МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Тульский государственный университет»

Институт прикладной математики и компьютерных наук
Кафедра «Прикладная математика и информатика»

Утверждено на заседании кафедры
«Прикладная математика и информатика»
24 января 2022 г., протокол № 5

Заведующий кафедрой

_____М.В. Грязев

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
по выполнению практических (семинарских) занятий
по дисциплине (модулю)
«Практикум на электронных вычислительных машинах»
Часть 3

основной профессиональной образовательной программы
высшего образования – программы бакалавриата

по направлению подготовки
01.03.02 Прикладная математика и информатика

с направленностью (профилем)
Прикладная математика и информатика

Форма обучения: очная

Идентификационный номер образовательной программы: 010302-01-22

Тула 2022 год

Разработчик методических указаний

Смирнов О.И., доцент каф. ПМиИ, к.ф.-м.н., доцент	
(ФИО, должность, ученая степень, ученое звание)	*(подпись)*

8 семестр
Язык программирования Java

Занятие 1.
Синтаксис и типовые программные конструкции языка Java

История создания и развития Java.

Java широко известна как новейший объектно-ориентированный язык, легкий в изучении и позволяющий создавать программы, которые могут исполняться на любой платформе без каких-либо доработок (кросс-платформенность). Программисты могут добавить к этому описанию, что язык похож на упрощенный C или C++ с добавлением garbage collector'a - автоматического сборщика "мусора" (механизм освобождения памяти, которая больше не используется программой). Также известно, что Java ориентирована на Интернет, и самое ее распространенное применение - небольшие программы, называющиеся апплеты, которые запускаются в браузере и являются частью HTML-страниц.

Изначально язык назывался Oak (дуб), а работа по его созданию началась еще в 1990 году внутри корпорации Sun. Сотрудники компании пришли к выводу, что назрела необходимость создания нового проекта - разработки доступного для пользователей языка, а также небольшой аппаратно-программной платформы, что позволит создавать аппаратно-независимые приложения. Первоначальной целью было - выяснить, что станет следующей волной развития компьютерной индустрии, и что необходимо разработать для успешного участия в ней. В итоге было решено, что будущее за объединением сетей, компьютеров, и других электронных устройств в единую согласованную инфраструктуру.

Сначала были попытки модифицировать C++, чтобы создать язык для написания программ, минимально ориентированных под конкретные платформы. Однако очень скоро стало понятно, что это практически невозможно. И тогда были начаты собственные разработки, но многое из идеологии C++ было взято за основу. Параллельно с разработкой языка шла работа над графической системой. Объектно-ориентированный язык Oak был готов стать достаточно мощным инструментом для написания программ, которые могут работать в сетевом окружении. Его объекты, свободно распространяющиеся по сети, работали бы на любом устройстве, начиная с персонального компьютера и заканчивая обычными бытовыми видеомагнитофонами и тостерами.

Для победного нашествия Oak не хватало последнего штриха - браузера, который бы поддерживал эту технологию. Поэтому в сентябре 1994 года WebRunner (браузер) уже демонстрировался руководству Sun. Небольшие программы, написанные на Oak для распространения через Интернет, назвали апплетами (applets). Его возможности, в частности, динамические элементы (двигающаяся объемная молекула или бегущие строки), устроили революцию в Интернете.

В начале 1995 года, когда стало ясно, что официальное объявление уже близко, за дело взялись маркетологи. По результатам их исследований Oak был переименован в Java, а WebRunner стал называться HotJava.С февраля 1995 г. Java, реализация платформы и HotJava свободно распространяются через Интернет. Интерес нарастал лавинообразно, поэтому в марте 1995 года в газете Sun Jose Mercury News вышла статья с описанием новой технологии, в которой приводился адрес официального сайта http://java.sun.com/, который по сей день является основным источником информации по Java.

23 мая 1995 года технология Java и HotJava были официально объявлены Sun, и тут же было сообщено, что новая версия самого поопулярного браузера Netscape Navigator 2.0 будет поддерживать новую технологию. По сути, это означало, что отныне Java становится такой же неотъемлемой составляющей WWW, как и HTML.

Основная линия развития Java остается связанной с браузерами. С помощью Java вебстраницу можно наполнить не только обычным текстом, но и динамическими элементами. Но на самом деле Java - это больше, чем симпатичное украшение HTML. Поскольку это полно-

ценный язык программирования, то с его помощью можно создавать сложный пользовательский интерфейс.

Про HotJava вскоре пришлось забыть, т.к. компании Sun не удалось создать конкурентоспособный браузер.

4 декабря 1995 года компании Netscape и Sun совместно объявляют новый "язык сценариев" (scripting language) Java Script. Как следует из пресс-релиза это открытый, кросс-платформенный объектный язык сценариев для корпоративных сетей и Интернета. Код Java Script описывается прямо в HTML тексте (хотя возможно и подгружать его из отдельных файлов с расширением .js). Этот язык предназначен для создания приложений, которые связывают объекты и ресурсы на клиентской машине или на сервере. Таким образом, Java Script с одной стороны расширяет и дополняет HTML, а с другой стороны - дополняет Java. С помощью Java пишутся объекты-апплеты, которыми можно управлять через язык сценариев.

В январе 1996 года было сформировано новое подразделение JavaSoft, которое и занялось разработкой новых Java-технологий и продвижением их на рынок. Главная цель - появление все большего количества самых разных приложений, написанных на этой платформе. В частности, 1 июля 1997 года было объявлено, что ученые NASA с помощью Java-апплетов управляют роботом, изучающим поверхность Марса.

Создатели Java разработали не просто еще один язык программирования, а универсальную платформу для исполнения приложений. Каким же образом можно "сгладить" различия и многообразие операционных систем? Способ не новый, но эффективный - виртуальная машина. Приложения на языке Java исполняются в специальной, универсальной среде, которая называется Java Virtual Machine. JVM - это программа, которая пишется специально для каждой реальной платформы, чтобы с одной стороны скрыть все ее особенности, а с другой - предоставить единую среду исполнения для Java-приложений. Фирма Sun и ее партнеры создали JVM практически для всех современных операционных систем. Когда говорится о браузере с поддержкой Java, также подразумевается, что в нем имеется встроенная виртуальная машина.

Особенности работы с Java.

Наличие виртуальной машины определяет многие свойства Java.

Является Java языком компилируемым или интерпретируемым? На самом деле, используются оба подхода.

Исходный код любой программы на языке Java представляется обычными текстовыми файлами, которые могут быть созданы в любом текстовом редакторе или специализированном средстве разработки и имеют расширение .java. Эти файлы подаются на вход Java-компилятора, который транслирует их в специальный Java байт-код. Именно этот компактный и эффективный набор инструкций поддерживается JVM и является неотъемлемой частью платформы Java.

Результат работы компилятора сохраняется в бинарных файлах с расширением .class. Java-приложение, состоящее из таких файлов, подается на вход виртуальной машине, которая начинает их исполнять, или интерпретировать, так как сама является программой.

В Java применяется строгая типизация. Это означает, что любая переменная и любое выражение имеет тип, известный уже на момент компиляции. Такой подход применен для упрощения выявления проблем, ведь компилятор сразу сообщает об ошибках и указывает их расположение в коде. Поиск же исключительных ситуаций (exceptions - так в Java называются некорректные ситуации) во время исполнения программы (runtime) потребует сложного тестирования, причем причина, породившая дефект, может обнаружиться в совсем другом классе. Таким образом, нужно прикладывать дополнительные усилия при написании кода, зато существенно повышается его надежность (а это одна из основополагающих целей, для которых и создавался новый язык).

В Java есть всего 8 типов данных, которые не являются объектами: 5 целочисленных типов: (byte, short, int, long, char), 2 дробных типа (float и double) и булевский тип boolean. Та-

кие типы называются простыми или примитивными (от английского primitive). Все остальные - объектные или ссылочные.

В Java был введен механизм автоматической сборки мусора (от английского garbage collector). Предположим, программа создает некоторый объект, работает с ним, а дальше наступает момент, когда он больше уже не нужен. Необходимо освободить занимаемую память, чтобы не мешать операционной системе нормально функционировать. Т.о., программист вообще не думает об освобождении памяти. Виртуальная машина сама подсчитывает количество ссылок на каждый объект, и если оно становится равным нулю, то такой объект помечается для обработки garbage collector. Программист должен следить лишь за тем, чтобы не оставалось ссылок на ненужные объекты.

Кроме введения garbage collector, были предприняты и другие шаги для облегчения разработки - отказ от множественного наследования, упрощение синтаксиса, возможность создание многопоточных приложений.

Синтаксис языка Java.

В Java комментарии бывают 2 видов:

• строчные

• блочные

Строчные комментарии начинаются с ASCII-символов // и длятся до конца текущей строки. Блочные комментарии располагаются между ASCII-символами /* и */, могут занимать произвольное количество строк.

Кроме этого, существует особый вид блочного комментария - комментарий разработчика. Он применяется для автоматического создания документации кода. В стандартную поставку JDK, начиная с версии 1.0, входит специальная утилита javadoc. Комментарий разработчика записывается так же, как и блочный. Единственное различие в начальной комбинации символов - для документации комментарий необходимо начинать с /**. Символ * в начале каждой строки и предшествующие ему пробелы и знаки табуляции игнорируются.

Ниже перечислены все виды лексем в Java:

• идентификаторы (identifiers);

• ключевые слова (key words);

• литералы (literals);

• разделители (separators);

• операторы (operators).

Идентификаторы - это имена, которые даются различным элементам языка для упрощения доступа к ним. Имена имеют пакеты, классы, интерфейсы, поля, методы, аргументы и локальные переменные (все эти понятия подробно рассматриваются в дальнейших главах). Идентификаторы можно записывать символами Unicode, то есть, на любом удобном языке. Длина имени не ограничена. Идентификатор состоит из букв и цифр. Имя не может начинаться с цифры. Java-буквы, используемые в идентификаторах, включают в себя ASCII-символы A-Z, a-z, а также знаки подчеркивания _ и доллара $. Знак доллара используется только при автоматической генерации кода (чтобы исключить случайное совпадение имен), либо при использовании каких-либо старых библиотек, в которых допускались имена с этим символом.

Ключевые слова:

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

Ключевые слова goto и const зарезервированы, но не используются.

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

• целочисленный (integer);

• дробный (floating-point);
• булевский (boolean);
• символьный (character);
• строковый (string);
• null-литерал (null-literal).

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

Дробные литералы представляют собой числа с плавающей десятичной точкой. Правила записи таких чисел такие же, как и в большинстве современных языков программирования.

Символьный литерал должен содержать строго один символ, например, `'a'`, или специальную последовательность, начинающуюся с \.

Строковые литералы состоят из набора символов и записываются в двойных кавычках. Длина может быть нулевой или сколь угодно большой.

Null литерал может принимать всего одно значение: null. Это литерал ссылочного типа, причем эта ссылка никуда не ссылается, объект отсутствует. Разумеется, его можно применять к ссылкам любого объектного типа данных.

Операторы.

Оператор присваивания = и оператор сравнения ==:

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

Наряду с 4 обычными арифметическими операциями +, -, *, /, есть оператор получения остатка от деления %.

Логические операторы "и" и "или" (& и |) можно использовать в двух вариантах. Первый вариант операторов (&, |) всегда вычисляет оба операнда, второй же - (&&, ||) не будет продолжать вычисления, если значение выражения уже очевидно.

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

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

Оператор if - оператор условного перехода. В общем случае конструкция выглядит так:

if (логическое выражение) выражение или блок 1
else выражение или блок 2

Оператор switch() в случае необходимости множественного выбора. Выбор осуществляется на основе целочисленного значения. Структура оператора:

switch(int value){
case const1:
выражение или блок
case const2:
выражение или блок
case constn:
выражение или блок
default:
выражение или блок
}

Причем фраза default не является обязательной

В языке Java имеется три основных конструкции управления циклами.

• цикл while

• цикл do

• цикл for

Основная форма цикла while может быть представлена так

while(логическое выражение)

повторяющееся выражение или блок;

В данной языковой конструкции повторяющееся выражение или блок, будет исполняться до тех пор, пока логическое выражение будет иметь истинное значение.

Если выражение или блок представляющий тело цикла будет завершен не нормальным образом по причине

• встретился оператор continue, то часть тела цикла следующая за оператором continue будет пропущена и выполнение цикла продолжится с начала. Если continue используется с меткой и метка принадлежит к данному while, то выполнение его будет аналогичным. Если метка не относится к данному while, то его выполнение будет прекращено.

• встретился оператор break, то выполнение цикла будет прекращено

• если выполнение блока будет прекращено по другим причинам (возникла исключительная ситуация), то выполнение while будет прекращено по тем же причинам.

Основная форма цикла do имеет следующий вид

do

повторяющееся выражение или блок;

while(логическое выражение)

В отличии от цикла while цикл do, будет выполняться до тех пор, пока логическое выражение будет ложным. Вторым важным отличием является то, что do будет выполнен как минимум один раз.

Основная форма цикла for выглядит следующим образом:

for(выражение инициализации; условие; выражение обновления)

повторяющееся выражение или блок;

В некоторых случаях требуется изменить ход выполнения программы. В традиционных языках программирования для этих целей используется оператор goto, однако в Java его использование не предусмотрено. Для этих целей применяются операторы break и continue.

Оператор continue может применяться только в циклах while, do, for. Если в потоке вычислений встречается оператор continue, то выполнение текущей последовательности операторов (выражений) должно быть прекращено и управление будет передано на начало блока содержащего этот оператор.

Оператор break изменяет последовательность выполнения, но не возвращает исполнение к началу цикла, а прерывает его.

Оператор return предназначен для возврата управления из вызываемого метода в вызываемый. Если в последовательности операторов выполняется return то управление немедленно (если это не оговорено особо) передает управление в вызывающий метод.

Переменные.

Переменные используются в программе для хранения данных. Любая переменная имеет три базовых характеристики: имя; тип; значение. О примитивных типах упоминалось ранее. Рассмотрим ссылочные типы.

Если каждая переменная имеет значение, то ссылочные же переменные хранят лишь ссылки на объекты, причем различные переменные могут ссылаться на один и тот же объект.

```
Point p1 = new Point(3,5);
Point p2=p1;
p1.x=7;
print(p2.x);
```

В примере выражение new Point(3,5) создает новый объект-точку с координатами (3,5), объект класса:

```
class Point {
int x, y;
}
```

В результате обе переменные p1 и p2 ссылаются на один объект.

Объект (object) - это экземпляр некоторого класса или экземпляр массива. Класс - это описание объектов одинаковой структуры. Объектов этого класса может не быть вовсе, а может быть создано сколь угодно много.

Объекты всегда создаются с использованием ключевого слова new, причем одно слово new порождает строго один объект (или вовсе ни одного, если происходит ошибка). После ключевого слова указывается имя класса, от которого мы собираемся породить объект. Создание объекта всегда происходит через вызов одного из конструкторов класса. Если конструктор отработал успешно, то выражение new возвращает ссылку на созданный объект. Эту ссылку можно сохранить в переменной, передать в качестве аргумента в какой-либо метод или использовать другим способом.

Над ссылочными значениями можно производить следующие операции:

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

Обращение к полям и методам объекта можно назвать основной операцией над ссылочными величинами. Осуществляется она с помощью символа . (точка). Имена бывают простыми (simple), состоящими из одного идентификатора (они определяются во время объявления), и составными (qualified), состоящими из последовательности идентификаторов, разделенных точкой.

```
// Объявляем класс Parent
class Parent {
}
// Объявляем класс Child, и наследуем
// его от класса Parent
class Child extends Parent {
     }
Parent p = new Child();
```

Теперь переменная типа Parent указывает на объект, порожденный от класса Child.

Добавим к уже объявленным классам еще один:

```
// Объявляем новый класс и наследуем
// его от класса Child
class ChildOfChild extends Child {
}
```

Теперь заведем переменную нового типа:

```
Parent p = new ChildOfChild();
print(p instanceof Child);
```

Имена и идентификаторы.

Простое имя состоит из одного идентификатора, а составное - из нескольких. Однако не всякий идентификатор входит в состав имени. Во-первых, в выражении объявления (declaration) идентификатор еще не является именем. Другими словами, идентификатор становится именем после первого появления в коде в месте объявления. Во-вторых, есть возможность обращаться к полям и методам объектного типа не через имя типа или объектной переменной, а через ссылку на объект, полученную в результате выполнения выражения.

```
country.getCity().getStreet();
```

В данном примере getStreet является не именем, а идентификатором, так как соответствующий метод вызывается у объекта, полученного в результате вызова метода getCity(). Причем, country.getCity - как раз является составным именем метода.

Выполняемый класс обязательно содержит метод main(...), который является точкой входа для выполнения программы. Метод main(...) получает набор строк, содержащий аргументы командной строки. Имя исходного файла должно совпадать с названием класса и иметь расширение java.

Пример кода простейшего консольного приложения:

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

Результатом работы будет:

```
Hello, world!
```

Занятие 2.
Средства разработки консольных приложения на Java под Windows.

Консольные приложения выполняют чтение из стандартного ввода и запись в вывод (операции ввода/вывода) без использования графического интерфейса пользователя.

В Java существует понятие главной точки входа консольного приложения и связанных методов для чтения и записи в консоль. В языке Java – это метод main.

Исходный код на языке Java содержится в файлах с расширением java.

Программа для трансляции в байт-код называется javac.exe.

Программа для выполнения байт-кода называется java.exe.

Оттранслированная в байт-код программа имеет расширение class.

Для запуска программы нужно вызвать интерпретатор java.exe, указав в параметрах вызова класс выполнения.

Среда для выполнения Java программ называется JRE (Java Runtime Environment).

Среда JRE содержит реализацию виртуальной Java машины для конкретной операционной системы.

Базовая среда разработки программ на Java называется Java 2 SDK (Java2™ Software Development Kit). Среда разработки содержит набор стандартных библиотек и утилитных программ для разработчика. Также Java 2 SDK включает в себя среду выполнения java программ. Пакет Java 2 SDK свободно доступен в сети интернет по адресу http://java.sun.com/j2se Установка SDK не требует инсталляции дополнительных компонентов.

Головной каталог содержит лицензионное соглашение и краткое описание системы. Так же в нем находится архив исходного кода src.zip.

bin\ - каталог инструментария разработчика. В нем находятся компилятор байт-кода javac.exe и интерпретатор java.exe, а также прочие утилиты

lib\ - набор стандартных библиотек Java для разработчиков

demo\ - каталог с примерами

include\ - каталог с файлами для взаимодействия с программами на C посредством технологии JNI (Java Native Interface)

jre\ - каталог, содержащий реализацию Java Runtime Environment

jre\bin\ - набор запускаемых файлов и DLL для Java-машины (JVM)

jre\lib\ - библиотеки и набор настроечных файлов для JRE

При трансляции программы задается полное имя файла, включая расширение, потому что это имя описывает путь до исходного файла.

Если запустить программу javac.exe без параметров, то она покажет список допустимых параметров с кратким описанием каждого параметра. Подробная информация об аргументах программы javac.exe доступна в стандартной документации - ссылка “Tool Documentation” на главной странице.

Пример команды для транслирования исходного кода в байт-код:

javac.exe HelloWorld.java

Скомпилированный класс является файлом, содержащим байт-код. При запуске программы задается полное имя класса (не файла) для выполнения, поэтому расширение .class не указывается.

Параметр classpath содержит список возможных путей до места хранения байт-кода. Пути разделяются с помощью символа ‘;’ в Windows и ‘:’ в UNIX системах.

Если параметр classpath не задан, интерпретатор использует системную переменную среды CLASSPATH.

Для получения информации о дополнительных параметрах программы java.exe используйте те же способы, что рекомендованы для программы javac.exe

Пример команды запуска:

java.exe -classpath . HelloWorld

Файлы с расширением jar являются специальными архивами, содержащими байт-код.

Для создания JAR архивов используется программа jar.exe, входящая в состав SDK.

JAR совместим по своему формату с архивом ZIP. Отличительной особенностью формата JAR является поддержка метаинформации об архиве.

Пример команды для создания архива:

jar -cf hello.jar HelloWorld.class

Интерпретатор javac.exe использует параметр classpath для поиска байт-кода, указанного для исполнения. Параметр classpath обычно содержит список библиотек в виде jar-архивов.

Пример выполнения консольного приложения, байт-код которого находится в архиве:

java -classpath hello.jar HelloWorld

Существует несколько сред для создания консольных приложений на Java.

Программа Eclipse является интегрированной средой разработки (Integrated Development Environment, сокращенно IDE). Основным разработчиком Eclipse является фирма IBM. Программа свободно доступна по адресу http://www.eclipse.org. Установка Eclipse не требует инсталляции дополнительных компонентов. В выбранном при инсталляции каталоге будет построена структура подкаталогов необходимых для работы среды Eclipse.

Среда IDE NetBeans.

Создание консольного приложения.

Выше уже упоминалось, что главный класс консольного приложения должен называться точно так же, как и файл исходного текста этого класса. В этом классе необходимо определить статический метод с именем main. Данный метод играет роль точки входа приложения, то есть ему будет передано управление сразу после запуска консольного приложения.

В качестве стандартного потока вывода консольное приложение использует поток java.lang.System.out, а в качестве стандартного потока ввода - java.lang.System.in. Есть и стандартный поток для вывода сообщений об ошибках - java.lang.System.err.

Методу main могут быть переданы аргументы - в качестве параметров запуска приложения. Статический метод main не возвращает никакого значения и потому имеет тип void.

Рассмотрим процесс создания консольного приложения на примере.

Определим класс SimpleConsoleApp, разместив его исходный текст в файле с именем SimpleConsoleApp.java. Файл должен называться именно так, в противном случае компилятор выдаст сообщение об ошибке.

В классе SimpleConsoleApp есть поле szAppName класса String, конструктор и один метод с именем main:

```
public class SimpleConsoleApp
{
  String szAppName;

  public SimpleConsoleApp()
  {
    . . .
  }

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

Статический метод main не возвращает никакого значения и потому имеет тип void. В качестве единственного аргумента этому методу передается массив строк параметров запуска приложения.

Конструктор класса SimpleConsoleApp вызывается при создании объекта данного класса. Заметим, что при запуске консольного приложения объект главного класса не создается. При необходимости вы можете создать такой объект в методе main.

В нашем приложении конструктор выполняет очень простую задачу - он инициализирует поле szAppName, записывая в него ссылку на строку названия приложения:

```
public SimpleConsoleApp()
{
  szAppName = new String(
    "Simple Console Application");
}
```

Рассмотрим теперь исходный текст метода main.

Внутри этого метода мы определили несколько переменных, которые будут использованы для ввода с консоли целого числа:

```
int i = 0;
byte bKbd[] = new byte[256];
String szStr = "";
StringTokenizer st;
```

Далее метод main создает объект sc класса SimpleConsoleApp:

```
SimpleConsoleApp sc =
 new SimpleConsoleApp();
```

Это приводит к вызову конструктора, инициализирующего поле szAppName. Следующая строка выводит название приложения на консоль, обращаясь к указанному полю:

```
System.out.println(sc.szAppName);
```

Для вывода в стандартный выходной поток мы применили метод println, добавляющий после выведенной строки символы возврата каретки и перевода строки. В результате приглашение для ввода числа, расположенное в исходном тексте строкой ниже, будет отображено с новой строки:

```
System.out.print("Enter number: ");
```

Здесь мы воспользовались методом print, не вызывающем при выводе перевод строки. Поэтому текстовый курсор остановится сразу за строкой "Enter number: ".

Теперь о вводе строки с клавиатуры.

Мы применили метод read, вызвав его для стандартного потока ввода java.lang.System.in. Заметим, что во время работы этого метода, а также в процессе преобразования полученного массива символов в целое число возможно возникновение исключений. Поэтому мы заключили фрагмент кода, выполняющий эти функции, в блок try-catch:

```
int i = 0;
byte bKbd[] = new byte[256];
String szStr = "";
StringTokenizer st;
try
{
  int iCnt = System.in.read(bKbd);
  szStr = new String(bKbd, 0, iCnt);

  st = new StringTokenizer(szStr, "\r\n");
  szStr =
    new String((String)st.nextElement());

  Integer intVal = new Integer(szStr);
  i = intVal.intValue();
}
catch(Exception ex)
{
  System.out.println(ex.toString());
```

}

Метод read считывает символы с клавиатуры и записывает их в массив байт с именем bKbd. Этот метод возвращает управление, когда пользователь закончил ввод и нажал клавишу <Enter>. При этом в переменную iCnt записывается количество прочитанных символов.

Далее мы должны преобразовать последовательность введенных символов в число (ожидается, что пользователь будет вводить цифры). Для этого мы выполняем промежуточное преобразование массива символов в текстовую строку класса String, удаляем из нее символы возврата каретки и перевода строки, а затем создаем на базе полученной строки объект класса Integer.

Создание строки выполняется так:

szStr = new String(bKbd, 0, iCnt);

Строка String содержит 16-разрядные символы Unicode. Чтобы преобразовать массив байт в строку Unicode, мы задаем значение старшего байта во втором параметре конструктора, равное нулю.

Удаление символов возврата каретки и перевода строки выполняется при помощи класса StringTokenizer, предназначенного для разбора текстовых строк. Создавая объект этого класса, мы передаем конструктору через второй параметр список символов-разделителей:

st = new StringTokenizer(szStr, "\r\n");

szStr = new String((String)st.nextElement());

Метод nextElement возвращает первый элемент строки до разделителя, то есть в нашем случае всю строку до символов возврата каретки и перевода строки.

Далее строка преобразуется в объект класса Integer (целое число):

Integer intVal = new Integer(szStr);

В процессе этого преобразования возможно возникновение исключения java.lang.NumberFormatException (например, если пользователь ввел текстовую строку, которую невозможно преобразовать в численное значение). Блок catch выводит название исключения на консоль.

На следующем этапе мы преобразуем объект Integer к примитивному типу int, вызывая для этого метод intValue:

i = intVal.intValue();

Таким образом мы воспользовались классом Integer для преобразования текстовой строки в значение типа int. Результат этого преобразования затем отображается на консоли:

System.out.println("Number: " + i);

Занятие 3.
Организация массивов и файлового ввода-вывода в программах на Java.

Массивы являются важной составляющей языка программирования. В Java работа с массивами во многом похожа на то, как это делается в других языках, но есть также и важные особенности.

Если массив имеет длину n, то значениями индексов являются числа от 0 до n-1. Массивы могут быть основаны как на примитивных типах (например, для хранения числовых значений 100 измерений), так и на ссылочных (например, если нужно хранить описание 100 автомобилей в гараже в виде экземпляров класса Car). Базовый тип может также быть массивом. Таким образом конструируется массив массивов, или многомерный массив.

В Java массив символов char[] и класс String являются различными типами. Их значения могут быть легко конвертированы друг в друга с помощью специальных методов, но они все же не относятся к идентичным типам.

В Java массивы являются объектами (примитивных типов в Java всего восемь, и их количество не меняется), их тип напрямую наследуется от класса Object, поэтому все элементы этого класса доступны у объектов-массивов.

В качестве примера рассмотрим объявление переменной типа массив, основанный на примитивном типе int

int a[];

Также допустимой записью является:

int[] a;

Количество пар квадратных скобок указывает на размерность массива. Для многомерных массивов допускается смешанная запись:

int[] a[];

Переменная a имеет тип "двумерный массив, основанный на int". Аналогично объявляются массивы с базовым объектным типом:

Point p, p1[], p2[][];

Создание переменной типа массив еще не создает экземпляры этого массива, изначально она имеет значение null. Чтобы создать экземпляр массива, нужно воспользоваться ключевым словом new, после чего указывается тип массива и в квадратных скобках указывается длина массива.

int a[]=new int[5];

Point[] p = new Point[10];

Переменная инициализируется ссылкой, указывающей на только что созданный массив. После его создания можно обращаться к элементам, используя ссылку на массив, далее в квадратных скобках указывается индекс элемента.

```
int array[]=new int[5];
for (int i=0; i<5; i++) {
array[i]=i*i;
}
for (int j=0; j<6; j++) {
System.out.println(j+"*"+j+"="+array[j]);
}
```

Результатом выполнения программы будет:

0*0=0
1*1=1
2*2=4
3*3=9
4*4=16

И далее появится ошибка времени исполнения, так как индекс превысит максимально возможное значение для такого массива.

Поскольку для экземпляра массива длина является постоянной характеристикой, для всех массивов существует специальное поле length, позволяющее узнать ее значение. Например:

```
Point p[]=new Point[5];
for (int i=0; i<p.length; i++) {
p[i]=new Point(i, i);
}
```

Поскольку массив является объектным типом данных, его значения могут быть приведены к типу Object или, что то же самое, быть присвоены переменной типа Object. Например,

Object o = new int[4];

Это дает интересную возможность для массивов, основанных на типе Object, хранить в качестве элемента ссылку на самого себя:

```
Object arr[] = new Object[3];
arr[0]=new Object();
arr[1]=null;
arr[2]=arr; // Элемент ссылается на весь массив!
```

Если создать массив на основе примитивного числового типа, то изначально после создания все элементы массива имеют значение по умолчанию, то есть 0. Если массив объявлен на основе примитивного типа boolean, то и в этом случае все элементы будут иметь значение по умолчанию false.

Рассмотрим создание массива на основе ссылочного типа. Предположим, это будет класс Point. При создании экземпляра массива с применением ключевого слова new не создается ни один объект класса Point, создается лишь один объект массива. Каждый элемент массива будет иметь пустое значение null. В этом можно убедится с помощью простого примера:

```
Point p[]=new Point[5];
for (int i=0; i<p.length; i++) {
System.out.pritnln(p[i]);
}
```

Результатом будут лишь слова null. Далее нужно инициализировать элементы массива по отдельности, например, в цикле.

Кроме того, есть и другой способ создания массивов - инициализаторы. В этом случае ключевое слово new не используется, а ставятся фигурные скобки, и в них перечисляются через запятую значения всех элементов массива. Например, для числового массива явная инициализация записывается следующим образом:

```
int i[]={1, 3, 5};
int j[]={}; // эквивалентно new int[0]
```

Аналогично можно порождать массивы на основе объектных типов, например:

```
Point p=new Point(1,3);
Point arr[]={p, new Point(2,2), null, p};
// или
String sarr[]={"aaa", "bbb", "cde"+"xyz"};
```

Рассмотрим многомерные массивы. Например, в следующем примере:

```
int i[][]=new int[3][5];
```

переменная i ссылается на двумерный массив, который можно представить себе в виде таблицы 3x5. Суммарно в таком массиве содержится 15 элементов, к которым можно обращаться через комбинацию индексов от (0, 0) до (2, 4). Пример заполнения двумерного массива через цикл:

```
int pithagor_table[][]=new int[5][5];
for (int i=0; i<5; i++) {
for (int j=0; j<5; j++) {
pithagor_table[i][j]=i*j;
System.out.print(pithagor_table[i][j]+ "\t");
```

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

Результатом выполнения программы будет:

0 0 0 0 0
0 1 2 3 4
0 2 4 6 8
0 3 6 9 12
0 4 8 12 16

Более точный подход заключается в том, что в Java нет двумерных, и вообще многомерных, массивов, а есть массивы, базовыми типами которых являются также массивы. Например, тип int[] означает "массив чисел", а int[][] означает "массив массивов чисел".

Если создать двумерный массив и определить переменную x, которая на него ссылается, то используя x и два числа в паре квадратных скобок каждое (например, x[0][0]), можно обратиться к любому элементу двумерного массива. Но в то же время используя x и одно число в паре квадратных скобок, можно обратиться к одномерному массиву, который является элементом двумерного массива. Его можно проинициализировать новым массивом с некоторой другой длиной, и таблица перестанет быть прямоугольной - она примет произвольную форму. В частности, можно одному из одномерных массивов присвоить даже значение null.

x[0]=new int[7];
x[1]=new int[0];
x[2]=null;

После таких операций массив, на который ссылается переменная x назвать прямоугольным никак нельзя. Зато хорошо видно, что это просто набор одномерных массивов или значений null. Полезно подсчитать, сколько объектов порождается выражением new int[3][5]. Правильный подсчет таков: создается один массив массивов (1 объект) и 3 массива чисел, каждый длиной 5 (3 объекта). Итого, 4 объекта.

Как обычно, массивы, основанные на простых и ссылочных типах, описываем раздельно. Переменная типа массив примитивных величин может хранить значения только точно такого же типа, либо null. Переменная типа массив ссылочных величин может хранить следующие значения:

• null;
• значения точно того же типа, что и тип переменной;
• все значения типа массив, основанный на типе, приводимом к базовому типу исходного массива.

Занятие 4.
Структура базовых классов системы разработки на Java.

Иерархия классов платформы Java.

Класс Object. В Java отсутствует множественное наследование. Каждый класс может иметь только одного родителя. Таким образом, мы можем проследить цепочку наследования от любого класса, поднимаясь все выше. Существует класс, на котором такая цепочка всегда заканчивается, это класс Object. Именно от него наследуются все классы, в объявлении которых явно не указан другой родительский класс.

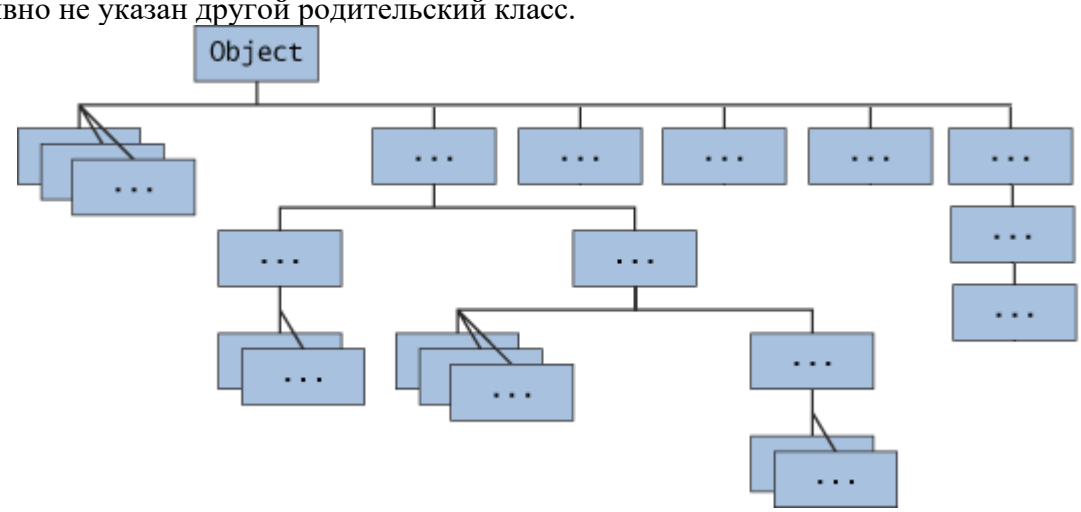

Рис. Иерархия классов

Т.о., любой класс напрямую или через своих родителей является наследником Object. Отсюда следует, что методы этого класса есть у любого объекта (поля в Object отсутствуют), а потому они представляют особенный интерес.

Рассмотрим основные из них:

• getClass()

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

```
String s = "abc";
Class cl=s.getClass();
print(cl.getName());
```

Результатом будет строка:

```
java.lang.String
```

• equals()

Этот метод имеет один аргумент типа Object и возвращает boolean. Служит для сравнения объектов по значению, а не по ссылке. Сравнивается состояние объекта, у которого вызывается этот метод, с передаваемым аргументом.

```
Point p1=new Point(2,3);
Point p2=new Point(2,3);
print(p1.equals(p2));
```

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

• hashCode()

Этот метод возвращает значение int. Цель hashCode() - представить любой объект целым числом. Особенно эффективно это используется в хэш-таблицах (в Java есть стандартная реализация такого хранения данных).

• toString()

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

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

• finalize()

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

Класс Class. Является метаклассом для всех классов Java. Когда JVM загружает файл .class, который описывает некоторый тип, в памяти создается объекта класса Class, который будет хранить это описание. Например, если в программе есть строка:

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

то это означает, что в системе созданы следующие объекты:

1. собственно, объект типа Point, описывающий точку (1,2)
2. объект класса Class, описывающий класс Point
3. объект класса Class, описывающий класс Object. Так как класс Point наследуется от Object, описание этого класса также необходимо.
4. объект класса Class, описывающий класс Class. Это обычный Java-класс, который должен быть загружен по общим правилам.

Наследование.

Класс, который наследуется от другого класса, называется подклассом (потомком). Класс, который наследуют называется суперклассом (родительским классом). Все классы за исключением класса Object имеют один и только один непосредственный суперкласс.

Рассмотрим пример наследования.

Вот возможная реализация класса Bicycle

```
public class Bicycle {

    // the Bicycle class has three fields
    public int cadence;
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public  Bicycle(int  startCadence,  int  startSpeed,  int
startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    // the Bicycle class has four methods
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
```

```
        }

        public void applyBrake(int decrement) {
            speed -= decrement;
        }

        public void speedUp(int increment) {
            speed += increment;
        }

    }
```

Класс MountainBike, являющийся потомком класса Bicycle может выглядеть следующим образом

```
    public class MountainBike extends Bicycle {

        // the MountainBike subclass adds one field
        public int seatHeight;

        // the MountainBike subclass has one constructor
        public MountainBike(int startHeight, int startCadence, int
startSpeed, int startGear) {
            super(startCadence, startSpeed, startGear);
            seatHeight = startHeight;
        }

        // the MountainBike subclass adds one method
        public void setHeight(int newValue) {
            seatHeight = newValue;
        }
    }
```

MountainBike наследует все поля и методы класса Bicycle и добавляет поле seatHeight и методы для работы с ним.

Подкласс не наследует члены суперкласса, отмеченные модификатором private. Однако, если суперкласс содержит методы для доступа к таким полям, они могут быть вызваны тем самым обеспечивая доступ к private полям.

Переопределение и скрытие методов.

Методы экземпляра класса. Если в подклассе определен метод, который имеет одинаковую сигнатуру с методом в базовом классе, то метод подкласса переопределяет метод базового класса.

Методы класса. Метод в подклассе, определен статический метод, который имеет одинаковую сигнатуру со статическим методом в базовом классе, то метод подкласса скрывает метод базового класса.

Различие между скрытием и переопределением демонстрирует следующий пример

```
public class Animal {
    public static void testClassMethod() {
        System.out.println("The class method in Animal.");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Animal.");
    }
}
```

Следующий класс, является потомком класса Animal:

```
public class Cat extends Animal {
```

```
        public static void testClassMethod() {
            System.out.println("The class method in Cat.");
        }
        public void testInstanceMethod() {
            System.out.println("The instance method in Cat.");
        }

        public static void main(String[] args) {
            Cat myCat = new Cat();
            Animal myAnimal = myCat;
            Animal.testClassMethod();
            myAnimal.testInstanceMethod();
        }
    }
```

Результат выполнения программы будет следующим

```
The class method in Animal.
The instance method in Cat.
```

Доступ к полям и методам базового класса.

Если метод подкласса переопределят метод базового класса, получить доступ к исходному методу можно с помощью ключевого слова super. Также super можно использовать для доступа к скрытым полям. Например:

```
public class Superclass {

    public void printMethod() {
        System.out.println("Printed in Superclass.");
    }
}
```

И класс, который расширяет базовый:

```
public class Subclass extends Superclass {

    public void printMethod() { //overrides printMethod in
Superclass
        super.printMethod();
        System.out.println("Printed in Subclass");
    }
    public static void main(String[] args) {

    Subclass s = new Subclass();
    s.printMethod();
    }
}
```

Результат выполнения примера:

```
Printed in Superclass.
Printed in Subclass
```

Конструктор подкласса.

Следующий пример демонстрирует использование ключевого слова super для вызова конструктора базового класса. Рассмотрим конструктор класса MountainBike, который расширяет класс Bicycle. Конструктор вызывает конструктор базового класса и выполняет свои действия по инициализации.

```
    public MountainBike(int startHeight, int startCadence, int
startSpeed, int startGear) {
        super(startCadence, startSpeed, startGear);
```

```
        seatHeight = startHeight;
    }
```

Вызов конструктора базового класса должен быть первой строкой конструктора подкласса. Если явный вызов конструктора не описан, компилятор автоматически добавляет вызов конструктора базового класса без параметров.

final методы и классы.

Ключевое слово final в описании метода показывает, что метод не может быть переопределен подклассом. Например, целесообразно использовать final для методов, вызываемых конструктором, для того, чтобы избежать их изменений, которые могут повлиять на корректность работы.

Также с модификатором final может быть описан весь класс, тогда класс расширять нельзя. Например, String.

abstract методы и классы.

Ключевое слово abstract в определении класса показывает, что класс является абстрактным, то есть нельзя создать экземпляр этого класса. Абстрактные методы – это методы объявленные без реализации, также с модификатором abstract. Например:

```
abstract void moveTo(double deltaX, double deltaY);
```

Если класс содержит абстрактные методы он сам должен быть объявлен абстрактным:

```
public abstract class GraphicObject {
   // declare fields
   // declare non-abstract methods
   abstract void draw();
}
```

Занятие 5.
Языковые средства обработки исключений при контроле конфликтных ситуаций времени выполнения.

При выполнении программы зачастую могут возникать ошибки. В одних случаях это вызвано ошибками программиста, в других внешними причинами. Например, может возникнутьошибка ввода/вывода при работе с файлом или сетевым соединением. В классических языках программирования, например в C, требовалось проверять некое условие которое указывало на наличие ошибки и, в зависимости от этого предпринимать определенные действия.

Например

```
…
int statusCode = someAction();
if (statusCode){
… обработка ошибки
}else{
if(statusCode){
… обработка ошибки …
}}
…
```

В Java появилось более простое и элегантное решение - обработка исключительных ситуаций.

```
try{
someAction();
anotherAction()
}catch(Exception e){
… обработка исключительной ситуации
}
```

Причины возникновения ошибок

Существует три причины возникновения исключительных ситуаций.

• Попытка выполнить некорректное выражение.

Например, деление на ноль, или обращение к объекту по ссылке, равной null, попытка использовать класс, описание которого (class-файл) отсутствует, и т.д. В таких случаях всегда можно точно указать, в каком месте произошла ошибка - именно в некорректном выражении.

• Выполнение оператора throw.

Очевидно, что и здесь можно легко указать место возникновения исключительной ситуации.

• Асинхронные ошибки во время исполнения программы.

Причиной таких ошибок могут быть сбои внутри самой виртуальной машины (ведь она также является программой), или вызов метода stop() у потока выполнения thread). В этом случае невозможно указать точное место программы, где происходит исключительная ситуация. Если мы пытаемся остановить поток выполнения (вызвав метод stop()), то мы не можем предсказать, при выполнении какого именно выражения этот поток остановится.

Таким образом, все ошибки в Java делятся на синхронные и асинхронные. Первые сравнительно проще, так как принципиально возможно найти точное место в коде, которое является причиной возникновения исключительно ситуации. Конечно, Java является строгим языком в том смысле, что все выражения до точки сбоя обязательно будут выполнены, в то же время ни одно последующее выражение никогда выполнено не будет.

Асинхронные ошибки гораздо сложнее в обнаружении и исправлении. Обычному разработчику очень затруднительно выявить причины сбоев в виртуальной машине. Это могут быть ошибка создателей JVM, несовместимость с операционной системой, аппаратный сбой и многое другое. Все же современные виртуальные машины реализованы довольно хорошо, и

подобные сбои происходят крайне редко (при условии использования качественных комплектующих).

Аналогичная ситуация наблюдается и в случае с принудительной остановкой потоков исполнения. Поскольку это действие выполняется операционной системой, никогда нельзя предсказать, в каком именно месте остановится поток. Это означает, что программа может многократно отработать корректно, а потом неожиданно дать сбой просто из-за того, что поток остановился в каком-то другом месте. По этой причине принудительная остановка крайне не рекомендуется.

При возникновении исключительной ситуации управление передается от кода, вызвавшего исключительную ситуацию, на ближайший блок catch (или вверх по стеку), и создается объект, унаследованный от класса Throwable или его потомков (см. диаграмму иерархии классов исключений), который содержит информацию об исключительной ситуации и используется при ее обработке. Собственно в блоке catch указывается именно класс обрабатываемой ситуации. Подробно обработка ошибок рассматривается ниже.

Иерархия по которой передается информация об исключительной ситуации зависит от того, где эта исключительная ситуация возникла. Если это

• метод, то управление будет передаваться в то место, где этот метод был вызван;

• конструктор, то управление будет передаваться туда, где попытались создать объект (как правило, применяя оператор new);

• если это статический инициализатор, то управление будет передано туда, где произошло первое обращение классу, потребовавшее его инициализацию.

Допускается создание собственных классов исключительных ситуаций. Осуществляется это с помощью механизма наследования, т.е. класс пользовательской исключительной ситуации, должен быть унаследован от класс Throwable или его потомков.

Обработка исключительных ситуаций

Конструкция try-catch. В общем случае конструкция выглядит так.

```
try{
…
}catch(SomeExceptionClass e){
…
}catch(AnotherExceptionClass e){
…
}
```

Работает она следующим образом. Сначала выполняется код заключенный в фигурные скобки оператора try. Если во время его выполнения не происходит никаких нештатных ситуаций, то далее управление передается, за закрывающую фигурную скобку, последнего оператор catch ассоциированного с данным оператором try.

В случае если в пределах try возникает исключительная ситуация, то далее выполнение кода производится по одному из ниже перечисленных сценариев.

• возникла исключительная ситуация, класс которой указан в качестве параметра одного из блоков catch. В этом случае производится выполнение блока кода ассоциированного с этим catch (заключенного в фигурные скобки). Далее если код в этом блоке завершается нормально, то и весть оператор try завершается нормально и управление передается на оператор (выражение) следующий за закрывающей фигурной скобкой последнего catch ассоциированного с данным try. Если код в catch завершается нештатно, то и весь try завершается нештатно по той же причине.

• если возникла исключительная ситуация, которая класс которой не указан в качестве аргумента, ни в одном catch, то выполнение всего try завершается нештатно.

Если в последовательности операторов могут возникнуть как ошибки ввода/вывода так и ошибки арифметических вычислений, вовсе нет нужды помещать различные фрагменты кода в разные операторы try{}catch(){}. Достаточно обеспечить несколько catch() для различных типов исключений.

Конструкция try-catch-finally. Оператор finally предназначен для того, что бы обеспечить гарантированное выполнение какого-либо фрагмента кода.

Последовательность выполнения такой конструкции будет следующей: Если оператор try выполнен нормально, то будет выполнен блок finally. В свою очередь, если оператор finally выполняется нормально, то весь оператор try выполняется нормально. Если происходит преждевременное окончание выполнения блока finally, то весть оператор try завершается предварительно по тем же причинам.

• существует оператор catch, который перехватывает данный тип исключения, происходит выполнение связанного с catch блока.

- Если блок catch выполняется нормально, то выполняется блок finally

• в свою очередь если блок finally завершается нормально, то весь try завершается нормально.

• Если finally завершается предварительно, то и весь оператор try завершается предварительно по той же причине.

- Если блок catch завершается ненормально, то выполняется блок finally

• в свою очередь, если блок finally завершается нормально, то оператор try завершается не нормально, по той же причине, по которой не нормально завершился блок catch

• если блок finally завершается ненормально, то весь блок try завершается ненормально, по той же причине, что и блок finally

• в списке операторов catch не находится такого, который обработал бы возникшее исключение. Все равно выполняется блок finally. В этом случае, если

- finally завершится нормально, весь try завершится не нормально по той же причине по которой было нарушено исполнение try.

- finally завершится ненормально, то try завершится ненормально по той же причине, по которой ненормально завершился finally

Если оператор try завершился нормально, то выполнится блок finally. И если

• блок finally завершится нормально, то весь try завершится нормально

• блок finally завершится не нормально, то весь try завершится не нормально, по той же причине.

Следует обратить внимание, что при использовании конструкции finally, блок кода ассоциированный с ним будет выполнятся всегда. Если во время обработки исключительной ситуации, возникнет новая исключительная ситуация, то исключительная ситуация, которая послужила первопричиной будет потеряна.

Рассмотрим пример применения конструкции try-catch-finally.

```
try{
byte [] buffer = new byte[128];
FileInputStream fis = new FileInputStream("file.txt");
while(fis.read(buffer) > 0){
… обработка данных
}
}catch(IOException es){
… обработка исключения …
}finally{
fis.flush();
fis.close();
}
```

Следует обратить внимание, что использование flush() не является обязательным в данном контексте, т.к. буфер ввода/вывода будет очищен при вызове close() указания, и здесь использован для большей наглядности.

Если в данном примере поместить операторы очистки буфера и закрытия файла сразу после окончания обработки данных, то при возникновении ошибки ввода вывода, корректного закрытия файла не произойдет. Следует отметить, что блок finally, будет выполнен в лю-

бом случае, вне зависимости от того произошла обработка исключения или нет, возникло это исключение или нет.

В конструкции try-catch-finally обязательным является использование одной из частей оператора catch или finally. То есть, конструкция

```
try{
...
}finally{
...
}
```

является вполне допустимой. В этом случае блок finally при возникновении исключительной ситуации будет выполнен, хотя сама исключительная ситуация обработана не будет и будет передана для обработки на более высокий уровень иерархии.

Следует рассмотреть два специальных случая использования finally.

```
try{
…
}catch(Exception e){
…
System.exit(0);
}finally{
…
}
```

В этом случае блок finally выполнен НЕ БУДЕТ т.к. выполнение данного потока будет прекращено.

Использование оператора throw. Помимо того, что предопределенная исключительная ситуация могла быть возбуждена исполняющей системой Java, программист сам может сгенерировать это условие. Делается это с помощью оператора throw. Например

```
….
public int calculate(int theValue){
if( theValue < 0){
throw new Exception("Параметр для вычисления не должен быть отрицательным");
}}
…
```

В данном случае, предполагается, в качестве параметра методу может быть передано только положительное значение, если это условие не выполнено, то с помощью оператора throw возбуждается исключительная ситуация. В действительности данный код не будет откомпилирован, т.к. компилятор выдаст сообщение об ошибке. Если в методе возбуждается исключительная ситуация, то должно быть выполнено одно из двух правил

• исключительная ситуация должна быть обработана в теле метода (т.е. код должен возбуждающий исключительную ситуацию, должен быть помещен в блок try {} catch(UserException ue){})

• метод должен делегировать обработку исключительной ситуации вызвавшему его коду. Для этого в сигнатуре метода применятся ключевое слово throws, после которого должны быть перечислены через запятую все исключительные ситуации, которые может вызывать данный метод. Т.е. приведенный выше пример должен быть приведен к следующему виду

```
….
public int calculate(int theValue) throws Exception{
if( theValue < 0){
throw new Exception("Some descriptive info");
}}
…
```

Т.о. возбуждение исключительной ситуации в программе производится с помощью оператора throw, слева от которого указывается объект, который может быть приведен к типу Throwable. (Как правило этот объект создается в этом же месте с помощью оператора new, хотя это условие и не является обязательным).

В некоторых случаях после обработки исключительной ситуации, возможно, возникнет необходимость передать информацию о ней в вызывающий код. В этом случае throw используется вторично. Например

```
…
try{
…
}catch(IOException ex){
…
// Обработка исключительной ситуации
…
// Повторное возбуждение исключительной ситуации
throw ex;
}
```

Встроенные исключения Java.

Непроверяемые исключения.

Подклассы класса RuntimeException – классы непроверяемых исключений.

Пакет java.lang определяет следующие стандартные непроверяемые исключения времени выполнения, которые, подобно всем другим классам пакета java.lang, неявно импортированы, и поэтому на них можно ссылаться с помощью их простых имен:

ArithmeticException: возникла исключительная арифметическая ситуация, такая как операция целочисленного деления с нулевым знаменателем.

ArrayStoreException: была сделана попытка сохранить в массиве компонент тип значения которого не совместим с типом компонент массива.

ClassCastException: была сделана попытка привести ссылку на объект несоответствующего типа.

IllegalArgumentException: метод передал недействительный или несоответствующий аргумент или вызывает несоответствующий объект. Подклассами этого класса являются:

IllegalThreadStateException: поток не был в соответствующем режиме для требуемого действия.

NumberFormatException: была сделана попытка конвертировать строку в числовое выражение, но строка не имела соответствующего формата.

IllegalMonitorStateException: поток попытался работать с другими потоками или давать сведения другим нитям, имея дело с объектами, которые она не блокировала.

IndexOutOfBoundsException: индекс (массива, строки или вектора) или поддиапазон,заданный с помощью двух значений индекса или индексом и длиной, были вне диапазона.

NegativeArraySizeException: была сделана попытка, создать массив с отрицательной длиной.

NullPointerException: была сделана попытка использовать пустую ссылку в случае, где требовалась ссылка на объект.

SecurityException: было обнаружено нарушение безопасности.

Проверяемые исключения.

Все стандартные подклассы класса Exception, за исключением класса RuntimeException, являются классами проверяемых исключений.

Пакет java.lang определяет следующие стандартные исключения, которые, как все другие классы пакета java.lang, неявно импортированы, и поэтому на них можно ссылаться с помощью их простых имен:

ClassNotFoundException: класс или интерфейс с указанным именем не был найден.

CloneNotSupportedException: для копирования объекта, был вызван метод clone класса Object, но класс этого объекта не реализует интерфейс Cloneable.

IllegalAccessException: была сделана попытка загрузить класс с помощью строки, дающей его полное квалифицированное имя, но в настоящее время метод не имеет доступа к определению указанного класса, потому что класс не является классом с типом доступа public и находится в другом пакете.

InstantiationException: была сделана попытка создать экземпляр класса, используя метод newInstance класса Class, но создание экземпляра названного объекта класса невозможно, потому что это - интерфейс, класс типа abstract , или массив.

InterruptedException: текущий поток ожидал, а другой поток прервал текущий поток, используя метод interrupt класса Thread.

Занятие 6.
Средства разработки графических приложений на Java с использованием пакета Java.awt.

Апплеты.

Апплеты (applets) - это маленькие приложения, которые размещаются на серверах Internet, транспортируются клиенту по сети, автоматически устанавливаются и запускаются на месте, как часть документа HTML. Когда апплет прибывает к клиенту, его доступ к ресурсам ограничен.

Ниже приведен исходный код канонической программы HelloWorld, оформленной в виде апплета:

```
import java.awt.*;
import java.applet.*;
public class HelloWorldApplet extends Applet {
public void paint(Graphics g) {
g.drawString("Hello World!", 20, 20);
}
}
```

Этот апплет начинается двумя строками, которые импортируют все пакеты иерархий java.applet и java.awt. Дальше в нашем примере присутствует метод paint, замещающий одноименный метод класса Applet. При вызове этого метода ему передается аргумент, содержащий ссылку на объект класса Graphics. Последний используется для прорисовки нашего апплета. С помощью метода drawString, вызываемого с этим объектом типа Graphics, в позиции экрана (20,20) выводится строка "Hello World".

Для того чтобы с помощью браузера запустить этот апплет, нам придется написать несколько строк html-текста.

<applet code="HelloWorldApplet" width=200 height=40>

</applet>

Вы можете поместить эти строки в отдельный html-файл, либо вставить их в текст этой программы в виде комментария и запустить программу appletviewer с его исходным текстом в качестве аргумента. На экране появится строка приветствия.

Перерисовка

В апплете HelloWorldApplet мы заместили метод paint, что позволило апплету выполнить отрисовку. В классе Applet предусмотрены дополнительные методы рисования, позволяющие эффективно закрашивать части экрана.

Метод repaint используется для принудительного перерисовывания апплета. Этот метод, в свою очередь, вызывает метод update.

Если вы хотите добиться ритмичной смены кадров изображения, воспользуйтесь методом *repaint(time)* - это позволит уменьшить количество кадров, нарисованных не вовремя.Здесь устанавливается крайний срок для перерисовки (период задается в миллисекундах относительно времени вызова repaint).

repaint(x, y, w, h)

Эта версия ограничивает обновление экрана заданным прямоугольником, изменены будут только те части экрана, которые в нем находятся.

repaint(time, x, y, w, h)

Этот метод - комбинация двух предыдущих.

Задание размеров графических изображений

Графические изображения вычерчиваются в стандартной для компьютерной графики системе координат, в которой координаты могут принимать только целые значения, а оси направлены слева направо и сверху вниз. У апплетов и изображений есть метод size, который

возвращает объект Dimension. Получив объект Dimension, вы можете получить и значения его переменных width и height:

```
Dimension d = size();
System.out.println(d. width + "," + d.height);
```

Цвет.

Цветовая система AWT разрабатывалась так, чтобы была возможность работы со всеми цветами. После того, как цвет задан, Java отыскивает в диапазоне цветов дисплея тот, который ему больше всего соответствует. Вы можете запрашивать цвета в той семантике, к которой привыкли - как смесь красного, зеленого и голубого, либо как комбинацию оттенка, насыщенности и яркости. Вы можете использовать статические переменные класса Color.black для задания какого-либо из общеупотребительных цветов - black, white, red, green, blue, cyan, yellow, magenta, orange, pink, gray, darkGray и lightGray.

Для создания нового цвета используется один из трех описанных ниже конструкторов.

• Color(int, int, int)

Параметрами для этого конструктора являются три целых числа в диапазоне от 0 до 255 для красного, зеленого и голубого компонентов цвета.

• Color(int)

У этого конструктора - один целочисленный аргумент, в котором в упакованном виде заданы красный, зеленый и голубой компоненты цвета. Красный занимает биты 16-23, зеленый - 8-15, голубой - 0-7.

Color(float, float, float)

Последний из конструкторов цвета, Color(float, float, float), принимает в качестве параметров три значения типа float (в диапазоне от 0.0 до 1.0) для красного, зеленого и голубого базовых цветов.

Метод класса Color

HSBtoRGB(float, float, float)

преобразует цвет, заданный оттенком, насыщенностью и яркостью (HSB), в целое число в формате RGB, готовое для использования в качестве параметра конструктора Color(int).

Метод RGBtoHSB(int, int, int, float[])

преобразует цвет, заданный тремя базовыми компонентами, в массив типа float со значениями HSB, соответствующими данному цвету.

Цветовая модель HSB (Hue-Saturation-Brightness, оттенок-насыщенность-яркость) является альтернативой модели Red-Green-Blue для задания цветов. В этой модели оттенки можно представить как круг с различными цветами (оттенок может принимать значения от 0.0 до 1.0, цвета на этом круге идут в том же порядке, что и в радуге - красный, оранжевый, желтый, зеленый, голубой, синий, фиолетовый). Насыщенность (значение в диапазоне от 0.0 до 1.0) - это шкала глубины цвета, от легкой пастели до сочных цветов. Яркость - это также число в диапазоне от 0.0 до 1.0, причем меньшие значения соответствуют более темным цветам, а большие - более ярким.

• getRed(), getGreen(), getBlue()

Каждый из этих методов возвращает в младших восьми битах результата значение соответствующего базового компонента цвета.

• getRGB()

Этот метод возвращает целое число, в котором упакованы значения базовых компонентов цвета, причем

```
red = Oxff & (getRGB() >> 16);
green = Oxff & (getRGB() >> 8);
blue = Oxff & getRGB();
```

Класс Graphics

У объектов класса Graphics есть несколько простых функций рисования. Каждую из фигур можно нарисовать заполненной, либо прорисовать только ее границы. Каждый из мето-

дов drawRect, drawOval, fillRect и fillOval вызывается с четырьмя параметрами: int x, int y, int width и int height. Координаты x и y задают положение верхнего левого угла фигуры, параметры width и height определяют ее границы.

• drawLine

drawline(int x1, int y1, int x2, int y2)

Эти линии представляют собой простые прямые толщиной в 1 пиксель. Поддержка разных перьев и разных толщин линий не предусмотрена.

• drawArc и fillArc

Сигнатура методов drawArc и fillArc следующая:

drawArc(int x, int y, int width, int height, int startAngle, int sweepAngle)

Эти методы вычерчивают (fillArc заполняет) дугу, ограниченную прямоугольником (x,y,width, height), начинающуюся с угла startAngle и имеющую угловой размер sweepAngle. Ноль градусов соответствует положению часовой стрелки на 3 часа, угол отсчитывается против часовой стрелки (например, 90 градусов соответствуют 12 часам, 180 - 9 часам, и так далее).

• drawPolygon и fillPolygon

Прототипы для этих методов:

drawPolygon(int[], int[], int)

fillPolygon(int[], int[], int)

Метод drawPolygon рисует контур многоугольника (ломаную линию), задаваемого двумя массивами, содержащими x и y координаты вершин, третий параметр метода - число пар координат. Метод drawPolygon не замыкает автоматически вычерчиваемый контур. Для того, чтобы прямоугольник получился замкнутым, координаты первой и последней точек должны совпадать.

• setPaintMode() и setXORMode(Color)

Режим отрисовки paint - используемый по умолчанию метод заполнения графических изображений, при котором цвет пикселей изменяется на заданный. XOR устанавливает режим рисования, когда результирующий цвет получается выполнением операции XOR (исключающее или) для текущего и указанного цветов (особенно полезно для анимации).

Шрифты.

Библиотека AWT обеспечивает большую гибкость при работе со шрифтами благодаря предоставлению соответствующих абстракций и возможности динамического выбора шрифтов. Вот очень короткая программа, которая печатает на консоли Java имена всех имеющихся в системе шрифтов.

```
/*
* <applet code="WhatFontsAreHere" width=100 height=40>
* </applet>
*/
import java.applet.*;
import java.awt.*;
public class WhatFontsAreHere extends Applet {
public void init() {
String fontList[];
// Устаревший способ получения набора шрифтов:
// Toolkit.getDefaultToolkit().getFontList()
fontList = GraphicsEnvironment.getLocalGraphicsEnvironment().
getAvailableFontFamilyNames();
for (int i=0; i < fontList.length; i++) {
System.out.println(i + ": " + fontList[i]);
}
}
}
```

В предыдущих примерах использовался метод drawString(String, x, y). Этот метод выводит строку с использованием текущих шрифта и цвета. Точка с координатами (x,y) соответствует левой границе базовой линии символов, а не левому верхнему углу, как это принято в других методах рисования.

В Java используются различные шрифты, а класс FontMetrics позволяет программисту точно задавать положение выводимого в апплете текста. Ниже приведены некоторые методы класса FontMetrics:

stringWidth

Этот метод возвращает длину заданной строки для данного шрифта.

bytesWidth, charsWidth

Эти методы возвращают ширину указанного массива байтов для текущего шрифта.

getAscent, getDescent, getHeight

Эти методы возвращают подъем, снижение и ширину шрифта. Сумма подъема и снижения дают полную высоту шрифта. Высота шрифта - это не просто расстояние от самой нижней точки букв g и y до самой верхней точки заглавной буквы T и символов вроде скобок. Высота включает подчеркивания и т.п.

getMaxAscent и getMaxDescent

Эти методы служат для получения максимальных подъема и снижения всех символов в шрифте.

Базовые классы

Изучим базовую архитектуру AWT, касающуюся интерфейсных объектов.

Component - это абстрактный класс, который инкапсулирует все атрибуты визуального интерфейса - обработка ввода с клавиатуры, управление фокусом, взаимодействие с мышью, уведомление о входе/выходе из окна, изменения размеров и положения окон, прорисовка своего собственного графического представления, сохранение текущего текстового шрифта, цветов фона и переднего плана (более 100 методов). Перейдем к некоторым конкретным подклассам класса Component.

• Container - это абстрактный подкласс класса Component, определяющий дополнительные методы, которые дают возможность помещать в него другие компоненты, что дает возможность построения иерархической системы визуальных объектов. Container отвечает за расположение содержащихся в нем компонентов с помощью интерфейса LayoutManager.

• Класс Panel - это очень простая специализация класса Container. В отличие от последнего, он не является абстрактным классом. Поэтому о Panel можно думать, как о допускающем рекурсивную вложенность экранном компоненте. С помощью метода add в объекты Panel можно добавлять другие компоненты. После того, как в него добавлены какие-либо компоненты, можно вручную задавать их положение и изменять размер с помощью методов setLocation, setSize и setBounds класса Component.

Основные компоненты AWT, которые можно вставлять в пустую Panel при создании графических приложений:

Canvas. Основная идея использования объектов Canvas в том, что они являются семантически свободными компонентами. Вы можете придать объекту Canvas любое поведение и любой желаемый внешний вид. Его имя подразумевает, что этот класс является пустым холстом, на котором вы можете "нарисовать" любой компонент - такой, каким вы его себе представляете.

Label. Функциональность класса Label сводится к тому, что он знает, как нарисовать объект String - текстовую строку, выровняв ее нужным образом. Шрифт и цвет, которыми отрисовывается строка метки, являются частью базового определения класса Component. Для работы с этими атрибутами предусмотрены пары методов getFont/setFont и getForeground/setForeground. Задать или изменить текст строки после создания объекта с помощью метода setText. Для задания режимов выравнивания в классе Label определены три константы - LEFT, RIGHT и CENTER.

Button. Объекты-кнопки помечаются строками, причем эти строки нельзя выравнивать подобно строкам объектов Label (они всегда центрируются внутри кнопки).

Класс Checkbox часто используется для выбора одной из двух возможностей. При создании объекта Checkbox ему передается текст метки и логическое значение, чтобы задать исходное состояние окошка с отметкой. Программно можно получать и устанавливать состояние окошка с отметкой с помощью методов getState и setState.

CheckboxGroup. Второй параметр конструктора Checkbox используется для группирования нескольких объектов Checkbox. Для этого сначала создается объект CheckboxGroup, затем он передается в качестве параметра любому количеству конструкторов Checkbox, при этом предоставляемые этой группой варианты выбора становятся взаимоисключающими (только один может быть задействован). Предусмотрены и методы, которые позволяют получить и установить группу, к которой принадлежит конкретный объект Checkbox - getCheckboxGroup и setCheckboxGroup. Вы можете пользоваться методами getCurrent и setCurrent для получения и установки состояни выбранного в данный момент объекта Checkbox.

Об остальных компонентах AWT: [10,13].

Занятие 7.
Принципы и средства организации многопоточных приложений на Java.

Виртуальные машины Java поддерживают многопоточность, т.е. поддержку нескольких потоков исполнения (threads) одновременно.

При одновременном обращении нескольких потоков к одним и тем же данным может возникнуть ситуация, когда результат программы будет зависеть от случайных факторов, таких как временное чередование исполнения операций несколькими потоками. В такой ситуации становятся необходимым механизмы синхронизации, обеспечивающие последовательный, или монопольный, доступ. В Java этой цели служит ключевое слово synchronized.

Базовые классы для работы с потоками.

Класс Thread.

Поток выполнения в Java представляется экземпляром класса Thread. Для того чтобы написать свой поток исполнения необходимо наследоваться от этого класса и переопределить метод run(). Например,

```
public class MyThread extends Thread {
public void run() {
// некоторое долгое действие, вычисление
long sum=0;
for (int i=0; i<1000; i++) {
sum+=i;
}
System.out.println(sum);
}
}
```

Метод run() содержит действия, которые должны исполнятся в новом потоке исполнения. Чтобы запустить его, необходимо создать экземпляр класса-наследника, и вызвать унаследованный метод start(), который сообщает виртуальной машине, что необходимо запустить новый поток исполнения и начать в нем исполнять метод run().

```
MyThread t = new MyThread();
t.start();
```

Когда метод run() завершен (в частности, встретилось выражение return) поток выполнения останавливается. Однако ничто не препятствует записи бесконечного цикла в этом методе. В результате поток не прервет своего исполнения, и будет остановлен только при завершении работы всего приложения.

Интерфейс Runnable

Описанный подход обладает одним недостатком. Поскольку в Java отсутствует множественное наследование, требование наследоваться от Thread может привести к конфликту. Если еще раз посмотреть на приведенный выше пример, то станет понято, что наследование производилось только с целью переопределения метода run(). Поэтому предлагается более простой способ создать свой поток исполнения. Достаточно реализовать интерфейс Runnable, в котором объявлен только один метод - уже знакомый void run(). Запишем пример, приведенный выше, с помощью этого интерфейса:

```
public class MyRunnable implements Runnable {
public void run() {
// некоторое долгое действие, вычисление
long sum=0;
for (int i=0; i<1000; i++) {
sum+=i;
}
```

```
System.out.println(sum);
}
}
```

Также незначительно меняется процедура запуска потока:

```
Runnable r = new MyRunnable();
Thread t = new Thread(r);
t.start();
```

Если раньше объект, представляющий сам поток выполнения, и объект с методом run(), содержащим полезную функциональность, были объединены в одном экземпляре класса MyThread, то теперь они разделены. Какой из двух подходов удобней, можно свободно решать в каждом конкретном случае. Подчеркнем, что Runnable не является полной заменой классу Thread, поскольку создание и запуск самого потока исполнения возможно только через метод Thread.start().

Работа с приоритетами

В Java потокам можно назначать приоритеты. Для этого в классе Thread существуют методы getPriority() и setPriority(), а также объявлены три константы:

MIN_PRIORITY
MAX_PRIORITY
NORM_PRIORITY

Из названия очевидно, что их значения описывают минимальное, максимально и нормальное (по умолчанию) значения приоритета.

Рассмотрим следующий пример:

```
public class ThreadTest implements Runnable {
public void run() {
double calc;
for (int i=0; i<50000; i++) {
calc=Math.sin(i*i);
if (i%10000==0) {
System.out.println(getName()+" counts " + i/10000);
}
}
}
public String getName() {
return Thread.currentThread().getName();
}
public static void main(String s[]) {
// Подготовка потоков
Thread t[] = new Thread[3];
for (int i=0; i<t.length; i++) {
t[i]=new Thread(new ThreadTest(), "Thread "+i);
}
// Запуск потоков
for (int i=0; i<t.length; i++) {
t[i].start();
System.out.println(t[i].getName()+" started");
}
}
}
```

В примере используется несколько новых методов класса Thread:

• getName()

Обратите внимание, что конструктору класса Thread передается два параметра. К реализации Runnable добавляется строка. Это имя потока, которое используется только для упрощения его идентификации. Имена нескольких потоков могут совпадать. Если его не за-

дать, то Java генерирует простую строку вида "Thread-" и номер потока (вычисляется простым счетчиком) Именно это имя возвращается методом getName(). Его можно сменить с помощью метода setName().

• currentThread()

Этот статический метод позволяет в любом месте кода получить ссылку на объект класса Thread, представляющий текущий поток исполнения.

Если запустить программу и проанализировать ее работу, то можно видеть, что все три потока были запущены один за другим, и начали проводить вычисления, причем потоки исполняются без определенного порядка, случайным образом. Тем не менее, в среднем они движутся с одной скоростью, никто не отстает и не догоняет.

Введем в программу работу с приоритетами, расставим разные значения для разных потоков и посмотрим, как это скажется на выполнении. Изменяется только метод main()

```
public static void main(String s[]) {
// Подготовка потоков
Thread t[] = new Thread[3];
for (int i=0; i<t.length; i++) {
t[i]=new Thread(new ThreadTest(), "Thread "+i);
t[i].setPriority(Thread.MIN_PRIORITY +
(Thread.MAX_PRIORITY-Thread.MIN_PRIORITY)/t.length*i);
}
// Запуск потоков
for (int i=0; i<t.length; i++) {
t[i].start();
System.out.println(t[i].getName()+" started");
}
}
```

Формула вычисления приоритетов позволяет равномерно распределить все допустимые значения для всех запускаемые потоки. На самом деле, константа минимального приоритета имеет значение 1, максимального - 10, нормального - 5. Так что в простых программах можно явно пользоваться этими величинами и указывать в качестве, например, пониженного приоритета значение 3.

Если теперь проанализировать работу программы, то видно, что потоки, как и раньше, стартуют последовательно. Но затем сразу видно, что чем выше приоритет, тем быстрее отрабатывает поток. Тем не менее, поток с минимальным приоритетом (Thread 0) также получил возможность выполнить одно действие раньше, чем отработал поток с более высоким приоритетом (Thread 1). Это говорит о том, что приоритеты не делают систему однопоточной, выполняющей единовременно лишь один поток с наивысшим приоритетом. Напротив, приоритеты позволяют одновременно работать над несколькими задачами с учетом их важности.

Синхронизация

При многопоточной архитектуре приложения возможны ситуации, когда несколько потоков будут одновременно работать с одними и теми же данными, используя их значения и присваивая новые. В таком случае результат работы программы становится невозможным предопределить, глядя только на исходный код. Финальные значения переменных будут зависеть от случайных факторов, исходя из того, какой поток какое действие успел сделать первым или последним.

Хранение переменных в памяти

Виртуальная машина поддерживает основное хранилище данных (main storage), в котором сохраняются значения всех переменных и которое используется всеми потоками. Под переменными здесь понимаются поля объектов и классов, а также элементы массивов. Что касается локальных переменных и параметров методов, их значения не могут быть доступы

другим потокам, поэтому они не представляют интереса. Для каждого потока создается его собственная рабочая память (working memory), в которую копируются значения всех переменных перед использованием.

Потоки никогда не взаимодействуют друг с другом на прямую, только через главное хранилище.

Модификатор volatile

При объявлении полей объектов и классов может быть указан модификатор volatile. Он устанавливает более строгие правила работы со значениями переменных.

Блокировки

В основном хранилище для каждого объекта поддерживается блокировка (lock), над которой можно произвести два действия - установить (lock) и снять (unlock). Только один поток в один момент времени может установить блокировку на некоторый объект. Если до того, как этот поток выполнит операцию unlock, другой поток попытается установить блокировку, его выполнение будет приостановлено до тех пор, пока первый поток не отпустит ее.

В Java-программе для того, чтобы воспользоваться механизмом блокировок, существует ключевое слово synchronized. Оно может быть применено в двух вариантах - для объявления synchronized-блока и как модификатор метода. В обоих случаях действие его примерно одинаковое. Synchronized-блок записывается следующим образом:

```
synchronized (ref) {
...
}
```

Рассмотрим пример:

```
public class ThreadTest implements Runnable {
private static ThreadTest shared = new ThreadTest();
public void process() {
for (int i=0; i<3; i++) {
System.out.println (Thread.currentThread().
getName()+" "+i);
Thread.yield();
}
}
public void run() {
shared.process();
}
public static void main(String s[]) {
for (int i=0; i<3; i++) {
new Thread(new ThreadTest(),
"Thread-"+i).start();
}
}
}
```

В этом простом примере три потока вызывают метод у одного объекта, чтобы тот распечатал три значения. Результатом будет:

```
Thread-0 0
Thread-1 0
Thread-2 0
Thread-0 1
Thread-2 1
Thread-0 2
Thread-1 1
Thread-2 2
```

Thread-1 2

То есть, все потоки одновременно работают с одним методом одного объекта. Заключим обращение к методу в synchronized-блок:

```
public void run() {
synchronized (shared) {
shared.process();
}
}
```

Synchronized-методы работают аналогичным образом. Также допустимы static synchronized методы. При их вызове блокировка устанавливается на объект класса Class, отвечающего за тип, у которого вызывается этот метод.

При работе с блокировками всегда надо помнить о возможности появления deadlock - взаимных блокировок, которые приводят к зависанию программы. Если один поток заблокировал один ресурс, и пытается заблокировать второй, а другой поток заблокировал второй и пытается заблокировать первый, то такие потоки уже никогда не выйдут из состояния ожидания.

Методы wait(), notify(), notifyAll() класса Object

Рассмотрим три метода класса Object, которые завершают описание механизмов поддержки многопоточности в Java.

Каждый объект в Java имеет не только блокировку для synchronized блоков и методов, но и так называемый wait-set, набор потоков исполнения. Любой поток может вызвать метод wait() любого объекта и таким образом попасть в его wait-set. При этом выполнение такого потока приостанавливается до тех пор, пока другой поток не вызовет у точно этого же объекта метод notifyAll(), который пробуждает все потоки из wait-set. Метод notify() пробуждает один, случайно выбранный поток из этого набора.

Однако применение этих методов связано с одним важным ограничением. Любой из них может быть вызван потоком у объекта только после установления блокировки на этот объект. То есть, либо внутри synchronized-блока с ссылкой на этот объект в качестве аргумента, либо обращение к методам должны быть в синхронизированных методах класса самого объекта.

Рассмотрим более сложный пример для трех потоков:

```
public class ThreadTest implements Runnable {
final static private Object shared=new Object();
private int type;
public ThreadTest(int i) {
type=i;
}
public void run() {
if (type==1 || type==2) {
synchronized (shared) {
try {
shared.wait();
} catch (InterruptedException e) {}
System.out.println("Thread "+type+
" after wait()");
}
} else {
synchronized (shared) {
shared.notifyAll();
System.out.println("Thread "+type+
" after notifyAll()");
}
}
```

```
}
public static void main(String s[]) {
ThreadTest w1 = new ThreadTest(1);
new Thread(w1).start();
try {
Thread.sleep(100);
} catch (InterruptedException e) {}
ThreadTest w2 = new ThreadTest(2);
new Thread(w2).start();
try {
Thread.sleep(100);
} catch (InterruptedException e) {}
ThreadTest w3 = new ThreadTest(3);
new Thread(w3).start();
}
}
```

Результатом работы программы будет:

Thread 3 after notifyAll()
Thread 1 after wait()
Thread 2 after wait()

Рассмотрим, что происходило. Во-первых, был запущен поток 1, который тут же вызвал метод wait() и приостановил свое выполнение. Затем то же самое произошло с потоком 2. Далее начинает выполняться поток 3.

Сразу обращает на себя внимание следующий факт. Еще поток 1 вошел в synchronized-блок, а стало быть установил блокировку на объект shared. Но судя по результатам видно, что это не помешало и потоку 2 затем зайти в synchronized-блок, а потом и потоку 3. Причем для последнего это просто необходимо, иначе как можно "разбудить" потоки 1 и 2?

Можно сделать вывод, что потоки, прежде чем приостановить выполнение после вызова метода wait(), отпускают все занятые блокировки. Итак, вызывается метод notifyAll(). Как уже было сказано, все потоки из wait-set возобновляют свою работу. Однако чтобы корректно продолжить исполнение необходимо вернуть блокировку на объект, ведь следующая команда также находится внутри synchronized-блока!

Получается, что даже после вызова notifyAll() все потоки не могут сразу возобновить работу. Лишь один из них сможет вернуть себе блокировку и продолжить работу. Когда он покинет свой synchronized-блок и отпустит объект, второй поток возобновит свою работу и так далее. Если по какой-то причине объект так и не будет освобожден, поток так никогда и не выйдет из метода wait(), даже если будет вызван метод notifyAll(). В рассмотренном примере потоки один за другим смогли возобновить свою работу.

Кроме этого определен метод wait() с параметром, который задает период тайм-аута, по истечении которого поток сам попытается возобновить свою работу. Но начать ему придется все равно с повторного получения блокировки.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК.

Основная литература

1. Аладьев, В.З. MAPLE 6:Решение математических, статистических и инженерно-физических задач / В.З. Аладьев, М.А. Богдявичюс.– М.: Лаборатория Базовых Знаний, 2001.– 824с.
2. Дьяконов, В.П. MathCAD 11/12/13 в математике: справочник / В.П.Дьяконов.– М.: Горячая линия-Телеком, 2007.– 958с.
3. Московский, А.В. Издательская система LATEX 2ε : учеб. пособие для вузов / А.В. Московский, Ю.В. Московская.– Тула : Изд-во ТулГУ, 2008 .– 172 с.
4. Юров, В.И. Assembler : учебное пособие для вузов / В.И.Юров.– 2-е изд. – М.[и др.] : Питер, 2006 .– 637с.
5. Шилдт, Schildt G. Искусство программирования на Java / Г.Шилдт, Д.Холмс;пер.с англ.и ред. Г.В. Галисеева .– М.и др. : Вильямс, 2005 .– 331с.

Дополнительная литература

1. Дьяконов, В.П. MAPLE 9.5/10 в математике, физике и образовании / В.П.Дьяконов .– М.: СОЛОН-Пресс, 2006.– 720с.
2. Бидасюк, Ю.М. Mathsoft Mathcad 12: самоучитель / Ю.М. Бидасюк .- М.; СПб.; Киев: Диалектика, 2006 .– 224с.
3. Гуссенс, М. Путеводитель по пакету LATEX и его Web-приложениям : Справочник / М.Гуссенс,С.Ратц;Пер.с англ.:Ю.В.Тюменцева,А.В.Чернышева под ред.Б.В.Тоботраса .— М. : Мир, 2001 .— 604с..
4. Зубков, С.В. Assembler для DOS, Windows и Unix / С.В.Зубков .– 3-е изд.,стер. – М. : ДМК, 2006 .– 608с.
5. Вязовик, Н.А. Программирование на Java : Курс лекций для вузов / Н.А.Вязовик .– М., 2003 .– 592с.

Периодические издания

1. Журнал «PC Magazine, Персональный компьютер сегодня».– М.: ЗАО "СК Пресс"

Программное обеспечение и Интернет-ресурсы

1. Львовский, С.М. Работа в системе LaTeX Дистанционный учебный курс. Интернет-университет информационных технологий – INTUIT.ru, 2009.
2. Вязовик, Н.А. Программирование на Java. Дистанционный учебный курс. Интернет-университет информационных технологий – INTUIT.ru, 2009.