МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ имени академика С. П. КОРОЛЕВА

С.В. Востокин, О.П. Солдатова

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЕ МЕТОДЫ И СИСТЕМЫ

Язык программирования Java

Курс лекций

САМАРА 2002

УДК 681.3

С.В. Востокин, О.П. Солдатова. **Объектно-ориентированные методы и системы. Язык программирования Java**: *Курс лекций* / Самар. гос. аэрокосм. ун-т. Самара, 2002. 60 с.

ISBN 5-7883-0225-0

Курс лекций предназначен для студентов заочной формы обучения специальности 22.02 — "Автоматизированные системы обработки информации и управления" — и содержит описание объектно-ориентированных методов программирования с использованием среды программирования и языка Java. Детально описывается синтаксис и объектная модель языка Java. Описываются инструментальные средства SDK, приводятся упражнения и примеры программ.

Данный курс лекций будет также полезен студентам других специальностей при изучении основ языка Java, обучающихся как по очной, так и заочной форме обучения. Подготовлен на кафедре ИСТ.

Библиогр.: 16 назв.

Печатается по решению редакционно-издательского совета Самарского государственного аэрокосмического университета имени академика С.П. Королева

Рецензенты: канд. техн. наук, доц. Л. А. Жаринова
канд. физ.-мат. наук Д. Л. Головашкин

ISBN 5-7883-0225-0

© Востокин С.В., Солдатова О.П., 2002
© Самарский государственный аэрокосмический университет, 2002

Введение

Цель данного курса лекций - познакомить студентов с объектно-ориентированным языком программирования Java, а также с инструментами среды программирования этого языка.

В работе рассматриваются объектно-ориентированная парадигма программирования на примере объектной модели языка Java и особенности ее реализации. Наибольшее внимание уделяется собственно языку программирования Java. Подробно описывается синтаксис Java, рассматриваются отличия этого языка от других популярных языков Pascal, C, C++. При этом мы намеренно опускаем из рассмотрения методы объектно-ориентированного проектирования и анализа, а также схемы разработки. В работе не рассматривается подробно интерфейс прикладного программирования Java. Предполагается, что эти вопросы изучаются студентами самостоятельно или в отдельных курсах.

Для работы с курсом лекций желательно иметь доступ к сети Internet. Многие материалы, предлагаемые для самостоятельного изучения, размещены на серверах компании Sun Microsystems.

Минимальная подготовка: освоенный курс "Алгоритмические языки программирования", знание основ процедурного программирования, умение программировать на языках Паскаль или Си.

1. Общие сведения о среде программирования и языке Java

Происхождение. Основные характеристики и отличительные особенности. Где найти информацию по Java.

Язык Java разработан инженерами компании Sun Microsystems. Создателем языка считается Джеймс Гослинг (James Gosling). Работы по проекту начались в 1990 году, а 1995 компания Sun Microsystems представила готовый продукт под названием Java.

Подробно с историей Java можно познакомиться на сайте компании Sun Microsystems по адресу:

`http://www.sun.ru/win/java/start/intro/history.html`.

Напомним, что на момент начала разработки Java такие "современные" языки программирования, как Pascal (автор — Никлаус Вирт, 1970), C (автор — Деннис Ричи, 1975), C++ (автор — Бьерн Страуструп, 1980), уже широко применялись. Зачем же понадобился новый язык программирования? Для ответа на этот вопрос рассмотрим свойства языка Java и цели его разработки:

`http://www.sun.ru/win/java/start/intro/properties.html`

Java — объектно-ориентированный язык. Это означает, что при разработке приложений программист мыслит в терминах классов, объектов и

их взаимодействия, а не на языке процедур. В отличие от языков Pascal и C++, Java с самого начала создавался как объектно-ориентированный язык. В нем отсутствуют записи (структуры). За исключением простых числовых и символьных типов все типы языка — объекты Java. Интерфейс прикладного программирования (API) среды Java также представляет собой библиотеку классов.

Java — интерпретируемый и высокоэффективный язык. Как известно, синтаксис языков программирования не накладывает специальных ограничений на то, является ли язык интерпретируемым или компилируемым. Обычно Pascal и C++ реализуются как компилируемые языки, однако имеются и интерпретаторы для этих языков. В отличие от языков Pascal и C++, Sun Microsystems документирует не только входной язык Java, но и результат его обработки — байт-код виртуальной машины Java (JVM). Виртуальная машина Java — это интерпретатор, который реализуется для каждой из целевых платформ (Windows, Linux, Mac OS и т.д.).

Конечно, скорость интерпретации байт-кода меньше, чем скорость исполнения машинных инструкций. Однако формат байт-кода рассчитан на динамическое преобразование в код для конкретного процессора во время исполнения программы.

Java — независимый от архитектуры язык. Архитектурная независимость (нейтральность) достигается благодаря использованию байт-кода независимого от архитектуры формата. Байт-код один и тот же для всех платформ, различаются только интерпретаторы байт-кода (виртуальные машины Java).

Java — динамический язык. Это означает, что язык, а точнее JVM, использует динамическую компоновку. Напомним, что в реализации традиционных языков (C, C++, Pascal) используется статическая компоновка кода с использованием специальных программ — редакторов связей или линковщиков. В Java код классов подгружается по мере необходимости. Динамическая компоновка оказывается удобной для написания распределенных приложений, работающих в сети Internet.

Java — простой язык. Разработчики Java ставили своей целью создать простой в изучении язык с минимальным количеством конструкций. Другой целью было сделать язык, который показался бы знакомым большинству программистов. Поэтому он сделан похожим на языки C и C++. Наиболее сильное упрощение, примененное в Java,— отсутствие указателей. В языке осуществляется автоматическое взятие ссылок и их разыменование и применяется автоматическое управление динамической памятью.

Java — устойчивый и безопасный язык. Язык Java позволяет писать более устойчивые и надежные программы. Это означает, что благодаря особенностям синтаксиса в нем устраняются некоторые ошибки программирования. К таким ошибкам относятся ошибки приведения типов; ошибки, связанные с неявным объявлением методов; ошибки при работе с указателями и динамической памятью; ошибки обращения за границы строк

и массивов. Кроме этого в языке имеются встроенные операторы обработки исключений с более строгой, чем в C++, нотацией.

Поддержка безопасности — это защита системы от последствий исполнения ошибочных или злонамеренных программ. Безопасность реализуется за счет нескольких механизмов. Они включают модель песочницы (sandboxing), верификацию байт-кода, а также цифровую подпись кода. Свойства устойчивости и безопасности крайне важны в задачах распределенной обработки.

Java — многопотоковый язык. Современные операционные системы могут одновременно исполнять несколько задач или процессов. Аналогичная возможность реализована непосредственно в языке Java. Потоки (или облегченные процессы) — это специальные объекты Java. Кроме того, в языке есть конструкции, служащие для синхронизации потоков.

Итак, в языке Java были реализованы несколько инженерных решений, которые позволили сделать язык, удобный для написания независимых от архитектуры и аппаратной платформы распределенных приложений для сети Internet.

Язык Java является собственностью компании Sun Microsystems. Техническую документацию, учебные материалы и примеры программ на Java можно найти на сервере компании по адресу: `http://java.sun.com/`, а также на русскоязычном сервере `http://www.sun.ru/win/java/`.

Много полезной информации можно найти по адресу `http://www.javapower.ru`.

Вопросы и упражнения:

1. Познакомьтесь с содержанием русскоязычного сервера Java по адресу `http://www.sun.ru/win/java/`.
2. Обсудите проблемы, которые возникают при переносе программ с одной аппаратной платформы на другую.

2. Настройка инструментальных средств SDK

Установка SDK. Настройка переменных окружения PATH и CLASSPATH. Тестирование инструментов SDK.

Установите на компьютере базовый набор средств разработки приложений на Java. Набор средств разработки называется software development kit (SDK). Для работы с примерами из данного пособия подойдет версия Java™ 2 SDK, Standard Edition, v 1.3.1. Средства SDK свободно распространяются компанией Sun Microsystems по адресу `http://java.sun.com/j2se/`. Там же Вы можете скачать документацию по Java на английском языке.

Установленный SDK представляет собой набор консольных программ, включающий компилятор, интерпретатор байт-кода, генератор документации, отладчик и другие инструменты. Заметим, что для профессиональной работы обычно используются интегрированные среды разработки (IDE), например, Forte for Java, Borland JBuilder, WebGain VisualCafé, Oracle JDeveloper, Metrowerks CodeWarrior.

После того как SDK был установлен, выполним его настройку и убедимся в его корректной работе. Для этого напишем простейшую программу на Java и протестируем на ней работу некоторых инструментов SDK.

1. Создайте рабочий каталог, например, `C:\MyJava`.

2. В рабочем каталоге с помощью любого текстового редактора создайте файл `hello.java`. Поместите в этот файл следующий код:

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

Обратите внимание на то, чтобы имя класса (`hello`) соответствовало имени файла `hello.java`.

3. Найдите каталог, в котором установлен SDK, например, `c:\jdk1.3`. В подкаталоге `c:\jdk1.3\bin` найдите программу компилятор `javac.exe`. Для того чтобы не указывать каждый раз полный путь к компилятору, добавьте в переменную окружения `PATH` Вашей системы маршрут к инструментам SDK, например, `c:\jdk1.3\bin`. Выполните, если необходимо, перезагрузку компьютера.

4. Перейдите в Ваш рабочий каталог и выполните компиляцию с помощью команды `javac hello.java`. В рабочем каталоге будет создан еще один файл с именем `hello.class`, который содержит байт-код программы.

5. Для запуска программы воспользуемся интерпретатором `java.exe`. Интерпретатору для работы необходима информация о размещении кода Вашего класса, а также классов, к которым обращается данный класс. Код системных классов (который находится в архиве `\jre\lib\rt.jar`) интерпретатор находит самостоятельно. Для того чтобы запустить код программы `hello`, требуется явно указать размещение класса `hello.class`. Это можно сделать двумя способами: 1) Передать маршрут к классам в качестве параметра (`java.exe -classpath C:\MyJava hello`). 2) Настроить переменную окружения `CLASSPATH`, записав в нее необходимые маршруты, разделив их точкой с запятой. Учтите, что поиск классов идет в порядке, заданном в переменной `CLASSPATH`; задание `-classpath` в качестве параметра перекрывает значение переменной окружения. Для задания текущего каталога используется точка, например, `set classpath= .;C:\MyJava`.

6. Выполнив настройки, как описано в п. 5, убедитесь, что программа запускается командами ***java.exe - classpath C:\MyJava hello*** и ***java.exe hello***.

7. Выполним документирование нашей программы. Для этого нужно перейти в каталог с файлом ***hello.java*** и выполнить команду ***javadoc hello***. В каталоге будут сгенерированы файлы документации в формате HTML.

8. Посмотрим, как выглядит байт-код нашего класса. Для этого запустим дизассемблер командой ***javap -c hello***. Дизассемблер покажет интерфейс класса, а также методы класса в мнемонических кодах, соответствующих командам байт-кода.

9. Запустим программу под управление отладчика и выполним некоторые команды отладки:

- выполните команду ***jdb hello*** — появится приглашение ***'>'*** для ввода команд отладчика;
- установите точку останова командой ***stop in hello.main*** — отладчик ответит, что установлена отложенная точка останова;
- запустите программу командой ***run*** — отладчик ответит, что программа остановлена в точке останова;
- посмотрите исполняемый код с помощью команды ***list***;
- выполните программу в пошаговом режиме с помощью команд ***step***.

Таким образом, мы выполнили установку и настройку стандартного пакета разработки SDK для Java и познакомились с компилятором, интерпретатором, документатором, дизассемблером и отладчиком из набора инструментов SDK.

Вопросы и упражнения:

1. Ознакомьтесь с опциями командной строки, рассмотренными в параграфе инструментов.
2. Самостоятельно изучите команды консольного отладчика ***jdb.exe***. Попробуйте выполнить отладку примеров приложений, поставляемых с SDK (например, из каталога ***\demo\jfc\SimpleExample***).
3. Попробуйте сначала запустить виртуальную машину Java, а затем подключить к ней отладчик (используйте документацию консольного отладчика ***jdb.exe***).

3. Лексическая структура языка Java

Пробельные символы. Комментарии. Идентификаторы. Ключевые слова. Литералы. Разделители. Операторы.

Начнем изучение языка Java с его лексической структуры. Лексическую структуру языков программирования составляют пробельные символы, комментарии, идентификаторы, ключевые слова, литералы, разделители и операторы.

Программы на Java пишутся в кодировке Unicode, однако язык определяет процедуры лексической трансляции, позволяющие использовать кодировку ANSII. Главное отличие этих кодировок в том, что в Unicode символ хранится в двух байтах, а в ANSII — в одном. Два байта достаточны для кодирования всех символов национальных алфавитов. Язык Java позволяет использовать национальные символы в названиях идентификаторов. Для представления символов Unicode в кодировке ANSII используется escape-последовательность вида `\uxxxx`, где `x` обозначает шестнадцатеричную цифру.

Кроме пробела пробельными символами языка являются символ табуляции, возврата каретки (*caret return*), прогона страницы (*form feed*), прогона строки (*line feed*), а также комбинация символов *<caret return> <line feed>*. Напомним, что наличие пробельных символов в спецификации языка позволяет выполнить удобное для чтения форматирование текста программы.

В Java используются комментарии двух типов. Это традиционный комментарий в стиле C и закомментированная строка:

```
/* Это комментарий в стиле C */
// Это закомментированная строка
```

Для комментариев определены следующие правила:

- комментарии не могут быть вложенными;
- все, что находится под знаком //, — это комментарий, включая комментарий в стиле C /* */ ;
- все, что находится внутри /* */, — это комментарий, включая // ;
- комментарии не содержатся внутри символьных и строковых литералов.

Идентификатор Java — это последовательность символов неограниченной длины, начинающаяся с буквы. Идентификаторами не являются ключевые слова, булевские литералы и *null*-литерал. Идентификаторы могут быть составлены из символов национальных алфавитов. Последнее свойство отличает язык Java от таких языков, как C и C++, которые допускают идентификаторы только в кодировке ASCII.

В языке Java определено 48 ключевых слов. Ключевые слова *const* и *goto* зарезервированы, но в настоящее время не используются. Основные

ключевые слова и их семантика рассматриваются в других параграфах пособия.

Литералами называют символьные последовательности, которые используются для обозначения значений примитивных типов языка. В языке Java определено шесть типов литералов.

Целые литералы. Целые литералы могут быть заданы в десятичной, шестнадцатеричной и восьмеричной системах счисления. Целые литералы по умолчанию имеют тип `int`. Суффикс (`L` или `l`) изменяет тип литерала на `long`. Шестнадцатеричные литералы начинаются с пары символов `0x` или `0X`, а восьмеричные с нуля. Примеры целых литералов языка Java:

```
0     2     0372 0xDadaCafe      1996 0x00FF00FF
01    0777L      0x100000000L    2147483648L     0xC0B0L
```

Литералы с плавающей точкой. Эти литералы имеют тип по умолчанию `double`. Если требуется, можно изменить тип по умолчанию на тип `float`. Для этого используется суффикс `F` или `f`. Также можно использовать необязательный суффикс `D` или `d` для типа `double`.

Примеры литералов с плавающей точкой:

```
1e1f 2.f   .3f   0f    3.14f      6.022137e+23f
1e1  2.    .3    0.0   3.14 1e-9d
```

Булевские литералы. Литералы `true` и `false` обозначают значения логических переменных «истина» и «ложь».

Символьные литералы. Символьный литерал — это одиночный символ или escape-последовательность, заключенная в кавычки.

В языке Java используются следующие escape-символы:

```
\b                /* \u0008: backspace BS */
\t                /* \u0009: horizontal tab HT */
\n                /* \u000a: linefeed LF */
\f                /* \u000c: form feed FF */
\r                /* \u000d: carriage return CR */
\"                /* \u0022: double quote " */
\'                /* \u0027: single quote ' */
\\                /* \u005c: backslash \ */
```

Аналогично языку C в Java используются escape-символы в восьмеричной форме.

Примеры символьных литералов:

```
'a'  '%'  '\t' '\\' '\'' '\u03a9'  '\uFFFF'  '\177'
```

Символьные литералы в Java имеют следующие особенности:

- наличие лексической трансляции Unicode-символов на первой стадии компиляции запрещает использование символов `'\u000a'` для обозначения перевода строки и символа `'\u000d'` для обозначения возврата каретки (для этого следует использовать символы `'\n'` и `'\r'`);
- в отличие от языка C, символьный литерал обозначает строго один символ.

Строковые литералы. Это нуль или несколько символов, заключенных в двойные кавычки. Символы внутри строки могут быть escape-символами. В

языке Java запрещен перенос символьных литералов на другую строку. Для переноса следует использовать оператор конкатенации +.

Примеры символьных литералов:

```
""                    // пустая строка
"\""                  // строка с одним символом "
"This is a string"      // строка из 16 символов
"This is a " +          // константное выражение,
    "two-line string"   // составленное из двух литералов
```

Каждый строковый литерал представляет собой ссылку на объект класса String. Подробнее строки в языке Java рассматриваются в параграфе 6.

Null-литерал. Этот литерал имеет значение `null`, обозначающее пустую ссылку.

Разделителями в языке Java являются символы `"(){}[];,."` . Заметим, что в отличие от языка Java, символы `"[],."` в языках C и C++ обозначают операторы.

В языке Java определено 37 операторов:

```
=    >    <    !    ~    ?    :
==   <=   >=   !=   &&   ||   ++   --
+    -    *    /    &    |    ^    %    <<   >>   >>>
+=   -=   *=   /=   &=   |=   ^=   %=   <<=  >>=  >>>=
```

Операторы во многом аналогичны операторам языка C. Однако имеются различия: новая семантика операторов «и» и «или» в булевских выражениях, новая семантика операторов сдвига. Более подробно эти различия рассматриваются в следующем параграфе.

Вопросы и упражнения:

1. Ознакомьтесь со спецификацией языка Java на сервере Sun Microsystems по адресу `http://java.sun.com/docs/books/jls/`.
2. Ознакомьтесь со спецификацией виртуальной машины Java на сервере Sun Microsystems по адресу `http://java.sun.com/docs/books/vmspec/`.

4. Переменные и простые типы данных языка Java

Классификация типов. Диапазон значений. Семантика операторов для типов. Переменные. Примеры объявлений переменных и приведения типов.

Язык Java использует строгую типизацию. Это означает, что тип переменных и выражений программы известен на этапе компиляции. Тип определяет множество значений, которые могут принимать переменные, а также множество операторов, которые могут быть применены к переменным, и их семантику.

Типы языка Java разделены на две категории — это примитивные типы и ссылочные типы. К примитивным типам относят тип **boolean** и числовые типы. Числовые типы включают интегральные типы **byte**, **short**, **int**, **long**, **char** и числа с плавающей точкой (вещественные) **float** и **double**. К ссылочным типам относятся типы классов, типы интерфейсов и типы массивов.

Переменные интегральных типов языка Java могут принимать следующие значения:

- **byte**, от *-128* до *127* включительно;
- **short**, от *-32768* до *32767* включительно;
- **int**, от *-2147483648* до *2147483647* включительно;
- **long**, от *-9223372036854775808* до *9223372036854775807* включительно;
- **char**, от *'\u0000'* до *'\uffff'* включительно (или от *0* до *65535*);
- переменные типа **boolean** могут принимать значения **true** и **false**.

С интегральными типами используются следующие операторы.

Операторы сравнения, возвращающие значение типа **boolean**:

- сравнение *<, <=, >, >=* ;
- проверка чисел на равенство *==* , *!=* .

Числовые операторы, возвращающие значение типа **int** или **long**:

- унарный плюс и минус *+* и *-* ;
- мультипликативные операторы **, /, %* ;
- операторы сложения/вычитания *+* и *-* ;
- оператор инкремента в префиксной и постфиксной форме *++* ;
- оператор декремента в префиксной и постфиксной форме *--*.

Операторы знакового и беззнакового сдвига *<<, >>* , *>>>* .

Оператор двоичного дополнения *~* .

Битовые операторы «и», «или», «исключающее или» *&, /, ^* .

Условный оператор *? :* .

Оператор приведения типа из любого интегрального в любой числовой тип.

Оператор конкатенации строк *+*, когда он применяется к строковой переменной и операнду интегрального типа (в этом случае результатом является строка, получающаяся при конкатенации исходной строки со строкой, представляющей число в десятичной форме).

С вещественными типами используются следующие операторы.

Операторы сравнения, возвращающие значение типа **boolean**:

- сравнение *<, <=, >, >=* ;
- проверка чисел на равенство *==* , *!=* .

Числовые операторы, возвращающие значение типа **float** или **double**:

- унарный плюс и минус *+* и *-* ;
- мультипликативные операторы **, /, %* ;
- операторы сложения/вычитания *+* и *-*;

- оператор инкремента в префиксной и постфиксной форме `++`;
- оператор декремента в префиксной и постфиксной форме `--`.

Условный оператор `? :` .

Оператор приведения типа из любого вещественного в любой числовой тип.

Оператор конкатенации строк `+`, когда он применяется к строковой переменной и операнду вещественного типа (в этом случае результатом является строка, получающаяся при конкатенации исходной строки со строкой, точно представляющей вещественное число).

С булевыми типами используются следующие операторы:

Операторы проверки на равенство `==` и `!=`.

Оператор логического дополнения `!` .

Логические операторы «и», «исключающее или», «или»: `&, ^, |`.

Условные операторы «и» и «или» `&&` и `||`.

Условный оператор `? :` .

Оператор конкатенации строк `+`, когда он применяется к строковой переменной и операнду булевого типа (в этом случае результатом является строка, получающаяся при конкатенации исходной строки со строками `"true"` или `"false"`).

Поясним семантику некоторых операторов. В Java операторы сдвига имеют следующие особенности. Операнды операторов сдвига должны иметь строго интегральный тип. Для операторов сдвига вправо различается сдвиг с распространением знакового разряда (`>>`) и без распространения знакового разряда (`>>>`). Введение специального оператора понадобилось по причине отсутствия в Java беззнаковых интегральных типов.

Для явного приведения типа в Java используется следующая нотация: `(<новый_тип>)<выражение>`. Здесь `<новый_тип>` — имя типа.

Кроме явного приведения типов, в выражениях языка Java используется приведение типов по умолчанию с расширяющей семантикой. Это означает, что неявно выполняется преобразование из типа, содержащего меньше значений, в тип, содержащий больше значений, но не наоборот. Интегральные типы неявно приводимы к вещественным типам, но не наоборот. Сужающее тип преобразование, за исключением нескольких случаев, может быть выполнено только явно.

Проверка на равенство для вещественных типов производится в соответствии со стандартом IEEE 754. То есть выполняются правила:

- Если один из операндов имеет значение `NaN` (не число), тогда результат выполнения `==` равен `false`, а результат выполнения `!=` равен `true`. Выражение `x!=x` истинно только в том случае, если `x` имеет значение `NaN`.
- Положительный и отрицательный ноль считаются равными. То есть `-0.0==0.0` равно `true`.
- Два отличающихся вещественных значения считаются неравными. В частности, плюс бесконечность сравнима только с плюс

бесконечностью, а минус бесконечность — только с минус бесконечностью.

В языках C и C++ оператор деления по модулю `%` используется только с интегральными операндами, однако в Java он используется также с вещественными операндами. Для интегральных типов остаток определяется так, чтобы выполнялось равенство `(a/b)*b+(a%b)==a`. Например,

```
5%3 равно 2          (заметим, что 5/3 равно 1)
5%(-3) равно 2       (заметим, что 5/(-3) равно -1)
(-5)%3 равно -2      (заметим, что (-5)/3 равно -1)
(-5)%(-3) равно -2   (заметим, что (-5)/(-3) равно 1)
```

Для вещественных типов операция остатка вначале выполняет округление методом отсечения до целого, затем выполняется взятие остатка и приведение к вещественному типу. Например,

```
5.0%3.0 равно 2.0
5.0%(-3.0) равно 2.0
(-5.0)%3.0 равно -2.0
(-5.0)%(-3.0) равно -2.0
```

Операции инкремента (увеличение на единицу) и декремента (уменьшение на единицу) применяются как к интегральным, так и к вещественным типам. При этом результат имеет тот же тип, что и операнд.

Условный оператор работает аналогично оператору *<условие>?<если_истина>:<если_ложь>* в C. Если выражение *<условие>* равно `true`, то значение оператора определяется вычислением выражения *<если_истина>*, иначе — вычислением выражения *<если_ложь>*. Особенностью языка Java является то, что тип операнда *<условие>* должен быть строго `boolean`.

Заметим, что при использовании операндов разного типа требуется более детальное понимание их семантики. Подробно семантика операторов описывается в спецификации языка. На практике рекомендуется избегать использования операндов разных типов или применять явное приведение типа.

С булевыми операндами в языке Java определено два новых, по сравнению с C и C++, оператора. Это условные операторы «и» и «или» (`&&`, `||`). Оператор `&&` аналогичен оператору «и» `&`, однако он вычисляет операнд справа только, если операнд слева равен `true`. Оператор `||` аналогичен оператору «или» `|`, однако он вычисляет операнд справа только, если операнд слева равен `false`.

Переменная — это место в памяти, где хранится значение и связанный с ним тип. Переменные примитивных типов хранят значения непосредственно. Переменные ссылочных типов хранят либо нулевую ссылку, либо ссылку на объект. Со ссылочными типами используется оператор `instanceof`, который возвращает значение `true`, если объект, стоящий с левой стороны, является экземпляром класса или реализацией интерфейса стоящего справа. В противном случае он возвращает `false`. Если стоящий слева операнд равен `null`, также возвращается значение `false`.

Язык определяет семь типов переменных в зависимости от контекста их использования:

- переменные класса;
- переменные экземпляра;
- переменные-компоненты массива;
- переменные-параметры методов;
- переменные-параметры конструкторов;
- переменные-параметры обработчиков исключений;
- локальные переменные.

Новое свойство переменных в Java (по сравнению с языками C, C++, Pascal) определяется ключевым словом ***final***. Значение ***final***-переменной присваивается только один раз. Если это ссылочная переменная, то состояние объекта может изменяться, но этой переменной нельзя присвоить ссылку на другой объект.

В языке Java (в отличие от языка C) строго определены начальные значения переменных классов и экземпляров для всех примитивных типов. У переменных числовых типов начальным значением является ноль, у переменных булевого типа — ***false***, у переменных ссылочных типов — ***null***. Значение локальным переменным должно быть явно присвоено.

Значение переменной можно изменить операторами присваивания. В языке Java, также как в языках C и C++, присваивание значения является обычным оператором (в отличие от языка Pascal). Операторы присваивания, как и другие операторы, могут использоваться внутри выражений. В языке определены операторы присваивания вида ***<бинарный_оператор>=*** для сокращенной записи выражения

<переменная>=<переменная><бинарный_оператор><выражение>.

В заключение рассмотрим несколько примеров объявлений переменных и приведения типов.

```
int a = 100;
byte b = (byte) a;
```

Здесь выполняется сужающее приведение типа. Так как тип ***a*** шире, чем тип ***b***.

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

Этот пример иллюстрирует автоматическое расширение типа для промежуточных результатов. Переполнение не происходит, так как типы операндов автоматически повышаются до ***int***.

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

Этот код вызывает ошибку «***Explicit cast needed to convert int to byte***». Причина ошибки — автоматическое повышение типа при умножении. Так как выражение ***((byte)b)*2*** имеет тип ***int***, неприводимый к ***byte***. Исправленный код использует явное приведение типа.

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

Вопросы и упражнения:

1. Имеется переменная типа `double`. Напишите выражение для вычисления первой цифры справа от десятичной точки, считая, что значение переменной записано в формате с фиксированной точкой.
2. Имеется переменная типа `double`. Напишите выражение для вычисления второй цифры слева от десятичной точки в представлении значения этой переменной в формате с фиксированной точкой. Результат запишите в переменную типа `char`. Если такой цифры в представлении числа нет, то присвойте переменной значение `'_'`.

5. Управление вычислениями в языке Java

Блоки. Управляющие конструкции: if, switch, while, do, for, break, continue, return.

Управление вычислениями в Java идентично средствам, используемым в C и C++. Особенностью языка Java является применение операторов `break` и `continue` с метками и отсутствие оператора `goto`. Заметим, что конструкция языка, которая отвечает за вычисления в Java, как и в языках C и C++, по-английски называется *statement*. Термин же оператор (*operator*) относится к логическим, арифметическим и т.д. *операциям*. Чтобы избежать разночтений, мы будем придерживаться более привычной терминологии, называя конструкции `if`, `switch`, `while`, `do` и т.д. операторами, а `(+, -, /, *` и т.д.) — операциями.

Блоки. Простейший оператор на Java — это выражение, заканчивающееся точкой с запятой. Если выражение пусто, то оператор называется пустым оператором. Операторы могут быть сгруппированы в блоки с использованием скобок `{}`. Блоки могут помечаться метками, например, `label:{…}`.

Оператор if. Этот оператор используется для организации ветвления.

```
if (bytesAvailable > 0){
     ProcessData();
     bytesAvailable -= n;
}

if (month == 12 || month == 1 || month == 2){
     season = "Winter";
} else if (month ==3 || month == 4 || month == 5) {
     season = "Spring";
} else if (month == 6 || month == 7 || month == 8) {
     season = "Summer";
```

```
} else if (month == 9 || month == 10 || month == 11) {
     season = "Autumn";
} else {
     season = "Bogus Month";
}
```

Оператор switch. Этот оператор также используется для организации ветвления.

```
switch (month) {
     case 12: // FALLSTHROUGH
     case 1: // FALLSTHROUGH
     case 2:
          season = "Winter";
          break;
     case 3: // FALLSTHROUGH
     case 4: // FALLSTHROUGH
     case 5:
          season = "Spring";
          break;
     case 6: // FALLSTHROUGH
     case 7: // FALLSTHROUGH
     case 8:
          season = "Summer";
          break;
     case 9: // FALLSTHROUGH
     case 10: // FALLSTHROUGH
     case 11:
          season = "Autumn";
          break;
     default:
          season = "Bogus Month";
}
```

Оператор while. Это оператор цикла с предусловием.

```
int n = 10;
while (n > 0) {
     System.out.println("tick " + n);
     n--;
}
```

Оператор do. Это оператор цикла с постусловием.

```
do {
     System.out.println("tick " + n);
} while (--n > 0);
```

Оператор for. Оператор цикла, содержащий секцию инициализации, проверки условия и итерации.

```
for(int n = 10; n > 0; n--)
     System.out.println("tick " + n);

for (a = 1, b = 4; a < b; a++, b--) {
     System.out.println("a = " + a);
     System.out.println("b = " + b);
}
```

Особенностями операторов, использующих логические выражения (*if*, *while*, *do*, *for*), является то, что они требуют выражения булевого типа. Иначе возникает синтаксическая ошибка.

Оператор break. Оператор выхода из блока. Имеется вариант оператора с меткой и без метки. Оператор прерывает исполнение блока, помеченного меткой. Если метка не используется, то происходит выход из самого «внутреннего» блока.

```
boolean t = true;
a:{
     b:{
          c:{
               System.out.println("Before the break");
               // Перед break
               if (t) break b;
               System.out.println("This won't execute");
               // Не будет выполнено
          }
          System.out.println("This won't execute");
          // Не будет    выполнено
     }
     System.out.println("This is after b"); //После b
}
```

Оператор continue. Оператор продолжения. Имеется вариант оператора с меткой и без метки. Оператор прерывает исполнение текущей итерации блока, помеченного меткой. Если метка не используется, то происходит прекращение итерации для самого «внутреннего» блока.

```
for (int i=0; i < 10; i++) {
     System.out.print(i + " ");
     if (i % 2 == 0) continue;
     System.out.println("");
}

outer:for(int i=0; i < 10; i++) {
     for (int j = 0; j < 10; j++) {
          if (j > i) {
               System.out.println("");
               continue outer;
          }
          System.out.print(" " + (i * j));
     }
}
```

Оператор return. Оператор возврата из метода. Если метод возвращает значение, то после оператора должно следовать выражение соответствующего типа.

```
class ReturnDemo {
     public static void main(String args[]){
          boolean t = true;
          System.out.println("Before the return");
          //Перед оператором return
          if (t) return;
```

```
        System.out.println("This won't execute");
        //Это не будет выполнено
    }
}
```

В заключение заметим, что в языке Java имеется еще один механизм управления вычислениями — это структурированная обработка исключений. Данный механизм рассматривается в отдельном параграфе.

Вопросы и упражнения:

1. Найдите все простые числа от 2 до 1000.
2. Найдите все простые делители числа в диапазоне от 2 до 1000.

6. Строки и массивы в языке Java

Способы создания массивов. Многомерные массивы. Доступ к элементам массивов. Особенности реализации строк в языке Java.

Массивы в языке Java являются разновидностью ссылочных типов. Массивы имеют следующие свойства:

- массивы обрабатываются по ссылкам;
- массивы создаются динамически при помощи ключевого слова new;
- массивы автоматически попадают в мусор, если на них не ссылаются.

Существует два способа создания массивов. В первом используется оператор new и указывается размер массива.

```
byte octet_buffer[]=new byte[1024];
Button buttons[] =new Button[10];
```

Элементы массивов, создаваемых таким способом,инициализируются значениями соответствующих типов по умолчанию. Напомним, что для числовых типов это ноль, а для ссылочных — пустая ссылка.

При втором способе массив создается путем инициализации его элементов, что похоже на инициализацию в языке C.

```
int lookup_table[]={1,24,8,16,32,64,128};
```

При этом динамически создается массив и его элементы инициализируются определенными значениями. Значения, используемые при инициализации, не обязательно являются константами.

Массивы могут создаваться и инициализироваться анонимно.

```
Menu m=creaeMenu("File",new String[] {"Open...", "Save", "Quit"});
```

Многомерные массивы в языке Java реализованы как массивы массивов. Память под многомерные массивы выделяется с использованием ключевого слова new, как показано ниже.

```
Byte TwoDimArray[][]=new byte[256][16];
```

При объявлении многомерного массива не обязательно определять количество элементов в каждой размерности. Например:

```
int threeD[][][]=new int[10][][];
```

Это выражение размещает массив из 10 элементов, каждый из которых имеет тип `int[][]`. Таким образом, размещается одномерный массив, хотя при соответствующей инициализации элементов массива значениями он может стать многомерным. Правило размещения таких массивов гласит: для первых n размерностей должно быть определено число элементов, а для остальных m размерностей этого делать не обязательно. Следующая запись правильна:

```
String lots_of_string[][][][] = new String[5][3][][];
```

А эта — нет:

```
double temperature_data[][][]=new double[100][][10];
```

Многомерные массивы можно объявлять и инициализировать при помощи вложенных инициализаций.

```
Sring param_info[][]={
     {"foreground", "Color", "foreground color"},
     {"background", "Color", "background color"},
     {"message", "String", "the banner to display"}
}
```

Так как массив реализуется в виде массива массивов, он не обязательно должен быть прямоугольным. Так выглядит создание непрямоугольного массива путем вложенной инициализации.

```
int [][] twodim = {{1,2},{3,4,5},{5,6,7,8}};
```

Заметим, что скобки в объявлении массива могут следовать непосредственно за названием типа. Можно также смешивать два вида объявлений, например:

```
byte[] row, column, matrix[];
```

Многомерные массивы можно имитировать с помощью одномерных, как это делается в языке С.

```
final int rows=600;
final int columns = 800;
byte pixels[]=new byte[rows*columns];
//доступ к элементу [i,j] осуществляется так
byte b = pixels[i+j* columns];
```

Доступ к элементам многомерных массивов осуществляется так же, как и в С. Размерность массива не является частью типа, как в языке С. Для определения длины массива используется единственная переменная массива *length*. Она определена как *final*, следовательно, ее значение не может быть изменено.

```
int a[]=new int[100];
a[0]=0;
for(int i=1;i<a.length;i++) a[i]=i+a[i-1];
```

При каждом обращении к массиву проверяется значение индекса. В случае обращения за границу массива генерируется исключение *Array Index Out Of Bounds Exeption*.

В языке Java строки не являются, как в C, массивами символов, оканчивающимися нулем. Строки — это экземпляры класса *java.lang.String*, которые обрабатываются компилятором, как если бы они принадлежали к простому типу. Строковые литералы в программе соответствуют объектам типа *String*. В языке определен специальный оператор для конкатенации (слияния) строк.

Особенностью объектов *String* является их неизменяемость. Нет методов, способных изменить содержимое объектов *String*. Если необходимо это сделать, то из объекта *String* следует создать объект *StringBuffer*, изменить его содержимое и из этого объекта создать новый объект *String*. Во многих случаях компилятор выполняет такое преобразование автоматически. Например, следующие фрагменты кода эквивалентны.

```
String s = new StringBuffer("He is ").append(age);
s.append(" years old.").toString();
```

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

В языке Java осуществляется динамическая проверка выхода за границы массивов и строк во время выполнения программы, поэтому невозможно определить значение символа, находящегося после последнего символа в строке.

Наиболее важные методы класса *String*: *length()* — длина строки; *charAt()* — символ в заданной позиции; *compareTo()* — проверка на равенство; *indexOf(), lastIndexOf()* — поиск заданного символа или подстроки; *substring()* — поиск подстроки в строке. Строковые литералы имеют тип *String*, следовательно, можно вызывать методы класса *String* непосредственно для строковых литералов. Например,

```
"Hello".toLowerCase(); // ->"hello"
"Hello".toUpperCase(); // ->"HELLO"
```

Подробное описание классов *String* и *StringBuffer* можно найти в документации по Java API.

Вопросы и упражнения:

1. Создайте и заполните значением 1.0 верхнетреугольную матрицу размером 10x10.
2. Запишите код для вычисления произведения двух матриц размером 20x10 и 10x20.
3. Найдите максимальный элемент в произвольной матрице.

7. Ввод-вывод в консольном режиме

Пример ввода действительного числа.

Обычно приложения на Java используют графический интерфейс для взаимодействия с пользователем. Однако для отладки, а также разработки небольших приложений и решения задач и упражнений из этого пособия Вам понадобится умение вводить и выводить данные в консольном режиме. Рассмотрим пример кода на Java, выполняющего ввод и проверку введенного значения вещественного типа.

```
import java.text.*;
import java.io.*;
import java.util.*;

public class input_test
{
     public static void main(String args[]) throws IOException
     {
          double in_value=0;
          String in_str;
          Number num;
          ParsePosition pp = new ParsePosition(0);
          boolean ok;

          //1
          NumberFormat nf =
               NumberFormat.getInstance(Locale.ENGLISH);
          //2
          BufferedReader in = new BufferedReader(
               new InputStreamReader(System.in)
          );
          do{
               System.out.println("enter a double number:");
               in_str = in.readLine();
               //3
               in_str = in_str.trim();
               ok=true; num=null;
               pp.setIndex(0);
               //4
               num = nf.parse(in_str,pp);
               //5
               if(pp.getIndex()!=in_str.length())
               {
                    ok=false;
                    System.out.println("wrong double format");
               }
          }while(!ok);
```

```
            in_value = num.doubleValue();
            System.out.println("in_str=\""+in_str+"\"");
            System.out.println("in_value="+in_value);
        }
}
```

Для выполнения операций ввода-вывода в Java используется специальный API, представляющий собой группу взаимодействующих классов из пакетов `java.io`, `java.text` и `java.util`. Подробно классы и пакеты объясняются ниже. Не вдаваясь в подробности функционирования классов, рассмотрим назначение операторов в приведенном листинге.

Первое, что необходимо для ввода данных, — это подготовить объект — источник данных. В качестве источника мы используем системный поток ввода `System.in`, который обертывается объектами `InputStreamReader` и `BufferedReader`. В итоге мы получаем объект `in`, в который можно считывать строки, вводимые с консоли (см. //2).

Далее считанную строку мы пытаемся преобразовать в число заданного типа. Для этого мы, во-первых, обрезаем лидирующие и концевые пробелы (см. //3).

Затем мы пытаемся выполнить разбор числа в соответствии с форматом чисел с плавающей точкой, принятом в локале `ENGLISH` (см. //4). Для этого мы построили объект `nf` (см. //1), передали ему нашу строку и еще один объект — `ParsePosition pp`. Последний объект является указателем по строке во время ее преобразования в число. По позиции этого указателя после преобразования мы можем убедиться в его корректности. Если проанализирована вся переданная строка, значит она сдержит число в нужном формате (см. //5).

Таким образом, преобразование включает следующие этапы: конструирование объекта для ввода; конструирование объекта для форматирования чисел и вспомогательного объекта для хранения текущей позиции; чтение введенных данных в строку; обрезка строки; проверка позиции после преобразования.

Вопросы и упражнения:

1. Напишите программы для ввода и проверки введенного значения для всех простых типов данных (для целых чисел используйте настройку `nf.setParseIntegerOnly(true)`).
2. Напишите программу для ввода и проверки длины введенной строки.

8. Классы и объекты

Основы ООП. Понятие класса, метода, объекта, взаимодействия объектов через посылку сообщений. Конструирование объектов. Удаление объектов.

Java — это объектно-ориентированный язык. В нем отсутствуют процедуры, структуры, глобальные переменные. Вместо них используются классы и объекты.

Концепции объектно-ориентированного программирования (ООП) берут начало в языках имитационного моделирования, таких как Simula и Smalltalk. Их идея состоит в том, чтобы рассматривать решаемую задачу как совокупность взаимодействующих между собой объектов. При этом объекту из материального мира соответствует программный объект, иначе некоторый фрагмент кода программы. В качестве объекта может выступать что угодно. Объектом может быть деталь устройства, процесс, и т.д. Иногда выбор объектов естественно следует из решаемой задачи, как в случае моделирования. Иногда для удачной декомпозиции задачи на объекты требуется проанализировать множество вариантов.

В материальном мире объекты не являются статическими. Они меняют свое состояние во времени и взаимодействуют друг с другом. Программные объекты тоже могут хранить свое состояние и взаимодействуют посредством посылки сообщений.

Таким образом, основа объектно-ориентированной концепции — это мышление в терминах объектов и их взаимодействия, а не данных и процедур их обработки. Конечно, вы можете представлять задачу в терминах объектов, а использовать процедурный язык для ее кодирования. Однако синтаксис объектно-ориентированных языков таков, что удобно и выразительно передает семантику объектной модели.

Объектно-ориентированное программирование — это естественное развитие процедурной парадигмы. Поэтому понять его проще, сопоставив с конструкциями процедурных языков программирования. Рассмотрим основные понятия ООП.

Класс — это набор данных и методов, применяемых к этим данным. Данные и методы в совокупности служат для определения содержания и возможностей объекта некоторого вида.

Например, описание класса окружностей на Java может выглядеть следующим образом.

```
public class Circle{
     public double x,y;//координаты центра
     public double r; // радиус
     public double area(){return 3.14159*r*r;}//площадь
}
```

Здесь `x,y,r` — данные, а `area()` — метод класса.

Определение класса соответствует структуре данных, а методы — процедурам, работающим с этими данными. Класс и определяемые в нем данные напоминают структуры или записи в процедурных языках программирования.

С классом окружностей невозможно что-либо делать — для этого требуется конкретная окружность. Необходим экземпляр (instance) класса — объект «окружность».

Определяя класс, мы создаем новый тип данных и тем самым можем объявлять переменные этого типа:

```
Circle c;
```

Но переменная — это только имя, которое может ссылаться на объект. Сам объект должен создаваться динамически. Это (почти) всегда осуществляется при помощи ключевого слова *new*:

```
Circle c;
c = new Circle();
```

Таким образом, создан объект «окружность» и присвоен переменной *c*, имеющей тип *Circle*.

Когда объект создан, можно использовать его переменные-члены:

```
Circle c = new Circle();
c.x = 2.0;// определяем положение
c.y = 2.0;// центра и
c.r = 1.0;// радиус окружности
```

Для доступа к методам применяется тот же синтаксис, что и для доступа к данным объекта.

```
Circle c = new Circle();
double a;
c.r = 2.5;
a = c.area();
```

Обратите внимание на то, что последняя строка содержит код:

```
a = c.area();
//а не
a = area(c);
```

В центре внимания находится объект, а не вызов функции. В синтаксисе языка подразумевается, что метод применяется к экземпляру класса, внутри которого он определен. С точки зрения модели ООП вызов метода — это посылка сообщения с именем метода объекту, указанному в вызове.

Каким же образом работает этот код? Посылка сообщения представляет собой (почти) обычный вызов функции. Только кроме параметров, указанных в вызове, в метод неявно передается ссылка на объект, с которым этот метод работает. В объявлении метода этой ссылки нет. Однако в самом методе к ней можно обратиться явно. Этот неявный аргумент метода называется *this*. А код метода может выглядеть так:

```
public double area(){return 3.14159*this.r*this.r;}//площадь
```

Рассмотрим еще раз, как был создан объект «окружность». В операторе *new* мы использовали имя класса, за которым следовали круглые скобки. Это делает конструирование объекта похожим на вызов функции. В

действительности так оно и происходит. Функция, которая вызывается при конструировании, носит название конструктор. В вышеприведенном примере код конструктора добавляется автоматически. Но можно определить конструктор явно, передать ему аргументы и проинициализировать переменные экземпляра объекта:

```
public class Circle{
      public double x,y,r;
      public Circle(double x,double y,double r){
            this.r=r;this.x=x;this.y;
      }
      public double area(){return 3.14159*r*r;}//площадь
}
```

С новым конструктором инициализация становится частью создания объекта:

```
Circle c = new Circle(1.4,2.0,2.0);
```

В языке Java имеется два важных ограничения на именование и объявление конструкторов:

- имя конструктора всегда совпадает с именем класса;
- всегда неявно подразумевается, что возвращаемый тип является экземпляром класса.

В объявлении конструктора не указывается тип возвращаемого значения и не используется ключевое слово ***void***. Соответственно в конструкторе нельзя использовать оператор ***return*** для возврата значения.

Иногда объект удобно инициализировать разными способами. Это возможно, так как для класса можно определить любое количество конструкторов, различающихся типом и числом аргументов. Если конструкторов несколько, то чтобы не дублировать код, один конструктор может быть вызван из другого. Для этого используется ключевое слово ***this***:

```
public Circle(double x,double y,double r)
{
      this.r=r;this.x=x;this.y;
}

public Circle(double r)
{
      this(0,0,r);
}
```

Так как в Java ссылочные типы не передаются по значению, в результате присваивания одного объекта другому не происходит копирования значения объекта. Оно только присвоит ссылку на объект. Рассмотрим следующий код.

```
Button a=new Button("Okay");
Button b=new Button("Cancel");
a = b;
```

После его выполнения переменная a содержит ссылку на объект, на который ссылается b. Объект, на который ссылалась a, утерян. Для

копирования данных одного объекта в другой следует использовать метод *clone()*.

```
Vector b=new Vector;
c = b.clone();
```

После выполнения этого кода переменная *c* ссылается на объект, являющийся дубликатом объекта, на который ссылается *b*. Следует заметить, что метод *clone()* поддерживается не всеми типами, а только классами, реализующими интерфейс *Cloneable*.

Массивы также принадлежат к ссылочным типам, и при присваивании одного массива другому осуществляется лишь копирование ссылок. Чтобы действительно скопировать значения, хранящиеся в массиве, надо копировать каждый элемент массива отдельно или воспользоваться методом *System.arraycopy()*.

Вследствие передачи объектов по ссылкам оператор == проверяет, на один и тот же объект ссылаются две переменных или нет, а не сравнивает значения переменных. Чтобы проверить, равны ли два объекта между собой, необходимо использовать специально написанный метод для объектов данного типа (подобно методу *strcmp()* для сравнения строк в С). В ряде классов Java для осуществления такой проверки реализован специальный метод *equals()*.

Язык Java чисто объектно-ориентированный. Поэтому возникает вопрос, как в нем моделируются процедуры, глобальные переменные и константы. Ответ заключается в том, что различаются переменные и методы, относящиеся к экземпляру класса и ко всем экземплярам класса. Последние называются статическими и объявляются с ключевым словом *static*. Рассмотрим пример. Пусть требуется подсчитывать количество созданных объектов «окружность».

```
public class Circle{
     public static int num_circles = 0;
     public double x,y,r;
     public Circle(double x,double y,double r){
          this.r=r;this.x=x;this.y; num_circles++;
     }
     public double area(){return 3.14159*r*r;}//площадь
}
```

Для всех объектов, созданных на основе класса *Circle*, существует единственный экземпляр переменной *num_circles*. Поэтому приведенный выше код подсчитывает количество объектов *Circle*.

Если необходимо обратиться к переменной *num_circles* извне определения класса *Circle*, то поступают следующим образом:

```
System.out.println(
     "number of circles created:"+Circle.num_circles
);
```

Таким образом, код работает как обычная глобальная переменная. Однако здесь есть дополнительное преимущество — устраняется возможный

конфликт имен глобальных переменных, потому что на каждую из них ссылаются по имени соответствующего класса.

Если требуется определить константу, то используется еще одно ключевое слово–модификатор *final*.

```
public class Circle{
     public static final double PI = 3.1415;
}
```

Модификатор запрещает изменение значения переменной *PI*, а в данном контексте это соответствует определению константы. Оказывается, что *final*-переменные обрабатываются компилятором как константы. То есть выражения, состоящие из таких переменных и литералов, вычисляются на стадии компиляции, а не на стадии исполнения.

Остается прояснить вопрос с функциями. Как мы видели, методы классов отличаются от функций тем, что содержат неявную ссылку на объект класса. Чтобы метод класса вел себя в точности как функция, нужно определить его с ключевым словом *static*. В этом случае метод работает с классом в целом и, следовательно, передавать ссылку на объект не нужно. Извне класса на такие методы ссылаются по имени класса.

Для инициализации переменных экземпляра существуют конструкторы. Для инициализации переменных класса в Java используются статические инициализаторы *static*:

```
public class Circle{
     public static final double PI = 3.1415;
     static double sines[] = new double[1000];
     static{
          double x, delta_x;
          int i;
          delta_x=(Circle.PI/2)/(1000-1);
          for(i=0,x=0.0;i<1000;i++,x+=delta_x)
               sines[i]=Math.sin(x);
     }
}
```

Код в секции *static* запускается при загрузке класса, поэтому не требуется никаких дополнительных параметров. Внутри класса может быть несколько таких секций.

По аналогии с инициализаторами классов существуют инициализаторы экземпляров. Они выглядят точно также, только не содержат ключевое слово *static*.

Что происходит с объектами, которые становятся ненужными в процессе выполнения программы? Традиционно ставшие ненужными структуры данных (объекты) требуется отслеживать и освобождать занятую ими память. В Java этого делать не нужно, так как существует автоматическая сборка мусора. Система узнает о том, что объект больше не будет использоваться, когда на него не ссылаются переменные. Отслеживаются также циклические ссылки. Периодически в системе запускается процесс сбора мусора и такие объекты удаляются.

Если требуется выполнить какие-то действия при удалении объекта, то используется метод завершения *finalize()*:

```
protected void finalize() throws IOException
{
    if(fd!=null) close();
}
```

Приведенный выше код закрывает файл *fd* при удалении объекта. Невозможно предсказать момент вызова метода завершения. Метод завершения может выбрасывать исключения. Кроме этого существует возможность того, что метод «воскресит объект», запомнив ссылку на него в некоторой переменной. Однако даже в этом случае метод *finalize()* не будет вызван второй раз.

Таким образом, мы познакомились с основами реализации парадигмы ООП в языке Java. Узнали о классах и объектах. Узнали, как моделируются конструкции процедурного языка на Java, не прибегая к введению процедур, глобальных переменных и констант. А также узнали, как смоделировать объекты и методы на процедурном языке.

Вопросы и упражнения:

1. Напишите на Java код класса «окружность», расширив его функциональность другими методами по вашему усмотрению.
2. Смоделируйте поведение класса «окружность» п.1 на Вашем любимом процедурном языке.

9. Сокрытие данных, инкапсуляция, управление областью видимости

Организация кода большой программы. Понятие сокрытия данных, инкапсуляции. Пакеты. Управление областью видимости.

По мере роста размера программы появляется необходимость в организации кода таким образом, чтобы сохранялась его обозримость и не возникало бы путаницы в именах объектов, классов и методов.

Одним из важных приемов ООП, применяемым для решения этой задачи, является сокрытие данных в классе и обеспечение доступа к ним только через методы. Этот прием известен как инкапсуляция (encapsulation). Он получил такое название потому, что в соответствии с ним данные (и внутренние методы) "запечатываются" в "капсулы" класса, откуда они могут быть доступны определенным пользователям, например, методам класса.

Наиболее важным мотивом инкапсуляции является сокрытие подробностей реализации класса. Другой причиной применения инкапсуляции является защита класса от случайных или преднамеренных изменений. Класс часто содержит большое число взаимосвязанных данных,

значения которых должны быть согласованными. Если разрешить программисту (в том числе и самому себе) манипулировать этими данными непосредственно, можно непреднамеренно изменить одну переменную и оставить без изменения другие, связанные с ней переменные, тем самым нарушить согласованность данных. Если для изменения переменной вызывать метод, то детали согласования можно реализовать в нем и в дальнейшем о них не заботиться. Один раз проверив и отладив методы доступа к данным, можно быть уверенным, что класс будет работать как положено.

Как же ограничить доступ к внутренним методам и данным класса? Для управления доступом к методам и данным используют специальные модификаторы доступа: `public`, `protected` `private`. Если модификатор отсутствует, то используется доступ по умолчанию. Эти модификаторы могут применяться к классам, методам классов и данным классов.

Кроме управления доступом на уровне классов в Java имеется еще один уровень — это уровень пакетов. Пакет образует группа связанных между собой и, возможно, согласованно действующих классов. Все члены классов пакета, не объявленные как `private`,видимы во всех классах пакета. Это верно, поскольку предполагается, что классы знают друг о друге и доверяют друг другу. Чтобы включить классы из некоторого файла в пакет,используется оператор `package`. Классы, принадлежащие разным пакетам, будут доступными друг для друга, если они объявлены с модификатором `public`.

В приведенной ниже таблице перечислены условия, при которых члены различных типов видимости доступны для других классов.

Доступен для:				
	Public	Protected	По умолчанию	Private
Того же класса	Да	Да	Да	Да
Класса из того же пакета	Да	Да	Да	Нет
Подклассов из того же пакета	Да	Да	Нет	Нет
Подклассов из других пакетов	Да	Нет	Нет	Нет

Подробности видимости членов классов в Java могут показаться весьма запутанными. Вот несколько простых правил использования модификаторов видимости.

Применяйте модификатор `public` для тех методов и констант, которые входят в состав открытого API класса. Некоторые очень важные или часто используемые переменные могут также объявляться как `public`, но общепринятым является объявлять переменные как не-`public` и инкапсулировать их в `public`-методах (см. пример ниже).

Применяйте модификатор `protected` для переменных и методов, которые необязательно используются в данном классе, но могут понадобиться при создании подкласса, входящего в другой пакет.

Используйте тип видимости по умолчанию для переменных и методов, которые не должны быть видны вне пакета, но к которым желательно иметь доступ из классов того же пакета, взаимодействующих с данным классом.

Применяйте модификатор ***private*** для переменных и методов, которые используются только внутри данного класса и должны быть скрыты для остальных.

В заключение рассмотрим, как можно организовать сокрытие переменных на примере класса окружности ***Circle***.

```
package shapes;
public class Circle{
     protected double x,y;//координаты центра
     protected double r; // радиус - видимы в подклассах Circle
     private static final double MAXR = 100.0;
     // максимальный радиус (постоянное значение)
     private boolean check_radius(double r){return (r<=MAXR);}
     // public-конструкторы
     public Circle(double x, double y, double r)
     {
          this.x=x; this.y;
          if(check_radius(r))this.r=r;else this.r=MAXR;
     }
     public Circle(double r){ this(0.0,0.0,r); }
     public Circle(){ this(0.0,0.0,1.0); }
     // public-методы для доступа к данным
     public double area(){return 3.14*r*r;}
     public double circumference(){return 2*3.14*r;}
     public void moveto(double x, double y){this.x=x;this.y=y;}
     public void move(double dx, double dy)
          {this.x+=dx;this.y+=dy;}
     public void setRadius(double r)
          {this.r=(check_radius()}?r:MAXR;}
     // методы объявлены как final для того чтобы компилятор мог
     //выполнить inline-подстановку
     public final double getX(){return x;}
     public final double getY(){return y;}
     public final double getRadius(){return r;}
}
```

Вопросы и упражнения:

1. В чем преимущества и недостатки непосредственного доступа к данным класса? Когда следует использовать доступ к данным через методы, а когда — через непосредственный доступ?
2. Перечислите случаи, когда в Java один объект может получить доступ к данным другого объекта.

10. Наследование

Цель, реализация, особенности наследования в Java. Особенности конструирования и удаления объектов. Сокрытие переменных и обращение к переменным суперклассов.

Разработанный ранее класс `Circle` может применяться для абстрактных математических действий. Если же будет нужно рисовать окружность на экране, то потребуется новый класс `GraphicCircle`. Он должен реализовывать все функции класса `Circle`, но при этом обладать возможностью нарисовать окружность. Хотелось бы реализовать новый класс так, чтобы он использовал код, написанный для класса `Circle`. Предполагается, что класс `Circle` разрабатывался отдельно, и мы не можем расширить его функциональность, просто добавив новые методы.

Для этого существует два способа. Вот пример первого способа, который называется "композиция".

```
public class GraphicCircle{
     public Circle c;
     // Это старые методы
     public double area(){return c.area();}
     public double circumference(){return c.circumference();}
     // Это новые переменные и методы
     public Color outline, fill;
     public void draw(DrawWindow dw){/* код опущен*/}
}
```

Этот код будет работать, однако проблема заключается в том, что приходится писать переходники для методов `area()`, `circumference()` и других из класса `Circle`. Если же работать с переменной `Circle c` напрямую, то нарушается принцип инкапсуляции.

В Java существует другой способ решения проблемы — использовать механизм наследования и объявить класс `GraphicCircle` как расширение или подкласс класса `Circle`.

```
public class GraphicCircle extends Circle{
     // Переменные и методы класса Circle наследуются
     // автоматически, поэтому необходимо указать только
     // новые элементы. Конструктор пока опущен
     public Color outline, fill;
     public void draw(DrawWindow dw){
          dw.drawCircle(x,y,r,outline,fill);
     }
}
```

Ключевое слово `extends` говорит Java о том, что `GraphicCircle` подкласс `Circle` и наследует переменные и методы класса. В определении метода `draw()` прослеживается наследование переменных. Этот метод использует переменные `x,y` и `r` класса `Circle`, так как если бы они были

определены в самом классе *GraphicCircle*. *GraphicCircle* также наследует и методы *Circle*.

Таким образом, если переменная *gc* ссылается на объект типа *GraphicCircle*, справедливо выражение: *double area = gc.area();*.

Другая особенность подклассов заключается в том, что объект класса *GraphicCircle* также является объектом класса *Circle*. Можно выполнить присваивание *Circle c = gc;* и работать с объектом как экземпляром класса *Circle*, не имеющим графических возможностей.

Механизм наследования в Java имеет такие особенности. Во-первых, все классы образуют единую иерархию наследования, так как все они наследуют от класса *Object*. Если наследование не используется, то новый класс неявно считается потомком класса *Object*. Во-вторых, наследовать свойства и методы новый класс может только от одного суперкласса. Таким образом, иерархия наследования всех классов Java представляет собой дерево с корнем *Object*. Заметим, что в языке C++ нет единой иерархии, но используется множественное наследование.

В приведенном выше примере был опущен код конструктора класса. Вот один из способов реализации конструктора.

```
public GraphicCircle(double x, double y, double r,
Color outline, Color fill)
{
     this.x = x; this.y = y; this.r = r;
     this.outline = outline; this.fill = fill;
}
```

В этой реализации конструктора происходит дублирование кода инициализации переменных *x*, *y* и *r*, унаследованных от класса *Circle*. Если бы в конструкторах производилась более сложная инициализация, такое дублирование стало бы слишком расточительным. Дублирование кода можно устранить, если вызвать конструктор суперкласса.

```
public GraphicCircle(double x, double y, double r,
Color outline, Color fill)
{
     super(x,y,r);
     this.outline = outline; this.fill = fill;
}
```

Применение ключевого слова super в этом примере аналогично применению слова *this* для вызова одного конструктора из другого. Правила применяемые для *super*, похожи на правила для *this*:

- ключевое слово *super* может использоваться для вызова конструктора только в теле конструктора;
- вызов конструктора суперкласса должен быть первым оператором в теле конструктора, он должен располагаться даже до объявления локальных переменных конструктора.

При инициализации объектов необходимо, чтобы каждый конструктор вызывал конструктор своего суперкласса. Если в качестве первого оператора

теле конструктора не используется явный вызов конструктора суперкласса, то Java вставляет вызов *super()* неявно (вызывает конструктор суперкласса без аргументов). Если первой строкой конструктора является вызов другого конструктора оператором *this()*, то неявный вызов конструктора суперкласса происходит в другом конструкторе. Если в классе *GraphicCircle* не объявлять конструктор, то будет добавлен конструктор по умолчанию: *public GraphicCircle(){super();}*.

В любом случае при конструировании объектов происходит исполнение цепочки конструкторов, начинающейся с вызова конструктора класса *Object*, включающей вызов конструкторов всей цепочки наследования для класса конструируемого объекта.

В Java имеется возможность разрешать или запрещать конструирование объектов. В классах, для которых необходимо запретить создание экземпляров вообще, должен быть определен *private*-конструктор. В тех классах, для которых нежелательно предоставлять возможность открыто создавать экземпляры класса, следует объявить *protected*-конструктор. Эти объявления должны перекрывать *public*-конструкторы, вставляемые по умолчанию.

Если при конструировании объектов происходит вызов цепочки конструкторов, то можно предположить, что при освобождении памяти происходит вызов цепочки методов завершения. Однако это не так. Цепочку методов завершения приходится строить в явном виде. Вызов метода *finalize()* для суперкласса обычно помещается в конце метода завершения и выглядит так: *super.finalize()*.

Итак, нам удалось реализовать повторное использование кода при создании класса *GraphicCircle* на основе класса *Circle*. Что будет, если имена новых переменных в *GraphicCircle* окажутся совпадающими с именами из класса *Circle*? Рассмотрим пример. Допустим, в классе объявлена новая переменная *r*, которая обозначает разрешение (в точках на дюйм) для объекта *DrawWindow*.

```
public class GraphicCircle extends Circle{
     public Color outline, fill;
     float r; // новая переменная скрывает r из класса Circle
     public void draw(DrawWindow dw)
          {dw.drawCircle(x,y,r,outline,fill);}
     public void setResolution(float resolution)
          {r = resolution;}
}
```

Теперь переменная *r* больше не соответствует радиусу окружности. Переменная разрешения *r* в *GraphicCircle* скрывает переменную радиуса *r* в Circle.

Переменная радиуса, тем не менее, в классе *GraphicCircle* присутствует. Существует несколько способов сослаться на такую переменную.

Если требуется сослаться на переменную с таким же именем из суперкласса, то используется синтаксис `super.x`. Если переменная находится глубже, чем в суперклассе в иерархии наследования, то применять выражения типа `super.super.x` нельзя. Общая форма для обращения к переменной, определенной в конкретном классе иерархии, выглядит так: `((Circle)this).x`. То есть следует использовать приведение типа для ссылки `this`.

Если в классе определяется метод с тем же именем, типом возвращаемого значения и типом и порядком следования аргументов, что и метод суперкласса, то говорят о переопределении (override) метода суперкласса. На переопределении методов основан важный механизм ООП — полиморфное поведение объектов, который рассматривается в следующем параграфе.

Вопросы и упражнения:

1. Предположим, что требуется написать проект на языке, который не является объектно-ориентированным. Как бы вы имитировали классы и методы? Как бы вы имитировали наследование?
2. Рассмотрите три геометрических понятия: линия (бесконечна в обоих направлениях), луч (начало в фиксированной точке, бесконечен в одном направлении), сегмент (отрезок прямой с фиксированными концами). Как бы вы построили классы, представляющие эти понятия, в виде иерархии наследования? Будет ли ваше решение другим, если вы обратите особое внимание на представление данных (поведение объектов)?

11. Реализация полиморфного поведения объектов

Различие между переопределением и перегрузкой методов. Понятие полиморфизма. Статический и динамический тип. Динамический поиск метода. Как отменить динамический поиск метода.

Рассмотрим, как работает класс, у которого имеются методы с одинаковыми именами. Пример такого класса нам встречался ранее: у класса `Circle` нами было определено несколько конструкторов с разными именами. То же самое можно делать и с обычными методами класса. Любые два метода воспринимаются как различные, если у них не совпадает один из следующих признаков: имена, количество аргументов, типы аргументов или порядок расположения аргументов в списке.

Определение методов с одинаковыми именами, но разными типами аргументов называется перегрузкой методов (method overloading). Такая перегрузка удобна, когда методы с одинаковыми именами выполняют сходные действия над несколько отличающимися типами входных данных.

Перегруженные методы могут возвращать значения различных типов, но лишь при условии, что им передаются аргументы разных типов.

Перегрузку методов следует отличать от их переопределения. Когда в классе определяется метод, использующий то же имя, тип возвращаемого значения и аргументы, что и метод суперкласса, метод класса переопределяет (override) метод суперкласса. Если теперь вызвать этот метод для объекта класса, то будет вызван переопределенный, а не старый метод суперкласса. Рассмотрим пример.

```
class A {
     int i = 1;
     int f() {return i;}
}

class B extends A {
     int i = 2; // скрытая переменная i класса A
     int f() {return -i;} // переопределенный метод f
                          / /класса A
}

public class override_test {
     public static void main(String args[])
     {
          B b = new B();
          System.out.println(b.i); // ссылается на B.i
                                   // выводится 2
          System.out.println(b.f()); // ссылается на B.f()
                                   // выводится -2

          A a = (A) b;          // приведение b к типу A
          System.out.println(a.i);// ссылается на A.i
                                  //выводится 1
          System.out.println(a.f());// ссылается на B.f()
                                  // выводится -2
     }
}
```

Переопределение метода — это прием, служащий для реализации полиморфного поведения объектов. В языках программирования полиморфный объект — это сущность (переменная или аргумент функции), хранящая во время выполнения программы значения различных типов. В данном примере переменная `a` хранит значение класса `B`. В языке Java любая переменная может хранить значение типа, являющегося подтипом ее типа. То есть любая переменная может быть полиморфной. Полиморфизм — это мощный механизм абстрагирования, позволяющий писать код, использующий общие свойства по-разному реализованных объектов.

Как же используется это свойство. Допустим, имеется набор геометрических объектов (окружностей, прямоугольников, треугольников и т.п.). Требуется определить общую площадь и периметр для объектов из этой группы. Обычный подход требует написания отдельной функции для определения площадей и периметров всех окружностей, прямоугольников

треугольников. Однако свойства "иметь площадь" и "иметь периметр" — общие для всех рассматриваемых объектов. Поэтому для решения задачи определения площади и периметра следует поступать так. Нужно определить общий суперкласс для геометрических объектов, например **`Shape`**. В нем определить функции для вычисления площади и периметра. В каждом из классов геометрических объектов нужно выполнить переопределение методов, вычисляющих площадь и периметр. Каждый из классов будет иметь реализацию методов соответствующую типу фигуры. Далее нужно реализовать функцию, вычисляющую периметр и площадь для массива объектов типа **`Shape`**. Элементами массива могут быть любые геометрические объекты, наследующие от класса **`Shape`**.

Как система определяет, какой именно метод должен быть вызван? Дело в том, что у переменной имеется два типа. Первый тип статический, он задается при объявлении переменной. Второй тип — динамический. Он зависит от типа значения, которое в действительности содержится в переменной. Вызов переопределенных методов выполняется в соответствии с динамическим типом переменной. Динамический тип не может быть определен на этапе компиляции. Поэтому определение вызываемого метода происходит на этапе выполнения программы. Этот процесс называется динамическим поиском метода. Динамический поиск метода в Java выполняется достаточно быстро, но не так быстро, как в случае вызова обычной процедуры. По умолчанию для всех методов используется динамический поиск. (Для программистов на C++ такое поведение методов выглядит так, как если бы они были определены с ключевым словом virtual). Избежать динамического поиска можно, если указать компилятору, что данный метод никогда не будет переопределен. Это можно сделать несколькими способами. Если метод объявлен с модификатором **`final`**, значит он объявлен окончательно и не будет переопределяться. Методы, объявленные как **`static`** и **`private`**, как и все методы класса, объявленного с модификатором **`final`**, также не могут быть переопределены. Когда метод не может быть переопределен, компилятор выполняет оптимизацию вызова.

Вопросы и упражнения:

1. В чем заключается разница между перегрузкой и переопределением методов?
2. Приведите пример перегруженных операций в Java.
3. Сравните накладные расходы на вызов статического метода, нестатического финального метода и переопределенного метода.

12. Интерфейсы, абстрактные классы и методы

Абстрактные классы и их свойства. Использование абстрактных классов. Использование интерфейсов. Константы в интерфейсах. Расширение интерфейсов. Интерфейсы-метки.

Рассмотрим еще раз пример с определением площадей и периметров фигур из предыдущего параграфа. Для реализации механизма переопределения класс геометрических фигур `Shape` должен иметь методы вычисления площади и периметра. Но родовой класс `Shape` в действительности не может реализовать эти методы, так как он не представляет ни одной действительной фигуры. В таких случаях используются абстрактные методы.

Java позволяет определять метод без реализации, объявляя его с модификатором `abstract`. Такой метод не имеет тела: за его определением следует точка с запятой. В Java определены следующие правила для абстрактных методов и классов:

1) Любой класс, включающий абстрактный метод, сам становится абстрактным; абстрактный класс должен содержать хотя бы один абстрактный метод.

2) Класс может быть объявлен с модификатором `abstract`, даже если он не содержит абстрактных методов. Этим гарантируется невозможность создания экземпляров такого класса.

3) Абстрактный класс не может иметь экземпляров.

4) Подкласс абстрактного класса может иметь экземпляры, если в нем переопределяются все абстрактные методы.

5) Если подкласс не реализует (переопределяет) все унаследованные абстрактные методы, то он сам становится абстрактным.

Теперь мы можем продемонстрировать, как на самом деле должен выглядеть пример с определением периметров и площадей фигур.

```
public abstract class Shape{
     public abstract double area();
     public abstract double circumference();
}

class Circle extends Shape{
     protected double r;
     protected static final double PI = 3.1415;
     public Circle(){r=1.0;}
     public Circle(double r){this.r=r;}
     public double area(){return PI*r*r;}
     public double circumference(){return 2*PI*r;}
     public double getRadius(){return r;}
}

class Rectangle extends Shape{
```

```
    protected double w,h;
    public Rectangle(){w=0.0; h=0.0;}
    public Rectangle(double w, double h){this.w=w;this.h=h;}
    public double area(){return w*h;}
    public double circumference(){return 2*(w+h);}
    public double getWidth(){return w;}
    public double getHeight(){return h;}
}

Shape[] shapes = new Shape[3];
shapes[0] = new Circle(2.0);
shapes[1] = new Rectangle(1.0,3.0);
shapes[2] = new Rectangle(4.0,2.0);

double total_area = 0;
for(int i=0;i<shapes.lenght;i++)
    total_area += shapes[i].area();
```

Следует обратить внимание на два важных момента.

1) Подклассы класса ***Shape*** могут быть присвоены элементам любого массива типа ***Shape***. То есть в приведении типа нет необходимости.

2) Методы ***area()*** и ***circumference()*** можно вызывать для объектов ***Shape***, даже если в классе не определены тела этих методов (если они объявлены как ***abstract***). Когда эти методы не объявлены в ***Shape*** вообще, то возникнет ошибка компиляции.

Часто возникает ситуация, когда требуется наследовать свойства сразу от нескольких классов. Допустим, мы хотим добавить возможность отображения на экране для разработанных классов графических объектов. Решение этой проблемы в Java осуществляется с помощью интерфейса. Во многом интерфейс похож на абстрактный класс, за исключением использования ключевого слова ***interface*** вместо ключевых слов ***abstract*** и ***class***. В интерфейсе все методы неявно определены как абстрактные. Ниже приводится описание интерфейса для отображаемых графических объектов.

```
public interface Drawble{
    public void setColor(Color c);
    public void setPosition(double x, double y);
    public void draw(DrawWindow dw);
}
```

Объявленный интерфейс используется следующим образом. Так же, как класс расширяет свой суперкласс, класс может реализовывать один или несколько интерфейсов. Например, класс ***DrawbleRectangle*** расширяет класс ***Rectangle*** и реализует интерфейс ***Drawble***.

```
public class DrawbleRectangle extends Rectangle
    implements Drawble
{
    private Color c;
    private double x,y;
    public DrawbleRectangle(double w, double h){super(w,h);}
```

```
     public void setColor(Color c){this.c=c;}
     public void setPosition(double x, double y)
{this.x=x;this.y=y;}
     public void draw(DrawWindow dw)
          {dw.drawRect(x,y,h,c);}
}
```

Предположим, что классы *DrawbleCircle* и *DrawbleSquare* реализуются аналогично классу *DrawbleRectangle*. Экземпляры этих классов могут рассматриваться как экземпляры абстрактного класса *Shape*. Они также могут считаться экземплярами интерфейса *Drawble*, как показано ниже.

```
Shape[] shapes = new Shape[3]; // создание массива фигур
Drawble[] drawbles = new Drawble[3]; // и рисуемых объектов
// создание нескольких рисуемых фигур
DrawbleCircle dc = new DrawbleCircle(1.1);
DrawbleSquare ds = new DrawbleSquare(2.1);
DrawbleRectangle dr = new DrawbleRectangle(2.3, 4.2);
// созданные фигуры могут быть присвоены обоим массивам
shapes[0]=dc; drawbles[0]=dc;
shapes[1]=ds; drawbles[1]=ds;
shapes[2]=dr; drawbles[2]=dr;
// вычисление общей площади и рисование фигуры с помощью
// вызова абстрактных методов классов Shape и Drawble
double total_area = 0;
for(int i=0; i<shapes.length; i++){
     total_area += shapes[i].area;
     drawbles[i].setPosotion(i*10.0, i*10.0);
     drawbles[i].draw(draw_window); // draw_window
//уже где-то определена
}
```

Аналогично классам интерфейсы могут иметь подинтерфейсы. Однако интерфейсы отличаются от классов одной важной особенностью: интерфейс может наследовать несколько интерфейсов одновременно.

В определениях интерфейсов могут использоваться константы (переменные, объявленные с модификаторами *static final*). Класс, реализующий такие интерфейсы, наследует константы и может использовать их, как если бы они были определены в самом классе.

Другая методика, используемая в Java, — определение совершенно пустых интерфейсов. Такие интерфейсы называются интерфейсами-метками. Класс может реализовывать интерфейс-метку с целью предоставления дополнительной информации о себе. Примером является интерфейс *Clonable* из пакета *java.lang*. Он позволяет определить, можно ли создать копию объекта с использованием метода *clone()*. Проверить, реализует ли данный объект интерфейс, можно с использованием оператора *instanceof*.

Таким образом, интерфейсы являются элегантным решением проблемы множественного наследования в языке Java.

Вопросы и упражнения:

1. Приведите примеры множественного наследования в ситуациях, не связанных с компьютерами.
2. Пусть определен класс `IntegerArray`, который наследует от классов `Integer` и `Array`. Является ли это хорошим примером множественного наследования? Обоснуйте ответ.

13. Структурированное управление исключениями

Типы исключений. Неперехваченные исключения. Операторы try и catch. Несколько разделов catch. Вложенные операторы try. Оператор throw, ключевое слово throws. Оператор finally. Подклассы Exception.

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

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

Ниже приведена общая форма блока обработки исключений.

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

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

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

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

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

Вот вывод, полученный при запуске нашего примера.

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

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

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

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

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

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

```
class Exc2 {
     public static void main(String args[]) {
          try {
               int d = 0;
               int a = 42 / d;
          }
          catch (ArithmeticException e) {
```

```
            System.out.println("division by zero");
        }
    }
}
```

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

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

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

Этот пример, запущенный без параметров, вызывает возбуждение исключительной ситуации деления на нуль. Если же мы зададим в командной строке один или несколько параметров, тем самым установив *a* в значение больше нуля, наш пример переживет оператор деления, но в следующем операторе будет возбуждено исключение выхода индекса за границы массива *ArrayIndexOutOfBounds*. Ниже приведены результаты работы этой программы, запущенной и тем,и другим способом.

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

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

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

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

throw ОбъектТипаThrowable;

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

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

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

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

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

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

```
class ThrowsDemo1 {
     static void procedure() {
          System.out.println("inside procedure");
          throw new IllegalAccessException("demo");
```

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

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

```
class ThrowsDemo {
    static void procedure() throws IllegalAccessException {
        System.out.println(" inside procedure");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[]) {
        try {
            procedure();
        }
        catch (IllegalAccessException e) {
            System.out.println("caught " + e);
        }
    }
}
```

Ниже приведен результат выполнения этой программы.

```
C:\> java ThrowsDemo
inside procedure
caught java.lang.IllegalAccessException: demo
```

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

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

```
    static void procB() {
        try {
            System.out.println("inside procB");
            return;
        }
        finally {
            System.out.println("procB's finally");
        }
    }
    public static void main(String args[]) {
        try {
            procA();
        }
        catch (Exception e) {}
        procB();
    }
}
```

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

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

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

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

14. Понятие о многопоточном программировании. Синхронизация потоков

Модель легковесных процессов в Java. Класс Thread, интерфейс Runnable. Приоритеты подпроцессов. Взаимодействие подпроцессов. Тупик (deadlock) и состояние гонки (race condition).

Параллельное программирование, связанное с использованием легковесных процессов или подпроцессов (multithreading, light-weight processes) — концептуальная парадигма, в которой вы разделяете свою программу на два или несколько процессов, которые могут исполняться одновременно. Во многих средах параллельное выполнение заданий представлено в том виде, который в операционных системах называется многозадачностью. Это совсем не то же самое, что параллельное выполнение подпроцессов. В многозадачных операционных системах вы имеете дело с полновесными процессами, в системах с параллельным выполнением подпроцессов отдельные задания называются легковесными процессами (light-weight processes, threads).

Исполняющая система Java во многом зависит от использования подпроцессов, и все ее классовые библиотеки написаны с учетом особенностей программирования в условиях параллельного выполнения подпроцессов. Java использует подпроцессы для того, чтобы сделать среду программирования асинхронной. После того, как подпроцесс запущен, его выполнение можно временно приостановить (`suspend`). Если подпроцесс остановлен (`stop`), возобновить его выполнение невозможно.

Приоритеты подпроцессов — это просто целые числа в диапазоне от 1 до 10. Имеют смысл только соотношения приоритетов различных подпроцессов. Приоритеты же используются для того, чтобы решить, когда нужно остановить один подпроцесс и начать выполнение другого. Это называется ***переключением контекста***. Правила просты. Подпроцесс может добровольно отдать управление — с помощью явного системного вызова или при блокировании на операциях ввода-вывода, либо он может быть приостановлен принудительно. В первом случае проверяются все остальные подпроцессы, и управление передается тому из них, который готов к выполнению и имеет самый высокий приоритет. Во втором случае, низкоприоритетный подпроцесс, независимо от того, чем он занят, приостанавливается принудительно для того, чтобы начал выполняться подпроцесс с более высоким приоритетом.

Поскольку подпроцессы вносят в ваши программы асинхронное поведение, должен существовать способ их синхронизации. Для этой цели в Java реализовано элегантное развитие старой модели синхронизации процессов с помощью ***монитора***.

```
    public static void main(String args[]) {
        try {
            compute(1);
            compute(20);
        }
        catch (MyException e) {
            System.out.println("caught" + e);
        }
    }
}
```

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

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

Обработка исключений предоставляет мощный механизм для управления сложными программами. *Try*, *throw*, *catch* дают простой и ясный путь для встраивания обработки ошибок и прочих нештатных ситуаций в программную логику.

Вопросы и упражнения:

1. Какие методы обработки ошибок, помимо механизма исключений, вы знаете? Сравните их со структурированной обработкой исключений.
2. Напишите цикл, вычисляющий сумму элементов массива, не проверяя его размера.

Коль скоро вы разделили свою программу на логические части — подпроцессы, вам нужно аккуратно определить, как эти части будут общаться друг с другом. Java предоставляет для этого удобное средство. Два подпроцесса могут "общаться" друг с другом, используя методы *wait* и *notify*. Работать с параллельными подпроцессами в Java несложно. Язык предоставляет явный, тонко настраиваемый механизм управления созданием подпроцессов, переключения контекстов, приоритетов, синхронизации и обмена сообщениями между подпроцессами.

Класс *Thread* инкапсулирует все средства, которые могут вам потребоваться при работе с подпроцессами. При запуске Java-программы в ней уже есть один выполняющийся подпроцесс. Вы всегда можете выяснить, какой именно подпроцесс выполняется в данный момент с помощью вызова статического метода *Thread.currentThread*. После того как вы получите дескриптор подпроцесса, вы можете выполнять над этим подпроцессом различные операции даже в том случае, когда параллельные подпроцессы отсутствуют. В очередном нашем примере показано, как можно управлять выполняющимся в данный момент подпроцессом.

```
class CurrentThreadDemo {
     public static void main(String args[]) {
          Thread t = Thread.currentThread();
          t.setName("My Thread");
          System.out.println("current thread: " + t);
          try {
               for (int n = 5; n > 0; n--) {
                    System.out.println(" " + n);
                    Thread.sleep(1000);
               }
          }
          catch (InterruptedException e) {
               System.out.println("interrupted");
          }
     }
}
```

В этом примере текущий подпроцесс хранится в локальной переменной *t*. Затем мы используем эту переменную для вызова метода *setName*, который изменяет внутреннее имя подпроцесса на *"My Thread"* с тем, чтобы вывод программы был удобочитаемым. На следующем шаге мы входим в цикл, в котором ведется обратный отсчет от 5, причем на каждой итерации с помощью вызова метода *Thread.sleep()* делается пауза длительностью в 1 секунду. Аргументом для этого метода является значение временного интервала в миллисекундах, хотя системные часы на многих платформах не позволяют точно выдерживать интервалы короче 10 миллисекунд. Обратите внимание — цикл заключен в *try/catch* блок. Дело в том, что метод *Thread.sleep()* может возбуждать исключение *InterruptedException*. Это исключение возбуждается в том случае, если какому-либо другому подпроцессу понадобится прервать данный подпроцесс. В данном примере

мы в такой ситуации просто выводим сообщение о перехвате исключения. Ниже приведен вывод этой программы:

```
C:\> java CurrentThreadDemo
current thread: Thread[My Thread,5,main]
5
4
3
2
1
```

Обратите внимание на то, что в текстовом представлении объекта *Thread* содержится заданное нами имя легковесного процесса — *My Thread*. Число 5 — это приоритет подпроцесса, оно соответствует приоритету по умолчанию, *"main"* — имя группы подпроцессов, к которой принадлежит данный подпроцесс.

Не очень интересно работать только с одним подпроцессом, а как можно создать еще один? Для этого нам понадобится другой экземпляр класса *Thread*. При создании нового объекта *Thread* ему нужно указать, какой программный код он должен выполнять. Вы можете запустить подпроцесс с помощью любого объекта, реализующего интерфейс *Runnable*. Для того чтобы реализовать этот интерфейс, класс должен предоставить определение метода *run*. Ниже приведен пример, в котором создается новый подпроцесс.

```
class ThreadDemo implements Runnable {
    ThreadDemo() {
        Thread ct = Thread.currentThread();
        System.out.println("currentThread: " + ct);
        Thread t = new Thread(this, "Demo Thread");
        System.out.println("Thread created: " + t);
        t.start();
        try {
            Thread.sleep(3000);
        }
        catch (InterruptedException e) {
            System.out.println("interrupted");
        }
        System.out.println("exiting main thread");
    }
    public void run() {
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println("" + i);
                Thread.sleep(1000);
            }
        }
        catch (InterruptedException e) {
            System.out.println("child interrupted");
        }
        System.out.println("exiting child thread");
    }
    public static void main(String args[]) {
        new ThreadDemo();
```

```
    }
}
```

Обратите внимание на то, что цикл внутри метода `run` выглядит точно так же, как и в предыдущем примере, только на этот раз он выполняется в другом подпроцессе. Подпроцесс `main` с помощью оператора `new Thread(this "Demo Thread")` создает новый объект класса `Thread`, причем первый параметр конструктора — `this` — указывает, что нам хочется вызвать метод `run` текущего объекта. Затем мы вызываем метод `start`, который запускает подпроцесс, выполняющий метод `run`. После этого основной подпроцесс (`main`) переводится в состояние ожидания на три секунды, затем выводит сообщение и завершает работу. Второй подпроцесс — `"Demo Thread"` — при этом по-прежнему выполняет итерации в цикле метода `run` до тех пор, пока значение счетчика цикла не уменьшится до нуля. Ниже показано, как выглядит результат работы этой программы после того, как она отработает 5 секунд.

```
C:\> java ThreadDemo
Thread created: Thread[Demo Thread,5,main]
5
4
3
exiting main thread
2
1
exiting child thread
```

Если вы хотите добиться от Java предсказуемого независимого от платформы поведения, вам следует проектировать свои подпроцессы таким образом, чтобы они по своей воле освобождали процессор. Ниже приведен пример с двумя подпроцессами с различными приоритетами, которые не ведут себя одинаково на различных платформах. Приоритет одного из подпроцессов с помощью вызова `setPriority` устанавливается на два уровня выше `Thread.NORM_PRIORITY`, то есть умалчиваемого приоритета. У другого подпроцесса приоритет, наоборот, на два уровня ниже. Оба эти подпроцесса запускаются и работают в течение 10 секунд. Каждый из них выполняет цикл, в котором увеличивается значение переменной-счетчика. Через десять секунд после их запуска основной подпроцесс останавливает их работу, присваивая условию завершения цикла `while` значение `true`, и выводит значения счетчиков, показывающих, сколько итераций цикла успел выполнить каждый из подпроцессов.

```
class Clicker implements Runnable {
     int click = 0;
     private Thread t;
     private boolean running = true;
     public clicker(int p) {
          t = new Thread(this);
          t.setPriority(p);
     }
     public void run() {
```

```
            while (running) {
                click++;
            }
        }
        public void stop() {running = false; }
        public void start() {t.start();}
}
class HiLoPri {
        public static void main(String args[]) {
            Thread.currentThread().
                setPriority(Thread.MAX_PRIORITY);
            clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
            clicker lo = new clicker(Thread.NORM_PRIORITY - 2);
            lo.start();
            hi.start();
            try Thread.sleep(-10000);
            catch (Exception e) {}
            lo.stop();
            hi.stop();
            System.out.println(lo.click + " vs. " + hi.click);
        }
}
```

По значениям, фигурирующим в распечатке, можно заключить, что подпроцессу с низким приоритетом достается меньше на 25 процентов времени процессора:

```
C:\>java HiLoPri
304300 vs. 4066666
```

Когда двум или более подпроцессам требуется параллельный доступ к одним и тем же данным (иначе говоря, к совместно используемому ресурсу), нужно позаботиться о том, чтобы в каждый конкретный момент времени доступ к этим данным предоставлялся только одному из подпроцессов. Java для такой синхронизации предоставляет уникальную, встроенную в язык программирования поддержку. В других системах с параллельными подпроцессами существует понятие ***монитора***. Монитор — это объект, используемый как защелка. Только один из подпроцессов может в данный момент времени владеть монитором. Когда подпроцесс получает эту защелку, говорят, что он ***вошел*** в монитор. Все остальные подпроцессы, пытающиеся войти в тот же монитор, будут заморожены до тех пор, пока подпроцесс-владелец не выйдет из монитора.

У каждого Java-объекта есть связанный с ним неявный монитор, а для того чтобы войти в него, надо вызвать метод этого объекта, отмеченный ключевым словом ***synchronized***. Для того чтобы выйти из монитора и тем самым передать управление объектом другому подпроцессу, владелец монитора должен всего лишь вернуться из синхронизированного метода.

```
class Callme {
    void call(String msg) {
        System.out.println("[" + msg);
        try Thread.sleep(-1000);
        catch(Exception e) {}
```

```
            System.out.println("]");
        }
}
class Caller implements Runnable {
        String msg;
        Callme target;
        public Caller(Callme t, String s) {
            target = t;
            msg = s;
            new Thread(this).start();
        }
        public void run() {
            target.call(msg);
        }
}
class Synch {
        public static void main(String args[]) {
            Callme target = new Callme();
            new Caller(target, "Hello.");
            new Caller(target, "Synchronized");
            new Caller(target, "World");
        }
}
```

Вы можете видеть из приведенного ниже результата работы программы, что `sleep` в методе `call` приводит к переключению контекста между подпроцессами, так что вывод наших 3 строк-сообщений перемешивается:

```
[Hello.
[Synchronized
]
[World
]
]
```

Это происходит потому, что в нашем примере нет ничего, способного помешать разным подпроцессам вызывать одновременно один и тот же метод одного и того же объекта. Для такой ситуации есть даже специальный термин — ***race condition*** (состояние гонки), означающий, что различные подпроцессы пытаются опередить друг друга, чтобы завершить выполнение одного и того же метода. В этом примере для того чтобы это состояние было очевидным и повторяемым, использован вызов `sleep`. В реальных же ситуациях это состояние, как правило, трудноуловимо, поскольку непонятно, где именно происходит переключение контекста, и этот эффект менее заметен и не всегда воспроизводится от запуска к запуску программы. Так что если у вас есть метод (или целая группа методов), который манипулирует внутренним состоянием объекта, используемого в программе с параллельными подпроцессами, во избежание состояния гонки вам следует использовать в его заголовке ключевое слово `synchronized`.

В Java имеется элегантный механизм общения между подпроцессами, основанный на методах `wait`, `notify` и `notifyAll`. Эти методы реализованы, как `final`-методы класса `Object`, так что они имеются в любом Java-классе.

```
            System.out.println("]");
        }
}
class Caller implements Runnable {
        String msg;
        Callme target;
        public Caller(Callme t, String s) {
            target = t;
            msg = s;
            new Thread(this).start();
        }
        public void run() {
            target.call(msg);
        }
}
class Synch {
        public static void main(String args[]) {
            Callme target = new Callme();
            new Caller(target, "Hello.");
            new Caller(target, "Synchronized");
            new Caller(target, "World");
        }
}
```

Вы можете видеть из приведенного ниже результата работы программы, что `sleep` в методе `call` приводит к переключению контекста между подпроцессами, так что вывод наших 3 строк-сообщений перемешивается:

```
[Hello.
[Synchronized
]
[World
]
]
```

Это происходит потому, что в нашем примере нет ничего, способного помешать разным подпроцессам вызывать одновременно один и тот же метод одного и того же объекта. Для такой ситуации есть даже специальный термин — ***race condition*** (состояние гонки), означающий, что различные подпроцессы пытаются опередить друг друга, чтобы завершить выполнение одного и того же метода. В этом примере для того чтобы это состояние было очевидным и повторяемым, использован вызов `sleep`. В реальных же ситуациях это состояние, как правило, трудноуловимо, поскольку непонятно, где именно происходит переключение контекста, и этот эффект менее заметен и не всегда воспроизводится от запуска к запуску программы. Так что если у вас есть метод (или целая группа методов), который манипулирует внутренним состоянием объекта, используемого в программе с параллельными подпроцессами, во избежание состояния гонки вам следует использовать в его заголовке ключевое слово `synchronized`.

В Java имеется элегантный механизм общения между подпроцессами, основанный на методах `wait`, `notify` и `notifyAll`. Эти методы реализованы, как `final`-методы класса `Object`, так что они имеются в любом Java-классе.

```
            }
        }
}
class PC {
        public static void main(String args[]) {
                Q q = new Q();
                new Producer(q);
                new Consumer(q);
        }
}
```

Хотя методы *put* и *get* класса Q синхронизированы, в нашем примере нет ничего, что бы могло помешать поставщику переписывать данные до того, как их получит потребитель, и наоборот, потребителю ничего не мешает многократно считывать одни и те же данные. Так что вывод программы содержит вовсе не ту последовательность сообщений, которую нам бы хотелось иметь:

```
C:\> java PC
Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
Put: 2
Put: 3
Put: 4
Put: 5
Put: 6
Put: 7
Got: 7
```

Как видите, после того как поставщик помещает в переменную n значение 1, потребитель начинает работать и извлекает это значение 5 раз подряд. Положение можно исправить, если поставщик будет при занесении нового значения устанавливать флаг, например, заносить в логическую переменную значение ***true***, после чего будет в цикле проверять ее значение до тех пор, пока поставщик не обработает данные и не сбросит флаг в ***false***.

Правильным путем для получения того же результата в Java является использование вызовов ***wait*** и ***notify*** для передачи сигналов в обоих направлениях. Внутри метода ***get*** мы ждем (вызов ***wait***) пока ***Producer*** не известит нас (***notify***), что для нас готова очередная порция данных. После того как мы обработаем эти данные в методе ***get***, мы извещаем объект класса ***Producer*** (снова вызов ***notify***) о том, что он может передавать следующую порцию данных. Соответственно, внутри метода ***put*** мы ждем (***wait***), пока ***Consumer*** не обработает данные, затем мы передаем новые данные и извещаем (***notify***) об этом объект-потребитель. Ниже приведен переписанный указанным образом класс Q.

```
class Q {
        int n;
```

```
    boolean valueSet = false;
    synchronized int get() {
        if (!valueSet)
        try wait();
        catch(InterruptedException e);
        System.out.println("Got: " + n);
        valueSet = false;
        notify();
        return n;
    }
    synchronized void put(int n) {
        if (valueSet)
        try wait(); catch(InterruptedException e);
        this.n = n;
        valueSet = true;
        System.out.println("Put: " + n);
        notify();
    }
}
```

А вот и результат работы этой программы, ясно показывающий, что синхронизация достигнута.

```
C:\> java Pcsynch
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
```

Клинч — редкая, но очень трудноуловимая ошибка, при которой между двумя легковесными процессами существует кольцевая зависимость от пары синхронизированных объектов. Например, если один подпроцесс получает управление объектом *X*, а другой — объектом *Y*, после чего *X* пытается вызвать любой синхронизированный метод *Y*, этот вызов, естественно, блокируется. Если при этом и *Y* попытается вызвать синхронизированный метод *X*, то программа с такой структурой подпроцессов окажется заблокированной навсегда. В самом деле, ведь для того чтобы один из подпроцессов захватил нужный ему объект, ему нужно снять свою блокировку, чтобы второй подпроцесс мог завершить работу.

Есть еще множество функций и несколько классов, например, *ThreadGroup* и *SecurityManager*, которые имеют отношение к подпроцессам, но эти области в Java проработаны еще не до конца. Скажем лишь, что при необходимости можно получить информацию об этих интерфейсах из документации по SDK API.

Вопросы и упражнения:

1. Почему необходимо синхронизировать доступ к объектам в многопоточной среде исполнения.
2. Перечислите способы, предусмотренные в API Java, которыми исполняющийся поток может вызвать принудительное переключение контекста.

Задания для самостоятельной разработки

Задание №1. Реализовать контейнер на основе односвязного списка. Предусмотреть операции вставки, удаления элементов и последовательностей элементов контейнера, итератор для обхода элементов контейнера, сортировку, функцию `forEach`, применяемую ко всем элементам последовательности, функцию `clone()`. На основе разработанных классов написать программу, показывающую возможности контейнера.

Задание №2. Реализовать контейнер на основе двусвязного списка. Предусмотреть операции вставки, удаления элементов и последовательностей элементов контейнера, итераторы для обхода элементов контейнера, сортировку, функцию `forEach`, применяемую ко всем элементам последовательности, функцию `clone()`. На основе разработанных классов написать программу, показывающую возможности контейнера.

Задание №3. Реализовать hash-таблицу, не используя классы SDK. На основе разработанных классов написать программу, показывающую возможности контейнера.

Задание №4. Реализовать контейнер, хранящий отсортированные последовательности с использованием двоичного дерева. На основе разработанных классов написать программу, показывающую возможности контейнера.

Задание №5. Реализовать контейнер, хранящий элементы в древовидной структуре, имитирующей файловую систему (структурированное хранилище). На основе разработанных классов написать программу, показывающую возможности контейнера.

Задание №6. Реализовать контейнер, осуществляющий сортировку слиянием. На основе разработанных классов написать программу, показывающую возможности контейнера.

Задание №7. Реализуйте представление бесскобочного алгебраического выражения вида `x+y*z+…+m=f` списковой структурой, в которой элементами являются переменные (`x,y` и т.д.), операции `(*, /, +, -)` и отношение (=). На основе этого представления разработайте алгоритм разрешения исходного выражения относительно любой входящей в него переменной. Результатом такого разрешения должна быть трансформированная списковая структура. На основе разработанных классов написать программу для работы с данной списковой структурой.

Задание №8. Реализовать представление многочлена с произвольными целыми коэффициентами в виде связанного списка. При этом, если a_i=0, то соответствующий элемент должен отсутствовать. Для разработанной структуры реализовать операции: проверка на равенство, вычисление значения, умножение двух многочленов.

Задание №9. Реализовать представление многочлена с произвольными целыми коэффициентами в виде связанного списка. При этом, если a_i=0, то соответствующий элемент должен отсутствовать. Для разработанной структуры реализовать операции: проверка на равенство, вычисление значения, сложение двух многочленов.

Задание №10. Реализуйте контейнер, моделирующий очередь элементов ограниченной длины с дисциплиной "первым пришел — первым вышел" на структуре циклического списка. Контейнер должен поддерживать операции: постановка и вывод из очереди, обход элементов, идентификация ситуаций "очередь полна" и "очередь пуста".

Список используемой литературы

1. Нотон П. JAVA:Справ.руководство: Пер.с англ./Под ред.А.Тихонова.- М.:БИНОМ:Восточ.Кн.Компания,1996.
2. Нотон П., Шилдт Г. Полный справочник по Java.- McGraw-Hill,1997. - Киев: Диалектика,1997.
3. Флэнэген Д. Java in a Nutshell.- O'Reilly & Associates, Inc., 1997, Издательская группа BHV, Киев, 1998.
4. Ренеган Э.Дж.(мл.)1001 адрес WEB для программистов:Новейший путеводитель программиста по ресурсам World Wide Web:Пер.с англ. - Минск:Попурри,1997.
5. Сокольский М.В. Все об Intranet и Internet.-М.:Элиот,1998.
6. Чен М.С. и др. Программирование на JAVA: 1001 совет: Наиболее полное руководство по Java и Visual J++: Пер.с англ. / Чен М.С., Грифис С.В., Изи Э.Ф. - Минск: Попурри, 1997.
7. Эферган М. Java: Справочник. - QUE Corporation, 1997, СПб.: Изд-во Питер, 1998.
8. Вебер Д. Технология Java в подлиннике.- QUE Corporation, 1996, Издательская группа BHV, Спб.,1997.
9. Мейнджер Д. Java: Основы программирования.- McGraw-Hill, Inc., 1996, Издательская группа BHV, Киев, 1997.
10. И.Ю. Баженова. Язык программирования Java.- М.: Диалог-МИФИ, 1997.
11. Родли Д. Создание Java-апплетов.- The Coriolis Group, Inc., 1996, Киев: НИПФ "ДиаСофт Лтд.", 1996.

12. Томас М., Пател П., Хадсон А., Болл Д. (мл.). Секреты программирования для Internet на Java. - Ventana Press, Ventana Communications Group, U.S.A.,1996; Спб.: Изд-во Питер, 1997.
13. А.Волш А. Основы программирования на Java для World Wide Web.- IDG Books Worldwide,Inc.,1996, Киев: Диалектика, 1996.
14. Арнольд К., Гослинг Д. Язык программирования Java.- Addison-Wesley Longman,U.S.A.,1996; Спб.: Изд-во Питер, 1997.
15. Бартлетт Н., Лесли А., Симкин С. Программирование на Java: Путеводитель.- The Coriolis Group,Inc.,1996, Киев: НИПФ "ДиаСофт Лтд.", 1996.
16. Крис Джамса. Библиотека программиста Java.- Jamsa Press, 1996, - Минск: Попурри, 1996.

СОДЕРЖАНИЕ

Учебное издание

Востокин Сергей Владимирович
Солдатова Ольга Петровна

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЕ МЕТОДЫ И СИСТЕМЫ
Язык программирования Java

Курс лекций

Редактор Л. Я. Чегодаева
Корректор Л. Я. Чегодаева

Подписано в печать 10.01.2003 г. Формат 60x84 1/16.
Бумага офсетная. Печать офсетная.
Усл. печ. л. 3,48. Усл. кр.-отт. 3,61. Уч.-изд.л. 3,75.
Тираж 100 экз. Заказ 21 . Арт. С-4(Д2)/2002

Самарский государственный аэрокосмический
университет им. академика С. П. Королева.
443086 Самара, Московское шоссе, 34.

Отпечатано в УПЛ.
443001 Самара, ул. Молодогвардейская, 151.