут быть скрыты где угодно в структуре каталогов. Это достигается с помощью указания пути к .class файлу в имени пакета, после ключевого слова package. Компилятор навязывает именно такую форму, но по соглашению, первая часть имени пакета зарезервирована - это доменное имя создателя класса в Интернет[3]. Поскольку доменные имена в Интернете гарантированно являются уникальными, то, следуя этому соглашению, Вы гарантированно получаете уникальные имена пакетов и никогда не получите конфликта имен. (Во всяком случае, пока Вы не отдадите это доменное имя кому-нибудь другому, кто начнет писать классы на Java с теми же именами путей, что и у Вас.) Конечно, если у Вас нет доменного имени, Вам придется придумать невероятную комбинацию (например, как Ваши имя и фамилия), для создания уникального имени пакета. Но если Вы решите опубликовывать код на Java , Вам стоит немного напрячься и получить собственное доменное имя.

Наиболее распространенное из соглашений об именах пакетов - это использование в качестве префикса имени пакета перевернутого имя домена организации в Интернет, например,

com.sun.games

Если имя пакета не указывается, класс становится частью безымянного пакета. Это вполне подходит для приложения (или аплета), которое используется отдельно от другого кода. Все классы, которые преднахначаются для использования в библиотеках, должны включаться в именованные пакеты.

Уровень доступа элемента языка является статическим свойством, задается на уровне кода и всегда проверяется во время компиляции. Попытка обратиться к закрытому элементу вызовет ошибку.

В Java модификаторы доступа указываются для:

- типов (классов и интерфейсов) объявления верхнего уровня;
- элементов ссылочных типов (полей, методов, внутренних типов);
- конструкторов классов.

Как следствие, например, массив, также может быть недоступен в том случае, если недоступен тип, на основе которого он объявлен.

Все четыре уровня доступа могут иметь только элементы типов и конструкторы. Это:

public

private

protected

если не указан ни один из этих трех типов, то уровень доступа определяется по умолчанию.

1. Открытый доступ (public) определяет, что к членам класса всегда можно обращаться из любого места, в котором доступен сам класс; такие члены наследуются в подклассах.
2. Закрытый (private): доступ к членам класса осуществляется только из самого класса. Если попытаться обратиться к private-данным или методам из другого класса, то компилятор Java выдаст сообщение об ошибке компиляции.
3. Защищенный (protected): к данным членам разрешается доступ из подклассов и из функций, входящих в тот же пакет, те наследникам может потребоваться доступ к некоторым элементам родителя, с которыми не приходится иметь дело внешним классам. Модификатор доступа protected позволяет обращаться к данным и методам класса лишь самому классу, классам, хранящимся в этом же пакете, и унаследованным классам. Обычно такой модификатор применяют для того, чтобы закрыть доступ к данным и методам для тех классов, которые не состоят в "родственных отношениях" с защищаемым классом. В Java классы считаются родственными, не только если они

унаследованы друг от друга, но и просто хранятся в одном и том же пакете. Например:

```
package Nums;
class First{
 protected int protVar;
 protected void protMethod() {
 System.out.println("protMeth called!");
 }
}
package Nums;
class Second {// не наследник First
 void protAccessMethod() {
  First ap = new First();
  ap.protVar = 345;
  ap.protMethod();
 }
}
```

Этот пример иллюстрирует доступ из класса того же пакета, мы используем в классе Second защищенное поле protVar и защищенный метод protMethod() класса First, оба класса принадлежат пакету Nums.

4. Пакетный:(package access) доступ к членам, объявленным без указания атрибута доступа, осуществляется только из того же пакета, где объявлен и сам этот класс. Более ограниченный по сравнению с protected, т.к. модификатор protected может быть указан для наследника из другого пакета, а доступ по умолчанию допускает обращения из классов-ненаследников, если они находятся в том же пакете.

Модификаторы доступа упорядочиваются следующим образом (от менее открытых – к более открытым):

1. private
2. none (package)
3. protected
4. public

Пакеты доступны всегда, поэтому у них нет модификаторов доступа (можно сказать, что все они public, то есть любой существующий в системе пакет может использоваться из любой точки программы).

Для типов (классы и интерфейсы) верхнего уровня объявления существуют всего две возможности: указать модификатор public или не указывать его. Если доступ к типу является public, то это означает, что он доступен из любой точки кода. Если же он не public, то уровень

доступа назначается по умолчанию: тип доступен только внутри того пакета, где он объявлен.

Массив имеет тот же уровень доступа, что и тип, на основе которого он объявлен (все примитивные типы являются полностью доступными).

Элементы и конструкторы объектных типов. Обладают всеми четырьмя возможными значениями уровня доступа. Все элементы интерфейсов являются public.

Следующая таблица иллюстрирует доступность элементов классов.

Таблица 6.5.1.Доступность переменной с данной комбинацией модификаторов (столбец) из указанного места (строка).

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

6.6. Описание и вызов методов

Методы – это действия, которые могут быть предприняты любым объектом класса. Вызов метода можно расценивать, как приказ объекту выполнить конкретное действие, используя конкретные данные, содержащиеся в этом объекте. Как уже отмечалось, иногда вместо термина вызвать метод объекта используется термин передать сообщение объекту.

Различают следующие два вида методов:

1. методы, возвращающие одно значение (функции в терминологии процедурных языков);
2. методы, выполняющие некоторое действие, но не возвращающие явно значения-результаты (процедуры).

Например, метод sqrt( ) класса Math возвращает одно значение типа double, а метод println - пример метода, выполняющего некоторое действие, эти методы называются void-методами.

Метод, определенный в классе, обычно вызывается с использованием объекта этого класса. Такой объект называется вызывающим объектом (calling object) и его имя используется при вызове метода:

Point p = new Point(); // p – объект класса Point

double r = p.distance (); // вызов метода объекта p

System.out.println ("r=" + r); // вызов метода println объектом out класса System

В некоторых специальных случаях (статические классы или методы) вместо имени объекта используется имя класса:

double d = Math.sqrt (a);

Общая форма объявления метода:

<модификатор> <тип> <имя_метода> (<список формальных параметров>) {

<тело метода>

}

Тип результата, который должен возвращать метод может быть любым, в том числе и типом void—в тех случаях, когда возвращать результат не требуется.

Например,

```
class Point {
  int x, y;
 void init (int a, int b) // заголовок описания метода init
 { //тело метода
  x = a;
  y = b;
  }
}
```

Заголовок в общем случае может включать:

- модификаторы (доступа в том числе);
- тип возвращаемого значения или ключевого слова void;
- имя метода;

- список аргументов (формальных параметров) в круглых скобках (аргументов может не быть);
- специальные throws-выражения.

Заголовок начинается с перечисления модификаторов. Для методов доступен любой из трех возможных модификаторов доступа. Также допускается использование доступа по умолчанию.

Кроме того, существует модификатор final, который говорит о том, что такой метод нельзя переопределять в наследниках. Можно считать, что все методы final-класса, а также все private-методы любого класса, являются final.

Аргументы метода перечисляются через запятую. Для каждого указывается сначала тип, затем имя параметра. В отличие от объявления переменной здесь запрещается указывать два имени для одного типа:

```
// void calc (double x, y); - ошибка!
void calc (double x, double y);
```

Если аргументы отсутствуют, указываются пустые круглые скобки. Одноименные параметры запрещены. Создание локальных переменных в методе, с именами, совпадающими с именами параметров, запрещено. Для каждого аргумента можно ввести ключевое слово final перед указанием его типа. В этом случае такой параметр не может менять своего значения в теле метода (то есть участвовать в операции присвоения в качестве левого операнда).

```
public void process(int x, final double y) {
  x=x*x+Math.sqrt(x);
 // y=Math.sin(x); – так писать нельзя,
 // т.к. y - final!
 }
```

Важным понятием является сигнатура (signature) метода[3]. Сигнатура определяется именем метода и его аргументами (количеством, типом, порядком следования). Если для полей запрещается совпадение имен, то для методов в классе запрещено создание двух методов с одинаковыми сигнатурами.

Например,

```
class Point {
  void get() {}
  void get(int x) {}
  void get(int x, double y) {}
  void get(double x, int y) {}
 }
```

Такой класс объявлен корректно.

Следующие пары методов в одном классе друг с другом несовместимы:

```
void get() {}
int get() {}
void get(int x) {}
void get(int y) {}
public int get() {}
private int get() {}
```

В первом случае методы отличаются типом возвращаемого значения, которое, однако, не входит в определение сигнатуры. Стало быть, это два метода с одинаковыми сигнатурами и они не могут одновременно появиться в объявлении тела класса.

Наконец, завершает заголовок метода throws-выражение. Оно применяется для корректной работы с ошибками в Java и будет подробно рассмотрено в разделе Обработка исключений.

Если формальные параметры имеют примитивные типы, то при замене формальных параметров на фактические используется механизм вызова по значению (call by value). Формальный параметр, используемый в определении метода, является локальной переменной, которая инициализируется значением фактического параметра, что защищает фактический параметр от возможных изменений при выполнении метода. Между числом и типом формальных и фактических параметров устанавливается взаимно-однозначное соответствие. В случае рассогласования по типу осуществляется автоматическое преобразование типов:

byte->short->int->long->float->double

Например:

```
public double distance(float x1, float y1, float x2, float y2) {
  float xdiff, ydiff;
  xdiff = x1 – x2;
  ydiff = y1 – y2;
  return Math.sqrt(xdiff^2 + ydiff^2);
}
int x3, x4, y3,y4;
double dist = distance (x3,y3,x4,y4);// параметры из типа int будут
//преобразованы в float …
```

Формальные параметры типа класса обрабатываются на основе другого механизма замены (call by reference). Формальный параметр типа класса – локальная переменная , в которой хранится адрес объекта этого класса.

При обращении к методу параметр инициализируется адресом соответствующего фактического параметра, задаваемого в вызове метода, т.е. формальный параметр служит в качестве альтернативного имени для объекта, заданного аргументом в вызове метода.

Следствие - метод способен менять значения переменных реализации аргумента, имеющего тип класса.

Предположим, используемый метод объявлен следующим образом:

```
public void process(int x) {
 x=5;
 }
```

Какое значение появится на консоли после выполнения

```
int x=3;
process(x);
print(x);
```

Параметр метода process(), хоть и имеет такое же имя x, на самом деле является полноценным хранилищем целочисленной величины. А потому присвоение ему значения 5 не скажется на внешних переменных. То есть результатом примера будет 3 и аргументы примитивного типа передаются в методы по значению. Единственный способ изменить такую переменную в результате работы метода – возвращать нужные величины из метода и использовать их при присвоении:

```
public int double(int x) {
  return x+x;
}
public void test() {
  int x=3;
  x=double(x);
}
```

Перейдем к ссылочным типам.

```
public void process(Point p) {
  p.x=3;
}
public void test() {
  Point p = new Point(1,2);//p.x = 1
   process(p);
   print(p.x);
}
```

Можно сказать, что ссылочные величины передаются по значению, но значением является именно ссылка на объект.

6.6.1. Метод main

Отдельно выделим метод main, который занимает в Java особое место. Виртуальная машина реализуется приложением операционной системы и запускается по обычным правилам. Программа, написанная на Java , является набором классов. Понятно, что требуется некая входная точка, с которой должно начинаться выполнение приложения.

Такой входной точкой, по аналогии с языками C/C++, является метод main(). Метод main() записывается как обычный метод, может содержать любые описания и действия, но он обязательно должен быть открытым (public), статическим (static), не иметь возвращаемого значения (void). Его аргументом обязательно должен быть массив строк (string[]). По традиции этот массив называют args , хотя имя может быть любым.

Пример его объявления:

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

О модификаторе static мы поговорим позже. Он позволяет вызвать метод main(), не создавая объектов. Метод не возвращает никакого значения, хотя в C есть возможность указать код возврата из программы. В Java для этой цели существует метод System.exit(), который закрывает виртуальную машину и имеет аргумент типа int.

Аргументом метода main() является массив строк. Он заполняется дополнительными параметрами, которые были указаны при вызове метода.

```
package test.first;
 public class Test {
 public static void main(String[] args) {
 for (int i=0; i<args.length; i++) {
 System.out.print(args[i]+" ");
 }
 System.out.println();
 }
}
```

Для вызова программы виртуальной машине передается в качестве параметра имя класса, у которого объявлен метод main(). Поскольку это имя класса, а не имя файла, то не должно указываться никакого расширения (.class или .Java) и расположение класса записывается через точку (разделитель имен пакетов), а не с помощью файлового разделителя. Компилятору же, напротив, передается имя и путь к файлу. Если приведенный выше модуль компиляции сохранен в файле

Test.Java , который лежит в каталоге test\first, то вызов компилятора записывается следующим образом:

Java c test\first\Test.Java

А вызов виртуальной машины:

Java test.first.Test

Чтобы проиллюстрировать работу с параметрами, изменим строку запуска приложения:

Java test.first.Test Hello, World!

Результатом работы программы будет:

Hello, World!

При вызове интерпретатора Java указывается класс, где записан метод main() , с которого надо начать выполнение. Поскольку классов с методом main() может быть несколько, можно построить приложение с дополнительными точками входа, начиная выполнение приложения в разных ситуациях из различных классов.
Часто метод main() заносят в каждый класс с целью отладки. В этом случае в метод main() включают тесты для проверки работы всех методов класса.

6.6.2. Параметр this

При определении метода в качестве имени вызывающего объекта можно использоваться ссылка на текущий объект (имя вызывающего объкта) - this. Оно будет заменено именем объекта, вызывающего метод:

```
public void move(double x, double y){
 this.x = x;//имена параметров скрывают
 this.y = y;//имена полей
 }
```

Естественно, обычно можно опускать фразу this (если нет параметров, совпадающих с переменными реализации), но в некоторых случаях, что и иллюстрирует вышеприведенный пример, она необходима, поскольку имена параметров могут скрывать имена полей - переменных реализации и вариант

x = x;

y= y;

явно не соответствует желаемым действиям.

6.6.3. Закрытые методы и переменные реализации

Методы класса могут быть объявлены с использованием модификатора private. В таком случае их нельзя вызывать за пределами определения класса, но можно вызывать в любом методе того же класса. Считается хорошим стилем программирования объявлять все переменные реализации классов с использованием модификатора private, вне определения класса эти переменные будут недоступны[5,7,18]. Внутри же любого метода этого определения класса имя private-переменной реализации можно использовать каким угодно образом, например, изменить значение переменной с этим именем.

Для доступа к таким переменным за пределами класса должны быть использованы специальные public методы[2,13]:

- аксессорный метод для чтения (accessor method) обычно с префиксом get…
- мутаторный метод для записи (mutator method) – префикс set…

т.е. сам программист планирует и управляет доступом к закрытым переменным. Например:

```
public class MyClass{
  private String name; //описание переменных реализации
  private int age;
  private String address;
 // описание методов для инициализации переменных реализации
  ......
// объявляем метод-мутатор
  public void setNewFields (String newName, int newAge, String
newAddress){
  name = newName;
  if (newAge >0)
  age = newAge;
  else {System.out.printin("ОШИБКА: использование
отрицательного возраста"); System.exit(0); }
  address = newAddress;
}
       // объявляем методы - аксессоры
  public String getName ( ) {
    return name;
  }
  public int getAge( ) {
     return age;
  }
  public String getAddress( ) {
```

```
    return address;
  }
}
```

Обращение к методам:

```
public class DemoClass {
… public static void main …{
MyClass cl1= new MyClass ( );
cl1.setNewFields (“А.Шварценеггер’’, 50,“Los Angeles,California”);
…
System.out.println (cl1.getName (),cl1.getAge(), cl1.getAddress() );
```

Закрытые методы и переменные реализации позволяют реализовать один из основных принципов ООП - инкапсуляцию (encapsulation). Это процесс сокрытия всех деталей определения класса, которые не являются необходимыми для понимания того, как используются объекты класса. Для того чтобы инкапсуляция была полезной, определение класса должно быть таким, чтобы программисту не нужно было беспокоиться насчет деталей определения класса. Инкапсуляция – это форма сокрытия информации. Выполненная корректно, инкапсуляция четко делит определение класса на две части, которые мы будем называть интерфейсом пользователя (или пользовательским интерфейсом) и разделом реализации. Несмотря на то, что программисту не обойтись без реализации класса для выполнения программы, которая использует этот класс, ему совершенно необязательно знать содержимое раздела реализации, чтобы написать эту программу. Определяя класс на основе принципов инкапсуляции, необходимо позаботиться о четком разделении частей интерфейса и реализации класса и представить интерфейс в виде простого и надежного описания класса. Для этого можно представить себе, что между интерфейсом и реализацией существует стена, оснащенная средствами коммуникации, находящимися под надлежащим контролем.

Если класс определен с использованием принципов инкапсуляции, и в его определении пользовательский интерфейс четко отделен от реализации, о таком классе можно сказать, что он хорошо инкапсулирован.

Сформулируем основные правила достижения хорошей инкапсулированности[2,15]:

- Перед определением класса поместить комментарий, как программист должен представлять себе данные и методы класса (в терминах предметной области)
- Все переменные реализации в классе должны объявляться с использованием модификатора private

•Предусмотреть public-методы аксессора и мутатора для чтения и изменения данных в объекте

•Предусмотреть другие public-методы, необходимые программисту для обработки данных в классе (методы ввода-вывода, например)

•Перед заголовком каждого public-метода поместите комментарий, как использовать данный метод

•Сделать все вспомогательные методы закрытыми

6.6.4. Статические методы и переменные

Иногда возникает потребность в методе, которому не нужно задавать какой-нибудь объект. Например, все математические методы не нуждаются в объектах. В таких случаях можно определить метод как статический (static). Статический метод определяется в классе, поэтому он является членом класса, но его можно вызвать, не используя никакого объекта.

В вызове статического метода вместо использования имени объекта обычно используется имя класса, в котором этот метод определен.

Статические методы иногда называются методами класса (class methods), в отличие от нестатических методов, называемых методами экземпляра (instance methods). они вызываются для целого класса, а не для каждого конкретного объекта, созданного на его основе. Например:

```
        /*Класс, содержащий статические методы для выполнения
вычислений, связанных с окружностями.*/
public class CircleFirstTry {
  public static final double PI = 3.14159;
  public static double area (double radius){
     return (PI*radius*radius) ;
  }
  public static double circumference(double radius){
    return (PI*(radius + radius));
  }
}
public class CircleDemo{
  public static void main(String[] args){
```

```
        double radius;
        System.out.println ("Введите радиус круга в см:");
        radius =System.In.readLineDouble();//имитация ввода
        System.out.println ("Круг радиусом "+ radius + " см");
        System.out.println ("имеет площадь» + CircleFirstTry.area(radius)
    + " квадратных см");
        System.out.println ("и длину окружности"
         +CircleFirstTry.circumference(radius) + " см.")
      }
    }
```

Очевидно, что использование статических классов накладывает определенные ограничения:

1. Если класс имеет переменные реализации, то к ним нельзя обращаться в определениях статических методов
2. В определении статического метода нельзя использовать переменную реализации или метод, который имеет явно или неявно заданный параметр this для вызывающего объекта.
3. Внутри определения любого статического метода нельзя вызывать нестатический метод (если не создать новый объект этого класса, а затем использовать этот объект в качестве вызывающего объекта для нестатического метода)

```
  /**************************************************
 Вызов нестатического метода внутри статического
 **************************************************/
public class PlayCircle {
  public static final double PI = 3.14159;
  private double diameter;//перем.реализации
  public void setDiameter(double newDiameter){
  diameter = newDiameter;
}
  public static double area(double radius){
    return (PI*radius*radius);
  }
  public void showArea(){
    System.out.println("Площадь равна " + area(diameter/2));
  }
  public static void areaDialog(){
    System.out.println("Введите диаметр круга:");
    double newDiameter = System.In.readLineDouble();
```

```
    PlayCircle c = new PlayCircle();
    c.setDiameter(newDiameter);
    c.showArea();
  }
}
public class PlayCircleDemo {
  public static void main (String [] args) {
    PlayCircle circle = new PlayCircle ();
    circle. setDiameter (2) ;
    System. out .println ("Если диаметр круга равен 2,");
    circle . showArea ( ) ;
    System.out.println("А теперь выберите сами значение диаметра.”);
    PlayCircle.areaDialog();
  }
}
```

Диалог развернется следующим образом:

```
Если диаметр круга равен 2,
Площадь равна 3.14159
А теперь выберите сами значение диаметра.
Введите диаметр круга:
4
Площадь равна 12.56636
```

В данном примере в классе PlayCircleDemo и в статическом методе мы обращаемся как к нестатическим методам класса PlayCircle-setDiameter и showArea (для чего создаем объект circle и c), так и к статическим - areaDialog, для вызова которого используется имя класса – PlayCircle.areaDialog().

Java требует, чтобы main-метод программы был статическим. Следовательно, внутри main-метода нельзя вызвать нестатический метод того же класса, не создав объект этого класса и не использовав его в качестве вызывающего объекта для этого нестатического метода.

Иногда возникает необходимость совместного использования переменной всеми объектами классаОни называются переменными класса, т.е. переменными относящимися ко всему классу, в отличие от переменных, относящихся к его отдельным объектам и описываются как static:

```
private static int numberOflnvocations = 0;
public static Point origin = new Point ( );
```

Подобно переменным реализации, статические переменные обычно объявляются закрытыми. Возможность чтения и изменения их значений

вне класса должна предоставляться только посредством методов доступа (акцессорных и мутаторных методов). Пример:

```
public class StaticDemo{
  private static int numberOfInvocations =0;
  public static void main(String[] args){
    int i;
    StaticDemo object1 = new StaticDemo();
    for (i = 1; i <=10 ; i++)
    object1.outPutCountOfInvocations();
    StaticDemo object2 = new StaticDemo();
    for (i = 1; i <=10 ; i++)
    object2.justADemoMethod();
    System.out.println("Общее количество вызовов = " +
    numberSoFar());
  }
  public void justADemoMethod() {
    numberOfInvocations++;….
  }
  public void outPutCountOfInvocations () {
    numberOf Invocations++;
    System.out.println(numberOflnvocations);
  }
  public static int numberSoFar () {
    numberOf Invocations++;
    return numberOflnvocations;
  }
}
```

В данном примере статическая переменная numberOfInvocations используется как счетчик числа вызовов методов, входящих в этот класс, исключая метод main.

Результат выполнения:

```
1
2
3
4
…
10
Общее кол-во вызовов = 21
```

Таким образом, для переменных класса выделяется только одна ячейка памяти, общая для всех экземпляров. Еще раз подчеркнем, что для нестатических переменных разные экземпляры одного класса

имеют совершенно независимые друг от друга поля, принимающие разные значения. Изменение поля в одном экземпляре никак не влияет на то же поле в другом экземпляре. В каждом экземпляре для таких полей выделяется своя ячейка памяти.

6.6.5. Перегрузка метода (overloading)

Наличие двух или больше определений методов под одним и тем же именем внутри одного и того же класса называется перегрузкой метода. Чтобы перегрузка успешно работала, необходимо предусмотреть, чтобы различные определения методов с одинаковыми именами имели какие-нибудь различные параметры,те различные сигнатуры.

Если в классе параметры перегруженных методов заметно различаются: например, у одного метода один параметр, у другого – два, то для Java это совершенно независимые методы и совпадение их имен может служить только для повышения наглядности работы класса. Каждый вызов, в зависимости от количества параметров, однозначно адресуется тому или иному методу. Рассмотрим следующий пример:

```
        /**Это класс для иллюстрации перегрузки.**
 ****************************************/

public class Statistician{
  public static void main(String[] args){
    double average1=Statistician.average(40.0, 50.0);
    double average2 = Statistician.average(1.0, 2.0, 3.0);
    char averages = Statistician.average('a', 'c');
  }
  public static double average (double first, double second){
    return ((first + second) /2.0) ;
  }
  public static double average (double first, double second, double
third){
    return ( (first + second + third) /3.0);
  }
  public static char average (char first, char second)
    return (char) (((int)first + (int)second)/2);
  }
}
```

Класс Statistician имеет три различных метода, причем все они называются average. Вызывая метод Statistician.average, Java как-то

"догадывается", какое именно определение метода average нужно использовать, но как? Сначала, допустим, что при вызове метода с именем average передаются все аргументы типа double. В этом случае Java может определить, какое использовать определение мет average, по количеству аргументов типа double. Если передано два аргумента double, значит, используется первое определение метода average. Если же три аргумента, используется второе определение метода average.

Теперь, предположим, что в программе встретился вызов метода average с двумя аргументами типа char. Благодаря типам переданных аргументов, Java "знает", что нужно использовать третье определение метода average. Ведь существует только одно определение метода average с двумя аргументами типа char.

Предположим, внутри одного и того же определения класса у нас есть несколько определений методов с одинаковыми именами. Когда в программе встречается вызов метода с этим именем для данного класса, Java по количеству аргументов и их типам определяет, какое определение нужно использовать. Если существует определение метода с этим именем, в котором количество параметров совпадает с количеством аргументов в вызове, и если их типы также совпадают, т.е. первый аргумент имеет такой же тип, как и первый параметр, второй аргумент имеет такой же тип, как и второй параметр, и т.д., то это определение метода и используется. Если в чем-то совпадения нет, Java пытается выполнить простое преобразование типов (в соответствии с порядком типов в списке, о котором шла речь выше), например тип int в тип double, чтобы понять, поможет ли такое приведение типов достигнуть совпадения. Если и это не помогает, вы получите сообщение об ошибке.

А теперь слегка отклонимся от темы, чтобы объяснить, как можно "усреднить" 2 буквы (метод char average). Для данного примера совершенно не имеет значения, какой способ (пусть кажущийся странным) мы используем для "усреднения" букв, но в действительности этот способ отнюдь не лишен смысла, особенно в случае, если мы находим среднее (вернее, среднюю) из двух букв. Если мы имеем дело с двумя строчными буквами, результате вычисления среднего значения мы получим строчную букву, расположенную посредине (в соответствии с алфавитным порядком) между двумя исходными буквами (Если мы не можем получить точного среднего "буквенного" значения, Java выберет ближайшее к среднему из двух возможных значений.) Аналогично, если заданы две прописные буквы, в результате вычисления среднего значения мы получим прописную букву, расположенную посредине (в соответствии с алфавитным

порядком) между, исходными буквами. Такой подход приемлем, поскольку буквам присваиваются последовательные номера. Номер, присвоенный букве ' b', на единицу больше номера, присвоенного букве 'a', а номер, присвоенный букве 'c', на единицу больше номера, присвоенного букве ' b', и т.д. Поэтому, если преобразовать две буквы в числа, вычислить среднее из двух чисел, а затем результат преобразовать снова в букву, мы получим букву, стоящую посредине между двумя исходными буквами.

Мы уже встречались с перегрузкой, явно не называя ее так. Например, метод max использует перегрузку на основе своих аргументов. Если два его аргумента имеют тип int, он возвращает значение int. Если два его аргумента имеют тип double, он возвращает значение типа double. Конечно, в данном случае использование перегрузки не слишком влияет на результат, поскольку различные определения метода max были бы идентичны, за исключением имен типов. Более интересен в смысле перегрузки пример операции деления(/). Если аргументы имеют тип double, используется деление с плавающей точкой, поэтому выражение 5.0/2.0 возвращает 2.5. Но если оба аргумента имеют тип int, выполняется операция целочисленного деления, и тогда выражение 5/2 возвращает целое число 2.

Итак, в пределах одного класса можно определить два (или больше) метода с одинаковыми именами. При перегрузке метода любые два определения методов с одинаковыми именами должны иметь либо различное количество параметров, либо параметры соответствующих позиций должны иметь различные типы. Java всегда старается использовать средство перегрузки до автоматического преобразования типов. Если Java в состоянии найти определение метода, в котором есть совпадение типов аргументов, то воспользуется именно этим определением, и автоматическое преобразование типов не используется до тех пор, пока Java не попытается найти определение метода, в заголовке которого типы параметров в точности совпадают с типами аргументов в вызове метода.

6.6.6. Конструкторы класса

Вы уже обратили внимание на то, что в операции new, определяющей экземпляры класса, повторяется имя класса со скобками. Это похоже на обращение к методу, но что за "метод", имя которого полностью совпадает с именем класса?

Такой "метод" называется конструктором класса (class constructor). Его своеобразие заключается не только в имени. Перечислим особенности конструктора[2,16,17].

1. Конструктор имеется в любом классе. Даже если вы его не написали, компилятор Java сам создаст конструктор по умолчанию (default constructor), который, впрочем, пуст, он не делает ничего, кроме вызова конструктора суперкласса.
2. Конструктор выполняется автоматически при создании экземпляра класса, после распределения памяти и обнуления полей, но до начала использования создаваемого объекта.
3. Конструктор не возвращает никакого значения. Поэтому в его описании не пишется даже слово void , но можно задать один из трех модификаторов public , protected или private.
4. Конструктор не является методом, он даже не считается членом класса. Поэтому его нельзя наследовать или переопределить в подклассе.

Тело конструктора может начинаться:

- с вызова одного из конструкторов суперкласса, для этого записывается слово super() с параметрами в скобках, если они нужны;
- с вызова другого конструктора того же класса, для этого записывается слово this() с параметрами в скобках, если они нужны.

Если же super() в начале конструктора не указан, то вначале выполняется конструктор суперкласса без аргументов, затем происходит инициализация полей значениями, указанными при их объявлении, а уж потом то, что записано в конструкторе.

Во всем остальном конструктор можно считать обычным методом, в нем разрешается записывать любые операторы, даже оператор return , но только пустой, без всякого возвращаемого значения.

В классе может быть несколько конструкторов. Поскольку у них одно и то же имя, совпадающее с именем класса, то они должны отличаться типом и/или количеством параметров.

Рассмотрим следующий пример:

```
/**************************************************
 Класс для сбора основных данных о домашних
 животных: имя, возраст и вес.
 **************************************************/
public class PetRecord {
 private String name;
```

```
 private int age; //в годах
 private double weight; //в кг
 public void writeOutput () {
 System. out. println ("Имя: " + name);
 System. out .println ("Возраст: " + age + " лет");
 System. out. println ("Вес: " + weight + " кг.")
 }
 public PetRecord (String initialName, int initialAge, double
initialWeight) {
 name = initialName;
 if ((initialAge < 0) || (initialWeight < 0))
 {System.out.println ("Ошибка: отрицательный возраст или вес.")
 System. exit (0) ;}
 else {age = initialAge; weight = initialWeight;}
 }
 public void set (String newName, int
 newAge, double newWeight){
 // тело как в конструкторе
 …
 }
 public PetRecord (String initialName) {
 name = initialName;
 age =0;
 weight =0;
 }
 public PetRecord (int initialAge) {
 name = "Пока без имени";
 weight = 0;
 if (initialAge < 0)
 {System.out.println("Ошибка: отрицательный возраст.");
 System.exit(0);}
 else age = initialAge;
 } ……
 public PetRecord ( ) {
 name = “Пока без имени”;
 age =0;
 weight =0;
 }
 }
 //вызовы конструкторов
 …public static void main…
```

```
PetRecord cow = new PetRecord (“Qween” , 10, 400);
…
PetRecord fish = new PetRecord(“Titanic”,2, 0,2);…
PetRecord newBorn= new PetRecord();
...
```

Данный пример иллюстрирует использование 4 конструкторов, их назначение ясно из контекста кода.

Кратко резюмируем рассмотренное.

Конструктор – это метод, который вызывается при создании объекта класса с помощью оператора new. Конструкторы используются для инициализации объектов. Имя конструктора должно совпадать с именем класса, которому он принадлежит.

Вызов конструктора, например, new PetRecord () возвращает ссылку на объект, т.е. он возвращает адрес памяти, выделенной объекту.

Зачем нужны конструкторы?

1. Некоторые классы не обладают разумным начальным состоянием, если не передать им параметры
2. При конструировании объектов некоторых видов передача исходного состояния оказывается самым удобным и разумным выходом
3. Конструирование объектов потенциально сопряжено с большими накладными расходами, так что желательно при создании объекта сразу устанавливать правильное исходное состояние. Например, если каждый объект класса содержит таблицу, то конструктор, получающий исходный размер таблицы в качестве параметра, позволит с самого начала создать объект с таблицей нужного размера.
4. Конструктор, атрибут доступа которого отличается от public, ограничивает возможности создания объектов данного класса.

Например, можно запретить программистам, работающим с вашим пакетом, расширять тот или иной класс, если сделать все конструкторы доступными лишь из пакета.
Кроме того, можно пометить ключевым словом protected те конструкторы, которые предназначены для использования исключительно в подклассах.

Если не объявлять для класса никаких конструкторов, Java создает безаргументный конструктор по умолчанию, который не делает ничего. Этот конструктор создается автоматически лишь в тех случаях, когда нет никаких других конструкторов

6.7. Наследование

Наследование (inheritance) - это отношение между классами, при котором класс использует структуру или поведение другого класса (одиночное наследование), или других (множественное наследование) классов. Наследование вводит иерархию "общее/частное", в которой подкласс наследует от одного или нескольких более общих суперклассов. Подклассы обычно дополняют или переопределяют унаследованную структуру и поведение.

Использование наследования способствует уменьшению количества кода, созданного для описания схожих сущностей, а также способствует написанию более эффективного и гибкого кода.

Множественное наследование на диаграмме изображается точно так же, как одиночное, за исключением того, что линии наследования соединяют класс-потомок сразу с несколькими суперклассами.

Не все объектно-ориентированные языки программирования содержат языковые конструкции для описания множественного наследования. В Java множественное наследование реализовано частично через интерфейсы - любой класс может реализовать несколько интерфейсов.

Понятие наследования классов – одно из ключевых в объектно-ориентированном программировании. В Java все создаваемые классы являются наследниками каких-то классов (которые называют базовыми или родительскими классами или классами-предками), так как в Java принята однокоренная иерархия наследования – есть один общий для всех классов класс-предок с именем Object, и если в создаваемом классе отсутствует явное указание родительского класса, то родительским классом станет класс Object.

Наследуемый класс также называют подклассом базового класса, производным классом или классом-потомком и говорят, что подкласс расширяет класс-предок (базовый класс), так как подкласс наследует от предка все открытые члены предка (поля и методы), и вдобавок к, таким, полученным «волшебным образом», возможностям может добавить какие-то дополнительные. В каком-то смысле базовый класс является подтипом производного класса, так как производный класс включает в себя интерфейс базового класса.

Давайте определим некоторый класс, на базе которого создадим подклассы.

```
public class Person {
 private String name;
```

```
public Person() {
name = "Пока без имени.";
}
public Person(String initialName) {
name = initialName;
}
public void setName(String newName){
name = newName;
}
public String getName( ){
return name;
}
public void writeOutput( ) { System.out.println("Имя: " + name);
}
public boolean sameName(Person otherPerson){
return (this.name.equalsIgnoreCase(otherPerson.name));
}
}
```

Данный класс описывает некоторого человека и содержит закрытое поле name, два конструктора, методы мутатора и акцессора, метод выводящий информацию о человеке и метод сравнивающий имена двух людей.

Производный класс (derived class) – это класс, определяемый путем добавления переменных реализации и методов в некоторый уже существующий класс. Существующий класс, на базе которого строится производный, называется, как мы уже говорили, базовым классом (base class). Производный класс имеет все переменные реализации и методы базового класса плюс все дополнительные переменные реализации и методы, которые программист считает нужным добавить.

Синтаксис описания производного класса:

```
public class Имя_Производного_Класса extends
Имя_Базового_Класса {
<Объявления_Добавляемых_Переменных_ Реализации>;
<Объявления_Добавляемых_И_Переопределяемых_Методов>;
}
```

Теперь определим на базе класса Person класс Student.

```
// Определение производного класса
public class Student extends Person {
private int studentNumber;
public Student ( ){
super () ;
```

```
studentNumber = 0;
// 0 - означает отсутствие номера.
}
public Student(String initialName, int initialStudentNumber) {
super(initialName);
studentNumber = initialStudentNumber;
}
public void reset (String newName, int newStudentNumber);{
setName (newName) ;
studentNumber = newStudentNumber;
}
public int getstudentNumber() {
return studentNumber;
}
public void setstudentNumber(int newStudentNumber) {
studentNumber = newStudentNumber;
}
public void writeOutput() {// переопределение метода баз.класса
System, out. println( "Имя: " + getName( );
System.out.println("Номер студента : " + studentNumber);
}
public boolean equals(Student otherStudent) {
return (this. sameName (otherStudent) && (this.studentNumber ==
otherStudent.studentNumber) )
}
}
    ….// пример использования класса Student
Student s1 = new Student ();
Student s2 = new Student(“Ringo Starr”, 5);
s1.setName (“Garry Potter”);
s1.setStudentNumber (20001);
if s1.equals (s2)
System.out.println (‘’ Один и тот же’’);…
```

Для определения производного класса необходимо описывать только дополнительные переменные реализации и дополнительные методы. Класс Student имеет все переменные реализации и все методы класса Person, хотя они и не упоминаются в определении класса Student. Каждый объект класса Student имеет переменную реализации name, но мы не определяем переменную реализации name в определении класса Student, т.е. класс Student наследует переменную реализации name и методы своего базового класса. Когда мы создали объект s1 класса

Student можно говорить о существовании переменной реализации s1.name. Но поскольку name - это закрытая переменная реализации, недопустимо напрямую устанавливать значение переменной s1.name вне определения класса (это было бы возможно, если бы мы объявляли ее с модификатором protected), но коль скоро эта переменная все-таки есть, то к ней можно получить доступ и изменить, используя акцессор и мутатор, соответственно (методы getName и setName).

В любой производный класс, подобный классу Student, можно добавить ряд переменных реализации и/или методов к тем, что он наследует от своего базового класса. Например класс Student, был расширен за счет добавления в него переменной реализации studentNumber, методов reset, getStudentNumber, setStudentNumber, wruteOutput и equals, а также нескольких конструкторов.

Теперь давайте поговорим о переопределении методов в производных классах. Если в производный класс включить определение метода с таким же именем и с таким же количеством параметров таких же типов, как в определении метода в базовом классе, то для производного класса это новое определение метода заменяет старое (writeOutput). В таких случаях тип значения, возвращаемого переопределяемым методом, должен совпадать с типом значения, возвращаемого методом в базовом классе, т.е., при переопределении метода нельзя менять тип значения, возвращаемого этим методом.

Отметим разницу между переопределением метода и перегрузкой. Когда речь идет о переопределении, новое определение метода, помещаемое в производный класс, имеет такое же количество и такие же типы параметров, как у старого метода в базовом классе. Но если метод в производном классе имеет другое количество параметров или параметры другого типа по сравнению с методом в базовом классе, то производный класс будет иметь оба метода. Такая ситуация позволяет говорить о перегрузке имени метода. Предположим, мы добавили в определение класса Student следующий метод:

```
public String getName (String title) {
 return (title + getName ())
 }
```

В этом случае класс Student будет иметь два метода с именем getName, унаследованный метод getName из базового класса Person и метод getName, который мы только что определили, т.е. эти два метода будут перегружены.

Если данное определение метода не должно заменяться новым определением в производном классе, достаточно добавить в заголовок метода модификатор final. Если метод объявляется с использованием

модификатора final, компилятор тем самым получает больше информации о характере его использования, что позволяет ему сгенерировать для данного метода более эффективный код. С помощью модификатора final можно объявить целый класс, и тогда его нельзя использовать в качестве базового класса для создания из него других (производных) классов.

Еще раз подчеркнем, к переменной реализации (или методу), которая закрыта в базовом классе, нельзя получить доступ по имени в определении метода любого другого класса, даже в определении метода производного класса: public void reset(String newName, int newStudentNumber) { name = newName;//НЕДОПУСТИМО // нужно - setName (newName) ;

6.7.1. Конструкторы в производных классах

Производный класс Student из предыдущего примера имеет собственные конструкторы. Базовый класс Person, из которого выведен класс Student, также имеет собственные конструкторы. При определении конструктора для производного класса, как правило, вызывается конструктор базового класса. Например, каково назначение конструктора класса Student? Вероятно, он должен инициализировать имя студента. Инициализация этого имени обычно выполняется конструкторами базового класса Person (поскольку переменная реализации name была введена в определении класса Person). Следовательно, вполне естественным действием для конструктора класса Student является вызов конструктора базового класса Person. Например, рассмотрим следующее определение конструктора производного класса Student:

```
public Student (String initialName, int initialStudentNumber){
 super( initialName);
 studentNumber = initialStudentNumber;
```

Строка super (initialName); представляет собой обращение к конструктору базового класса в данном случае к конструктору класса Person. Обратите внимание, что для вызова конструктора базового класса используется зарезервированное слово super, а не имя самого конструктора, т.е. мы не должны использовать вызов Person (initialName). Используя вызов super, необходимо выполнять некоторые требования. Вызов super должен всегда быть только первым действием, предпринимаемым в определение конструктора. Его нельзя использовать "потом". Если не включить вызов конструктора базового класса, Java автоматически включит обращение к конструктору

базового класса, действующему по умолчанию, в качестве первого действия любого конструктора производного класса.

Если мы в классе Student определим конструктор следующим образом:

```
public Student () {
 studentNumber = 0;//Признак отсутствия номера.
}
```

Java автоматически вставит обращение к конструктору базового класса, действующему по умолчанию –super ():

```
public Student () {
 super();
 studentNumber = 0;//Признак отсутствия номера.
 }
```

6.7.2. Метод this

При определении конструктора еще одним распространенным действием является обращение к одному из других конструкторов того же класса. В этом случае можно использовать зарезервированное слово this, подобно тому как мы использовали фразу super. Однако использование this обеспечивает вызов конструктора того же класса, а не конструктора базового класса. Например:

```
public Student (String initialName) {
 this (initialName, 0);
 }
```

Этот единственный оператор в теле конструктора представляет собой вызов конструктора, описание которого начинается следующим образом:

```
public Student (String initialName, int initialStudentNumber);
```

Как и в случае использования зарезервированного слова super, любое использование слова this должно быть первым действием в определении конструктора. Следовательно, определение конструктора не может содержать как вызов, использующий слово super, так и вызов, использующий слово this. Но как быть, если нужно иметь вызов с использованием слова super и вызов с использованием слова this? В этом случае используйте вызов, реализуемый словом this, и нужно позаботится о наличии конструктора, который вызывается с использованием слова this и имеет в качестве своего первого действия вызов super.

6.7.3. Обращение к переопределенному методу

Итак, определяя конструктор производного класса, в качестве имени конструктора базового класса можно использовать слово super. Слово super можно также использовать для вызова метода базового класса, который переопределяется в производном классе, но это делается несколько по-другому.

Рассмотрим, например, метод writeOutput класса Student. Для вывода имени студента используется следующая строка.

System.out.println ("Имя: " + getName());

В качестве альтернативного варианта можно вывести имя, вызвав метод writeOutput класса Person поскольку метод writeOutput класса Person предназначен для вывода имени человека. Единственная проблема состоит в том, что если просто использовать имя writeOutput в классе Student, оно будет означать метод writeOutput, определенный в классе Student. Нам же нужен способ указать, что мы вызываем метод writeOutput (), определенный в базовом классе. Этот способ состоит в использовании выражения super.writeOutput (). Поэтому альтернативное определение метода writeOutput для класса Student имеет следующий вид:

```
public void writeOutput() {
 super.writeOutput();
 System.out.println("Номер студента: " + studentNumber);
 }
```

Если заменить "старое" определение метода writeOutput в определении класса Student этим вариантом определения того же метода, поведение класса Student ничем не будет отличаться от поведения "старого" определения этого класса.

Итак, переопределенные методы позволяют Java поддерживать полиморфизм времени выполнения. Большое значение полиморфизма для объектно-ориентированного программирования обусловлено следующей причиной: он позволяет общему классу указывать методы, которые станут общими для всех его производных классов, в то же время позволяя подклассам определять конкретные реализации некоторых или всех этих методов. Переопределенные методы – еще один используемый в Java способ реализации аспекта полиморфизма под названием "один интерфейс, множество методов". Одно из основных условий успешного применения полиморфизма понимание Того, что суперклассы и подклассы образуют иерархию по степени увеличения специализации. В случае его правильного применения суперкласс предоставляет все элементы, которые подкласс может использовать непосредственно. Он определяет также те методы, которые производный класс должен реализовать самостоятельно. Это

позволяет подклассу определять собственные методы при сохранении единообразия интерфейса. Таким образом, объединяя наследование и переопределенные методы, суперкласс может определять общую форму методов, которые будут использоваться всеми его подклассами.

Динамический, реализуемый во время выполнения полиморфизм один из наиболее мощных механизмов объектно-ориентированной архитектуры, обеспечивающих повторное использование и надежность кода. Возможность существующих библиотек кода вызывать методы применительно к экземплярам новых классов без повторной компиляции при сохранении четкого абстрактного интерфейса чрезвычайно мощное средство.

6.7.4. Многоуровневые производные классы

Из любого производного класса можно создать еще один производный класс. Например, ниже мы определим класс Undergraduate, который является производным от класса Student. Это означает, что объект класса Undergraduate будет иметь все методы и переменные реализации класса Student. Но Student, в свою очередь, является производным классом от класса Person. Поэтому это также означает, что объект класса Undergraduate имеет все методы и переменные реализации класса Person. Объект класса Person имеет переменную реализации name. Объект класса Student имеет переменные реализации name и studentNumber. Объект класса Undergraduate имеет переменные реализации name, studentNumber и еще одну переменную реализации level. Однако для доступа и изменения переменных реализации name и studentNumber объект класса Undergraduate должен использовать акцессорные и мутаторные методы.

Определим класс Undergraduate как производный класс производного класса Student.

```
public class Undergraduate extends Student {
 private int level;//1 для студента первого курса,
 //2 для студента-второкурсника и т.д.
 public Undergraduate () {
 super();
 level = 1;
 }
 public Undergraduate (String initialName, int initialStudentNumber, int
initialLevel) {
 super(initialName, initialStudentNumber);
```

```
 level = initialLevel;
 }
 public void reset (String newName, int newStudentNumber, int newLevel) {
 reset(newName, newStudentNumber);
 level = newLevel;
 }
 public int getLevel() {
 return level;
 }
 public void setLevel (int newLevel) {
 level = newLevel;
 }
 public void writeOutput () {
 super.writeOutput();
 System.out.println("Уровень студента: " + level)
 }
 public boolean equals (Undergraduate otherUndergraduate) {
 return (super.equals(otherUndergraduate) && (this.level ==
otherUndergraduate.level));
 }
}
```

Заметьте, что такая цепочка производных классов может способствовать многократному использованию программного кода с высокой эффективностью. Оба класса Undergraduate и Student (а также любые другие классы, выведенные из них), по сути, повторно используют код методов, заключенный в определении класса Person, поскольку они наследуют все методы класса Person.

Обратите внимание на конструкторы класса Undergraduate. Все они начинаются с вызова super, который в этом контексте заменяет вызов конструктора базового класса Student. Но конструкторы базового класса Student также начинаются с вызова super, который в этом случае означает вызов конструктора базового класса Person. Следовательно, при вызове конструктора класса Undergraduate (с помощью оператора new) сначала вызывается конструктор класса Person, затем конструктор класса Student, а затем выполняются все операторы, следующие за вызовом super в конструкторе класса Undergraduate.

Рассмотрим следующее определение метода reset в классе Undergraduate.

```
public void reset (String newName, int newStudentNumber, int newLevel) {
 reset (newName, newStudentNumber); )
```

```
 level = newLevel;
}
```

Обратите внимание, что этот метод начинается с вызова еще одного метода reset, принимающего только два аргумента. Это – вызов метода reset, который определен в базовом классе Student. В классе Undergraduate метод reset перегружен, т.е. в классе Undergraduate существует два определения метода с именем reset. Один принимает два аргумента, а другой – три. Тот, который принимает два аргумента, унаследован от класса Student, тем не менее, он вполне "полноправный" метод класса Undergraduate.

В версии метода reset с тремя аргументами (который определен в самом классе Undergraduate и воспроизведен выше) посредством следующего вызова метод reset базового класса Student устанавливает переменные реализации name и studentNumber (равными значениям newName и newStudentNumber, соответственно), reset (newName, newStudentNumber);

Затем при выполнении оператора присваивания новая переменная реализации level устанавливается равной значению переменной newLevel.

Вспомните, что внутри определения класса Undergraduate к private-переменным реализации name и studentNumber (базовых классов) нельзя обращаться по имени, поэтому для изменения их значений необходимо использовать мутаторный метод. Метод reset класса Student вполне подходит для этой цели.

Сравните с определением метода reset в классе Undergraduate определение метода writeOutput, которое приводится ниже.

```
public void writeOutput () {
  super.writeOutput ();
  System.out.println ("Уровень студента: " + level);
}
```

Определение метода reset послужило для нас примером перегрузки. Метод writeOutput – это пример переопределения.

При использовании одного и того же имени метода reset версия, определенная в классе Undergraduate, отличается от той, что определена в классе Student, количеством параметров. Поэтому в производном классе Undergraduate не возникает никого конфликта с наличием обоих версий метода reset. Это пример перегрузки, отличие от метода reset, метод writeOutput, определенный в производном классе Undergraduate, имеет точно такой же список параметров, как и версия, определенная в базовом классе Student. Следовательно, при вызове метода writeOutput Java должна решить, какое именно определение метода writeOutput

использовать. Для объекта производного класса Undergraduate используется версия writeOutput, содержащаяся в определении класса Undergraduate.

Это пример переопределения метода (writeOutput) в производном классе. Однако определение метода writeOutput, которое содержится в базовом классе Student, по-прежнему наследуется производным классом Undergraduate. Но, чтобы вызвать версию, определенную в базовом классе Student, внутри определения производного класса Undergraduate, необходимо перед именем метода writeOutput поместить префикс super и точку, как в первой строке определения метода writeOutput, содержащегося в определении производного класса Undergraduate и приведенного в предыдущем абзаце.

Заметьте, что абсолютно допустимо (и общеупотребимо) переопределять в производных классах такие имена методов, как writeOutput и equals. Версии базового класса никуда не исчезают, и к ним можно легко получить доступ, используя префикс super и точку.

При рассмотрении производных классов приинято использовать терминологию, "взятую" из сферы семейных отношений. Базовый класс часто называется родительским классом (parent class), а производный – дочерним классом (child class). Это придает языку, используемому при обсуждении темы наследования, определенную "гладкость". Например, мы можем сказать, что дочерний класс наследует переменные реализации и методы от своего родительского класса. Эту аналогию можно проследить и дальше. Класс, который является родителем родителя еще одного класса (можно использовать любое другое число повторений слова "родителя") часто называется классом-предком (ancestor class). Если класс A является предком класса B, то класс в часто называется потомком (descendant) класса A.

Рассмотрим методы с именем equals в классах Student и Undergraduate. Они имеют различные списки параметров. Метод в классе Student имеет параметр типа Student, в то время как метод в классе Undergraduate имеет параметр типа Undergraduate. Эти методы принимают по одному параметру, т.е. налицо одинаковое количество параметров, но эти параметры в разных определениях имеют разные типы. Различия в типе вполне достаточно для квалификации наличия перегрузки. Для перегрузки имени двух методов в определениях этих методов должно быть разное количество параметров или параметры соответствующих позиций должны иметь различные типы. Поэтому второе определение метода equals, данное в производном классе Undergraduate, создает перегрузку, а не переопределение.

Тогда почему мы использовали префикс super в определении метода equals, содержащемся в производном классе Undergraduate? Чтобы облегчить анализ ситуации воспроизведем это определение в следующих строках (оно приведено в определении производного класса Undergraduate).

```
boolean equals (Undergraduate otherUndergraduate) {
 return ( super.equals (otherUndergraduate) && (this.level ==
otherUndergraduate.level) );
 }
```

Поскольку вызов super, equals – это вызов перегруженной версии метода equals, почему нужно добавлять префикс super? Чтобы понять почему, попробуем опустить его, как показано ниже.

```
return (equals (otherUndergraduate) && (this.level ==
otherUndergraduate.level) );
```

Аргумент otherUndergraduate имеет тип Undergraduate, поэтому Java предполагает, что он относится к определению метода equals, данному в классе Undergraduate. Чтобы заставить Java использовать определение метода equals из базового класса Student, нам пришлось использовать префикс super и точку.

Итак, внутри определения метода производного класса можно вызвать перегруженный метод базового класса, предварив его имя префиксом super и точкой. Однако нельзя повторно использовать super для вызова метода из какого-нибудь класса-предка, если он не является прямым родителем. Предположим, что класс Student выведен из класса Person, а класс Undergraduate выведен из класса Student. Если вы подумали, что можно вызвать метод класса Person внутри определения класса undergraduate, используя вызов super.super, как показано в следующем выражении, то это предположение ошибочно.

```
 super.super. writeOutput ( ) ; / /НЕВЕРНО!
```

Итак, стоит запомнить, что в языке Java "сцепление" вызовов super.super не работает, поэтому вызов версии метода writeOutput, определенного в классе Person, с помощью предыдущего оператора невозможен.

Отметим также, объект производного класса имеет несколько типов. Класс Undergraduate является производным от класса Student. В реальной жизни каждый первокурсник одновременно является и студентом. Эта взаимосвязь сохраняется и в языке Java . Каждый объект класса Undergraduate является также объектом класса Student. Следовательно, если у вас есть метод, который имеет формальный параметр типа Student, то аргументом в вызове этого метода может быть объект типа Undergraduate. В этом случае данный метод мог бы

использовать только те переменные реализации и методы, которые принадлежат классу student, но каждый объект класса Undergraduate имеет все те же переменные реализации и методы, поэтому рассматриваемый метод вполне может в качестве аргумента принять и обработать объект типа Undergraduate.

Таким образом, в результате наследования объект реально может иметь больше двух типов. Вспомните, что ' класс Undergraduate является производным от класса Student, а класс Student – производный от класса Person. Это значит, что каждый объект класса Undergraduate является также объектом типа Student и объектом типа Person. Следовательно, все, что работает для объектов класса Person, работает и для объектов класса Undergraduate.

В языке Java предусмотрен класс "Eve" (канун), т.е. класс, который является предком каждого класса. В Java каждый класс является производным от производного класса от ... (возможен целый ряд повторений фразы "производного класса от") класса Object. Поэтому каждый объект каждого класса имеет тип Object, а также тип своего класса (а также тип любого другого класса-предка). Даже классы, которые вы определяете сами, являются классами-потомками класса Object. Если вы не сделали свой класс производным от некоторого другого класса, то Java действует так, как если бы вы сделали его производным от класса Object.

Мы уже отмечали, что в языке Java каждый класс является потомком встроенного класса Object. Поэтому каждый объект каждого класса имеет тип Object, а также тип своего класса (а также тип любого другого класса-предка). Поскольку объект производного класса имеет тип всех своих классов-предков (также как свой "собственный" тип), то объект этого класса можно присвоить переменной любого типа предка, но не наоборот. Например, если класс Student – производный от класса Person, а класс Undergraduate – производный от класса Student, то следующие операторы вполне допустимы.

```
Person p1, p2;
p1 = new Student () ;
p2 = new Undergraduate ();
```

Но все операторы из следующей группы недопустимы.

```
Student s = new Person (); //НЕВЕРНО!
Undergraduate ug = new Person (); //НЕВЕРНО!
Undergraduate ug2 = new Student'(); //НЕВЕРНО!
```

Во всем этом есть здравый смысл. Например, любой студент (объект типа Student) – это человек (объект типа Person), но любой человек (объект типа Person) – необязательно студент (объект типа

Student). Подобные рассуждения весьма полезны для программистов при принятии решения о том, какие типы может иметь объект и какие-либо присваивания переменным допустимы. Объект производного класса можно присвоить переменной любого типа предка, но не наоборот.

Поскольку каждый класс в языке Java – потомок класса Object, должны существовать методы, которые каждый класс наследует от класса Object. И действительно, такие методы существуют, более того, это очень полезные методы. Например, каждый объект наследует методы equals и toString от некоторого класса-предка, которым либо является класс Object, либо этот класс-предок сам, в конечном счете, унаследовал эти методы от класса Object.

Однако унаследованные методы equals и toString не будут работать правильно (почти) для любого определяемого вами класса. Поэтому вам нужно переопределить эти унаследованные методы, используя новые более подходящие определения. Следовательно, определяя метод equals для любого класса, вы, с формальной точки зрения, переопределяете его.

Унаследованный метод toString не принимает никаких аргументов. Метод toString должен возвратить все данные, содержащиеся в объекте, преобразованным в тип String. Однако автоматически вы не получите подходящего строкового представления своих данных. Унаследованная версия метода toString практически всегда бесполезна в своем исходном варианте. Поэтому вам следует переопределить метод toString, чтобы он возвращал соответствующее String-представление для данных, содержащихся в объектах определяемого вами класса.

Например, следующее определение метода toString можно добавить в класс Student:

```
public String toStrir-g(){
 return("Имя: "'+ getName() + "\nНомер студента: "+
Integer.toString(studentNumber) );
 }
```

Если бы этот метод toString был добавлен в класс Student, его можно было бы использовать для вывода информации следующим способом.

```
Student s = new Student ("Студент Джо", 2001);
System. out.println (s.toString ( ) ) ;
```

Результат выполнения этих строк имел бы следующий вид.

```
Имя: Студент Джо
Номер студента: 2001
```

Еще одним методом, наследуемым от класса Object, является метод clone. Этот метод не принимает аргументов и возвращает копию вызывающего объекта. Предполагается, что возвращаемый объект содержит данные, идентичные данным вызывающего объекта, но все же это другой объект (двойник, или "клон"). Как и в случае других методов наследуемых от класса Object, метод clone (если вы ждете от него надлежащего функционирования) нуждается в переопределении.

6.8. Вложенные и внутренние классы

Java позволяет определять класс внутри другого класса. Такие классы называют вложенными классами. Область определения вложенного класса ограничена областью определения внешнего класса. Таким образом, если класс B определен внутри класса A, класс B не может существовать независимо от класса.A. Вложенный класс имеет доступ к членам, в том числе закрытым (private), класса, в который он вложен. Однако внешний класс не имеет доступ к членам вложенного класса. Вложенный класс, который объявлен непосредственно внутри области определения своего внешнего класса, является его членом.

Можно также объявлять вложенные классы, являющиеся локальными для блока.

Существует два типа вложенных классов: статические и нестатические. Статический вложенный класс, к которому применен модификатор static. Поскольку он является статическим, он должен обращаться к своему внешнему классу посредством объекта. То есть он не может непосредственно ссылаться на члены своего внешнего класса.

Из-за этого ограничения статические вложенные классы используются редко. Наиболее важный тип вложенного класса – внутренний класс. Внутренний класс это нестатический вложенный класс. Он имеет доступ ко всем переменным и методам своего внешнего класса и может непосредственно ссылаться на них так же, как это делают остальные нестатические члены внешнего класса.

Следующая программа иллюстрирует определение и использование внутреннего класса. Класс Outer содержит одну переменную экземпляра outer x, один метод экземпляра test () и определяет один внутренний класс Inner.

```
// Демонстрация использования внутреннего класса.
class Outer {
 int outer_x = 100;
 void test () {
  Inner inner = new Inner();
```

```
  inner.display();
}
// это внутренний класс
class Inner {
 void display(){
 System.out.printlп("вывод: outer x = " + outer_x);
 }
 }
}
class InnerClassDemo {
 public static void main(String args[]) {
 Outer outer = new Outer();
 outer.test();
 }
}
```

Это приложение генерирует следующий вывод:

```
вывод: outer_x = 100
```

В этой программе внутренний класс Inner определен в области определения класса Outer. Поэтому любой кода в классе Inner может непосредственно обращаться к переменной outer x. Метод экземпляра display () определен внутри класса Inner. Этот метод отображает значение переменной outer_x в стандартном выходном потоке. Метод main () экземпляра InnerC1assDemo создает экземпляр класса Outer и вызывает его метод test (). Этот метод создает экземпляр класса Inner и вызывает метод display (). Важно понимать, что экземпляр класса Inner может быть создан только внутри области определения класса Outer. Компилятор Java сгенерирует сообщение об ошибке, если любой код вне класса Outer пытается инициализировать класс Inner. (В общем случае экземпляр внутреннего класса должен создаваться содержащим его диапазоном.). Однако экземпляр класса Inner можно создать снаружи класса Outer, уточняя имя внутреннего класса именем внешнего класса, например Outer. Inner.

Как уже было сказано, внутренний класс имеет доступ ко всем элементам своего внешнего класса, но не наоборот. Члены внутреннего класса известны только внутри области определения внутреннего класса и не могут быть использованы внешним классом.

Например:

```
// Компиляция этой программы будет не возможна.
class Outer {
 int outer
 x = 100;
```

```
 void test () {
 Inner inner = new Inner ();
 inner.display ();
 }
// это внутренний класс
class Inner {
 int y = 10;
// y - локальная переменная класса Inner
 void display () {
 System.out.printlп("вывод: outer_x = " + outer_x);
 }
}
 void showy () {
 System.out.println(y); // ошибка; здесь переменная у не известна!
 }
}
class InnerClassDemo (
 public static void main(String args[]) {
 Outer outer
 new Outer ();
 outer.test ();
 }
}
```

В этом примере переменная у объявлена как переменная экземпляра класса Inner. Поэтому она не известна за пределами класса и не может использоваться методом showy().

Хотя мы уделили основное внимание внутренним классам, определенным в качестве членов внутри области определения внешнего класса, внутренние классы можно определять внутри области определения любого блока. Например, вложенный класс можно определить внутри блока, определенного методом, или даже внутри тела цикла for, как показано в следующем примере:

```
 // Определение внутреннего класса внутри цикла for.
class Outer {
 int outer_x =100;
 void test () {
  for(int i=0; i<10; i++) {
 class Inner {
  void display () {
   System. out. println ("вывод: outer_x " + outer_x) ;
   }
```

```
  }
  Inner inner = new Inner ();
  inner.display ();
  }
 }
}
class InnerClassDemo (
 public static void main (String args[]) (
 Outer outer = new Outer ();
 outer.test ();
 }
}
```

Вывод, генерируемый этой версией программы, показан ниже.

```
вывод: outer x = 100
вывод: outer x = 100
вывод: outer x = 100
вывод: outer x = 100
вывод: outer x = 100
вывод: outer x = 100
вывод: outer x = 100
вывод: outer x = 100
вывод: outer x = 100
вывод: outer x = 100
```

Хотя вложенные классы применимы не во всех ситуациях, они особенно удобны при обработке событий, когда представлены внутренние классы, которые можно использовать для упрощения кода, предназначенного для обработки определенных типов событий.

6.9. Реализация абстрактных методов абстрактных классов

В ряде ситуаций нужно будет определять суперкласс, который объявляет структуру определенной абстракции без предоставления полной реализации каждого метода. То есть тогда придется создавать суперкласс, определяющий только обобщенную форму, которую будут совместно использовать все его подклассы, добавляя необходимые детали. Такой класс определяет сущность методов, которые должны реализовать подклассы.

Например, такая ситуация может возникать,. когда суперкласс не в состоянии создать полноценную реализацию метода.

Как вы убедитесь в процессе создания собственных библиотек классов, отсутствие полного определения метода в контексте суперкласса не столь уж редкая ситуация.

Могут существовать методы, которые должны быть переопределены подклассом, чтобы подкласс имел какой либо смысл. Рассмотрим класс Triang1e. Он лишен всякого смысла, если метод area () не определен. В этом случае необходим способ убедиться в том, что подкласс действительно переопределяет все необходимые методы. В Java для этого служит абстрактный метод.

Потребовать, чтобы определенные методы переопределялись подклассом, можно посредством указания модификатора типа abstract. Иногда такие методы называют относящимися к компетенции подкласса, поскольку в супер-классе для них никакой реализации не предусмотрено. Таким образом, подкласс должен переопределять эти методы он не может просто использовать версию, определенную в супер-классе. Для объявления абстрактного метода используют следующую общую форму:

abstract тип имя(список параметров);

Как видите, в этой форме тело метода отсутствует.

Любой класс, который содержит один или более абстрактных методов, должен быть также объявлен как абстрактный. Для этого достаточно поместить ключевое слово abstract перед ключевым словом c1ass в начале объявления класса. Абстрактный класс не может содержать какието объекты. То есть абстрактный класс не может быть непосредственно конкретизирован с помощью операции new. Такие объекты были бы бесполезны, поскольку абстрактный класс определен не полностью. Нельзя также объявлять абстрактные конструкторы или абстрактные статические методы. Любой подкласс абстрактного класса должен либо реализовать все абстрактные методы суперкласса, либо также быть объявлен абстрактным.

Ниже приведен простой пример класса, содержащего абстрактный метод, и класса, который реализует этот метод.

```
// Простой пример применения абстракции.
abstract class A {
 abstract void callme ();
// абстрактные классы Все же могут содержать конкретные методы
 void callmetoo () {
  System.out.println ("Это конкретный метод.");
  }
```

```
}
class B extends A {
 void callme () {
  System.out.printlп("Реализация метода callme класса B.");
 }
}
class AbstractDemo {
 public static void main (String args[ ]){
  B b = new B () ;
  b.callme () ;
 b.callmetoo();
 }
}
```

Обратите внимание, что в этой программе класс A не содержит объявлений каких либо объектов. Как уже было сказано, конкретизация абстрактного класса невозможна. И еще один нюанс: класс A реализует конкретный метод callmetoo () . Это вполне допустимо. Абстрактные классы. Могут содержать любое необходимое количество конкретных реализаций. Хотя абстрактные классы не МогуТ быть использованы для конкретизации объектов, их можно применять для создания ссылок на объекты, поскольку в Java полиморфизм времени выполнения реализован посредством ссылок на суперкласс. Поэтому должна существовать возможность создания ссылки на абстрактный класс, которая может использоваться для указания на объект подкласса. Применение этого свойства показано в следующем примере.

Поскольку понятие площади неприменимо к неопределенной двумерной фигуре, следующая версия программы объявляет метод area () внутри класса Figure как abstract. Конечно, это означает, что все классы, производные от Figure, должны переопределять метод area ( ) .

```
// Использование абстрактных методов и классов.
abstract class Figure {
 double dim1;
 double dim2;
 Figure (double a, double b) {
 double dim1= a;
 double dim2 =b;
 };
// теперь метод area является абстрактным
 abstract double area ();
```

```
}
class Rectangle extends Figure {
 Rectangle (double a, double b) {
  super (a, b);
}
// переопределение метода area для четырехугольника
 double area () {
  System.out.println ("В области четырехугольника.");
  return dim1 * dim2;
 }
}
class Triangle extends Figure {
 Triangle (double a, double b){
  super (a, b);
 }
// переопределение метода area для треугольника
 double area () {
  System.out.println("В области треугольника.");
  return dim1 * dim2 / 2;
 }
}
class AbstractAreas {
 public static void main (String args[]) {
// Figure f = new Figure(10, 10); // недопустимо
  Rectangle r = new Rectangle (9, 5);
  Triangle t= new Triangle (10, 8);
  Figure figref; // этот оператор допустим, никакой объект не
//создается
  figref = r;
  System.out.println ("Площадь равна" + figref.area());
  figref = t;
  System.out.println ("Площадь равна" + figref.area());
 }
}
```

Как видно из комментария внутри метода main () , объявление объектов типа Figure недопустимо, поскольку этот класс является абстрактным. И все подклассы класса Figure должны переопределять метод area () . Чтобы убедиться в этом, попытайтесь создать подкласс, который не переопределяет метод area () . Это приведет к ошибке времени компиляции.

Хотя создание объекта типа Figure недопустимо, можно создать ссылочную переменную типа Figure. Переменная figref объявлена как ссылка на Figure, т.е. ее можно использовать для ссылки на объект любого класса, производного от Figure. Как мы уже поясняли, разрешение переопределенных методов во время выполнения осуществляется путем ссылки на суперкласс.

7. ОБРАБОТКА ИСКЛЮЧЕНИЙ

В этой главе рассматривается механизм обработки исключений Java. Исключение – это ненормальная ситуация, возникающая во время выполнения последовательности кода. Другими словами, исключение – это ошибка времени выполнения. В языках программирования, которые не поддерживают обработки исключений, ошибки должны проверяться и обрабатываться "вручную" обычно, путем использования кодов ошибок и тому подобного. Этот подход как обременителен, так и чреват проблемами. Обработка исключений Java позволяет избежать этих проблем, а, кроме Того, переносит управление ошибками времени выполнения в объектно-ориентированный мир.

7.1. Основы обработки исключений

Исключение Java представляет собой объект, который описывает исключительную (то есть, ошибочную) ситуацию, возникающую в части программного кода. Когда такая исключительная ситуация возникает, создается объект, представляющий исключение, который возбуждается в методе, вызвавшем ошибку. Этот метод может либо обработать исключение самостоятельно, либо пропустить его. В обоих случаях, в некоторой точке исключение перехватывается и обрабатывается. Исключения могут генерироваться системой времени выполнения Java, либо они могут быть сгенерированы вручную вашим кодом. Исключения, которые возбуждает Java, имеют отношение к фундаментальным ошибкам, которые

нарушают правила языка Java либо ограничения системы выполнения Java. Вручную сгенерированные исключения обычно применяются для того, чтобы сообщить о некоторых ошибочных ситуациях тому, кто вызвал данный метод. Обработка исключений Java управляется пятью ключевыми словами: try, catch, throw, throws и finally. Если кратко, они работают следующим образом. Операторы программы, которые вы хотите отслеживать на предмет исключений, помещаются в блок try. Если исключение возникает в блоке try, оно возбуждается. Ваш код может перехватить исключение (используя catch) и обработать его некоторым осмысленным способом.

Типы ошибок, возникающих при выполнении приложения можно разделить на следующие:

- ошибки ввода
- сбои оборудования
- физические ограничения
- ошибки программирования

В соответствии с этой классификацией в Java создана иерархия классов-исключений. В вершине иерархии исключений находится класс Throwable (от throw able – "способный возбудить исключительную ситуацию"). Все классы-исключения расширяют класс Throwable – непосредственное расширение класса Object. Фрагмент этой классификации приведен на рис. 7.1[3].

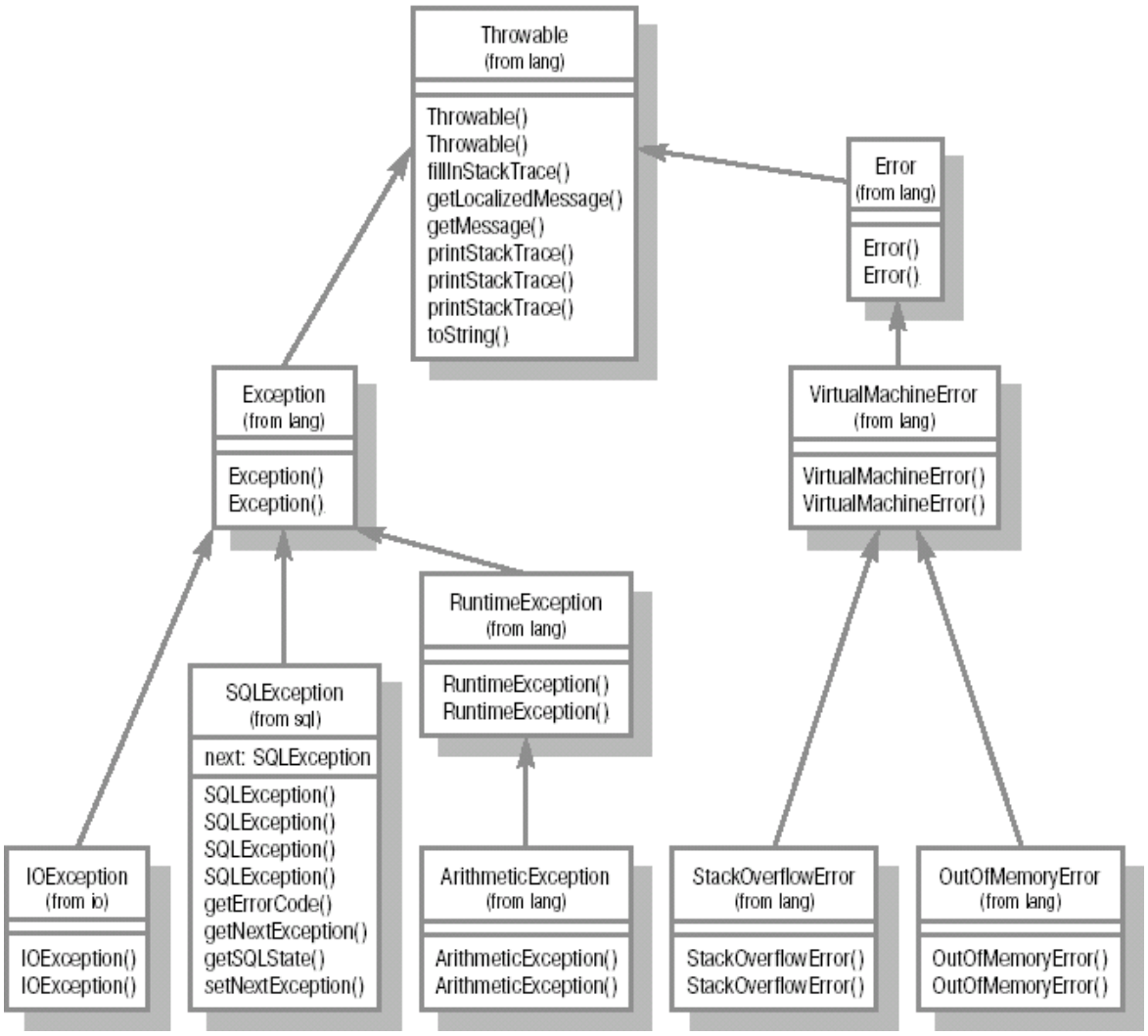

Рис.7.1

У класса Throwable и у всех его расширений, как правило, два конструктора:

- Throwable () – конструктор по умолчанию;
- Throwable (String message) – создаваемый объект будет содержать произвольное сообщение message.

Это сообщение можно получить методом public String getMessage ().

Если объект создавался конструктором по умолчанию, то данный метод возвратит null.

В JVM имеется специальный стек, в котором сохраняются имена всех вызываемых методов, в порядке обращения к ним.

Трассировка стека – это список вызовов методов для данной точки программы.

Три метода выводят сообщения обо всех методах, встретившихся по пути "полета" исключения:

public void printstackTrace() — выводит сообщения в стандартный вывод, как правило, это консоль;

public void printStackTrace(PrintStream stream) — выводит сообщения в байтовый поток Stream;

public void printStackTrace(PrintWriter stream) — выводит сообщения в символьный поток Stream.

В основном программисты работают с классом Exception и его расширениями, класс Error и его наследники предназначены для обработки системных ошибок и программистами не обрабатывается.

Ошибки, порожденные от Exception (и не являющиеся наследниками RuntimeException), являются проверяемыми – т.е. во время компиляции проверяется, предусмотрена ли обработка возможных исключительных ситуаций. Это, например, классы

- ClassNotFoundException
- IllegalAccessException
- InstantiationException
- NoSuchMethodException
- NoSuchFieldException и др.

Исключения, порожденные от RuntimeException и его подклассы и Error и его подклассы, являются непроверяемыми и компилятор не требует обязательной их обработки –

- ArithmeticException
- ArrayStoreException
- ClassCastException
- IllegalArgumentException
- NumberFormatException
- IndexOutOfBoundsException
- NegativeArraySizeException
- NullPointerException и др.

Исключение в Java – это объект, который описывает исключительное состояние, возникшее в каком-либо участке программного кода. Когда возникает исключительное состояние, системой создается объект класса Throwable. Этот объект пересылается в код, обрабатывающий данный тип исключительной ситуации. Исключения могут возбуждаться и «вручную» (throw) для того, чтобы сообщить о некоторых нештатных ситуациях. Базовый механизм обработки исключительной ситуации реализуется триадой блоков try throw-catch или диадой try-catch

Если исключение не генерируется, то catch блоки игнорируются.

Общая форма блока обработки исключений.

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

Сначала выполняется код, заключенный в фигурные скобки оператора try. Если во время его выполнения не происходит никаких нештатных ситуаций, то далее управление передается за закрывающую фигурную скобку последнего оператора catch, ассоциированного с данным оператором try. Если в пределах try возникает исключительная ситуация, то далее выполнение кода производится по одному из нижеперечисленных сценариев:

1. производится выполнение блока кода, ассоциированного с данным catch

2. если код в этом блоке завершается нормально, то и весь оператор try завершается нормально
3. если код в catch завершается нештатно, то и весь try завершается нештатно по той же причине
4. если возникла исключительная ситуация, класс которой не указан в качестве аргумента ни в одном catch, то выполнение всего try завершается нештатно
5. оператор finally предназначен для того, чтобы обеспечить гарантированное выполнение какого-либо фрагмента кода, вне зависимости от того, возникла ли исключительная ситуация
6. если блок finally завершается ненормально, то весь try завершится ненормально по той же причине.

Программист может создавать свои собственные классы исключений, на базе стандартных классов существующей иерархии исключений. Ниже показана структура собственного класса исключений:

```
public class MyException extends Exception {
 public MyException () {};
 public MyException (String msg) {
 super (msg);
  }
}
```

Отсюда видны требования, предъявлемые к такому классу – он должен быть расширением встроенного класса исключений и иметь два конструктора – без параметров и с параметром типа String.

Естественно объекты-события такого класса должны генерироваться “вручную” самим программистом с помощью оператора throw. Такие собственные классы исключений используются достаточно редко и их роль состоит в увеличении наглядности текста программы и приближение ее к описываемым объектам рассматриваемой предметной области. Следующий пример описывает собственный класс исключений, реагирующий на ошибки деления на ноль.

//собственный класс исключений

```
public class DivideByZeroException extends Exception{
 public DivideByZeroException () {
  super ("Деление на нуль!");
 }
 public DivideByZeroException ( String message){
  super(message);

 }
}
//демонстрация использования класса собственных
// исключений
public class DivideByZeroExceptionDemo{
 private int numerator; //числитель
 private int denominator;//знаменатель
 private double quotient;//результат деления
 public static void main (String []  args){
  DivideByZeroExceptionDemo oneTime =
  new DivideByZeroExceptionDemo ();
  oneTime.doIt ();
 }
 public void doIt () {
  try {// начало блока try
 System.out.println ("Введите числитель:");
 numerator = System.in.readLinelnt();
  System.out.println ("Введите знаменатель:") ;
  denominator = System.in.readLinelnt();
  if (denominator ==0)// деление на ноль
  throw new DivideByZeroException ();// генерация
//исключения
  quotient = numerator/(double) denominator;
  System.out .println (numerator + "/"+ denominator + " = " +
  quotient);
 }// конец блока try
  catch (DivideByZeroException e) {//перехват
  // собственного исключения
 System.out.println(e.getMessage ());
```

```
 secondChance () ;//вторая попытка выполнения деления
 }
System.out.println("Конец программы.");
 }// конец описания метода doIt
 public void secondChance () {
  System.out.println ("Попробуйте еще раз.");
  System.out.println ("Введите числитель :");
  numerator = System.in.readLinelnt ();
  System.out.println ("Введите знаменатель:");
  System.out.println ("Позаботьтесь о том, чтобы" +
  "знаменатель не был равен нулю.");
  denominator = System.in.readLineInt();
  if (denominator ==0){
 System.out.println ("Невозможно выполнить деление на
 нуль.");
 System.out.println ( "Поскольку вы хотите от меня
 невозможного,");
 System.out.println ("программа будет завершена.");
 System.exit (0);
  }
  quotient = ( (double) numerator)/denominator;
  System.out.println (numerator + "/" + denominator + " = " +
 quotient);
  }
 }// end of secondChance
}
```

В данном приложении обработка исключительной ситуации в блоке catch заключается в предоставлении второй попытки ввода значения знаменателя (метод secondChance ()).

7.2. Вложенные блоки try

Операторы try могут быть вложенными, то есть оператор try может находиться внутри блока другого try. Всякий раз, когда управление попадает в блок try, контекст этого

исключения помещается в стек. Если вложенный оператор try не имеет обработчика catch для определенного исключения, стек "раскручивается" и проверяются на соответствие обработчики catch следующего (внешнего) блока try. Это продолжается до тех пор, пока не будет найден подходящий оператор catch либо пока не будут проверены все уровни вложенных try. Если подходящий оператор catch не будет найден, то исключение обработает система времени выполнения Java. Ниже приведен пример, в котором используются вложенные операторы try.

```
// Пример вложенных операторов try.
class NestTry {
 public static void main (String args[])
  try {
  int a = args.length;
/* Если не указаны параметры командной строки,
следующий оператор сгенерирует
исключение деления на ноль. *'/
  int b = 42 / a;
  System.out.println ("a " + a);
 try{ // вложенный блок try
/* Если используется один аргумент командной строки,
то исключение деления на ноль
будет сгенерировано следующим кодом. */
 if(a==1) a = a/(a-a); // деление на ноль
/* Если используется два аргумента командной строки,
то генерируется исключение выхода за пределы массива. */
 if (a==2)  {
 int c [] = [1];
 c[42] = 100; /* генерируется исключение выхода за пределы массива */
 }
 } catch(ArraylndexOutOfBoundsException e) {
  System.out.println ("Индекс за пределами массива: " + e);
 }
  } catch(ArithmeticException e) {
```

```
  System.out.println ("деление на 0: " + e);
 }
 }
}
```

Как видите, в этой программе один блок try вложен внутрь другого. Программа работает следующим образом. Когда вы запускаете ее без аргументов командной строки, внешним блоком try генерируется исключение деления на ноль. Запуск программы с одним аргументом вызывает генерацию исключения деления на ноль во вложенном блоке try. Поскольку вложенный блок не обрабатывает это исключение, оно передается внешнему блоку try, который обрабатывает его. Если программе передается два аргумента командной строки, то генерируется исключение выхода индекса за границы массива во внутреннемнем блоке try. Вот примеры запуска этой программы, иллюстрирующие каждый случай:

```
C:\>java NestTry
Деление на 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One
a = 1
Деление на 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One Two
a = 2
Индекс за пределами массива:
java.lang.ArraylndexOutOfBoundsException:42
```

Вложение операторов try может быть не столь очевидным, если в процессе. Выполняются вызовы методов. Например, вы можете в пределах блока try вызывать метод, а внутри этого метода иметь еще один блок try. В этом случае try в теле метода находится внутри внешнего блока try, который вызывает этот метод.

7.3. Конструкция throws

Если метод может породить исключение, которое он сам не обрабатывает, он должен специфицировать это поведение

так, чтобы вызывающий его код мог позаботиться об этом исключении. Это делается добавлением к объявлению метода конструкции throws. Конструкция throws перечисляет типы исключений, которые метод может возбуждать. Это необходимо для всех исключений, кроме имеющих тип Error, RuntimeException либо их подклассов. Все остальные исключения, которые может возбуждать метод, должны быть объявлены в конструкции throws. Если этого не сделать, получится ошибка во время компиляции.

Вот общая форма объявления метода, которая включает фразу throws:

тип имя_метода (*список параметров*) throws *список_исключений*

{

// тело метода

}

Здесь *список_ исключений* – это разделенный запятыми список исключений, которые метод может возбуждать.

Ниже представлен пример неправильной программы, пытающейся возбудить исключение, которое сама она не перехватывает. Поскольку в программе не указана фраза throws для отображения этого факта, такая программа не скомпилируется.

```
// Эта программа содержит ошибку и потому не компилируется.
class ThrowsDemo {
 static void throwOne () {
  System.out.println("Внутри throwOne.");
throw new. IllegalAccessException ("демо");
 }
 public static void main (String args[]) {
  throwOne ();
 }
}
```

Чтобы скомпилировать этот пример, нужно внести в него два изменения. Во-первых, вы должны объявить, что

throwOne () возбуждает исключение IllegalAccessException. Во-вторых, main () должен определять блок try-catch, который перехватит это исключение.

Исправленный пример выглядит следующим образом:

```
// Теперь код корректен.
class ThrowsDemo {
 static void throwOne () throws IllegalAccessException
  System.out.println ("Внутри throwOne.");
  throw new Illega1AccessException("demo");
 }
 public static void main (String args[]}
  try {
 throwOne ();
  }catch (IllegalAccessException e) {
 System.out.println ("Перехвачено" + e);
 }
 }
}
```

Вот результат, полученный при запуске этой программы:

```
Внутри throwOne
Перехвачено java.lang.IllegalAccessException: demo
```

7.4. Блок finally

Когда исключение возбуждено, выполнение метода направляется по нелинейному пути, изменяющему нормальный поток управления внутри метода. В зависимости от того, как закодирован метод, существует даже возможность преждевременного возврата управления. В некоторых методах это может служить причиной серьезных проблем.

Например, если метод при входе открывает файл и закрывает его при выходе, вероятно, вы не захотите, чтобы выполнение кода, закрывающего файл, было пропущено из-за применения механизма обработки исключений. Ключевое слово finally предназначено для того, чтобы справиться с

такой ситуацией. finally создает блок кода, который будет выполнен после завершения блока try-catch, но перед кодом, следующим за try-catch. Блок finally выполняется независимо от того, возбуждено исключение или нет. Если исключение возбуждено, блок finally выполняется, даже если ни один оператор' catch этому исключению не соответствует. В любой момент, когда метод собирается возвратить управление вызывающему коду изнутри блока try-catch из-за необработанного исключения, или явным применением оператора return блок finally будет выполнен перед возвратом управления из метода. Это может быть удобно для закрытия файловых дескрипторов либо освобождения других ресурсов, которые были получены в начале метода и должны быть освобождены перед возвратом. Оператор finally необязателен. Однако каждый оператор try требует наличия, по крайней мере, одного оператора catch или finally.

Обработка исключений представляет собой мощный механизм для управления сложными программами, которые имеют много динамических характеристик времени выполнения. Важно думать о try, throws и catch, как о чистом способе обработки ошибок и необычных краевых условиях в вашей программной логике. В отличие от ряда других языков, в которых для индикации сбоев используются коды ошибок, в Java применяются исключения. То есть, когда метод может завершиться сбоем, он возбуждает исключение. Это более чистый способ справиться со сбойными ситуациями.

Одно последнее замечание: операторы управления исключениями Java не должны рассматриваться как общий способ нелокального ветвления. Если вы будете это делать, это только запутает ваш код и усложнит его сопровождение. Также не следует увлекаться большой вложенностью блока try и даже предпочтительней использовать оператор if для анализа исключительных ситуаций типа деления на 0.

СПИСОК ЛИТЕРАТУРЫ

1. Кнут Д.Э. Искусство программирования в 3 томах. – М.: Вильямс, 2000.
2. Савитч У. Язык JAVA. Курс программирования. – М.: Вильямс, 2002
3. Арнольдс К., Гослинг Д.Язык программирования JAVA. – СПб.: Питер, 1977.
4. Вязовик Н.А. Программирование на Java, Интернет-университет информационных технологий [Электронный ресурс]. – Режим доступа: http://www.intuit.ru/studies/courses/
5. Хорстман К.С., Корнелл Г Java™ 2. Том I. Основы. Библиотека профессионала. 7 издание – М., С-П,К: Вильямс, 2007.
6. Кей С. Хорстман, Гари Корнелл Java™ 2 · Том II. Тонкости программирования. Библиотека профессионала. 7 издание – М., С-П,К: Вильямс, 2007
7. Ноутон П., Шилдт Г.. Язык Java™ 2. Наиболее полное руководство – СПб.: BHV, 2001.
8. Шилдт Г. Полный справочник по Java SE™ 6 Edition 7 -е издание. – М.: Вильямс, 2007.
9. Картузов А.В. Программирование на языке JAVA. [Электронный ресурс]. – Режим доступа: http://www.mstu.edu.ru/study/materials/java/
10. Юдахин Р.В основы программирования на JAVA.учебное пособие. – Томск: Изд-во ТУСУР, 2004.
11. Васильев А.Н. Java. Объектно-ориентированное программирование. – СПб.: Питер, 2013.
12. Deitel P. J., Deitel H. M. Java™ How to Program, Seventh Edition. – Upper Saddle River, New Jersey. 2007.
13. Horstmann C. Big Java. San Jose State University, John Wiley & Sons, Inc, 2010.
14. Perry J. S. Introduction to Java programming, Part 1: Java language basics, Copyright IBM Corporation, 2010
15. Perry J. S. Introduction to Java programming, Part 2:Constructs for real-world applications, Copyright IBM Corporation, 2010
16. Хабибуллин И. Java 7. – СПб.: BHV, 2012.
17. Хортон А. Java 2 в 2 томах. – М.: Лори, 2013.
18. Эккель Б. Философия Java – СПб.: Питер, 2013

19. Курс лекции по языку Java. [Электронный ресурс]. – Режим доступа: http://www.studmed.ru/docs/document10693

20. The Java Tutorials. [Электронный ресурс]. – Режим доступа: http://docs.oracle.com/javase/tutorial/index.html

Учебное издание

ФОФАНОВ Олег Борисович

ИНФОРМАТИКА И ПРОГРАММИРОВАНИЕ

Учебное пособие

Научный редактор
доктор технических наук, профессор В.А.Силич

Выпускающий редактор *Д.В. Заремба*
Редактор *В.Ю. Пановица*
Компьютерная верстка *К.С. Чечельницкая*