Программирование (на языке java).

Лектор: Костин Алексей Николаевич.

Java – универсальный объектно-ориентированный язык программирования высокого уровня, со строгой типизацией данных и С-подобным синтаксисом. Программы, созданные на Java, транслируются в байт-код, который затем выполняется виртуальной машиной.

История развития языков программирования.

Изначально программирование происходило в машинных кодах (0 и 1). Данные и команды для их обработки записывались в двоичном коде и помещались в память компьютера. Управление передавалось первой из команд и далее последовательно выполнялась вся программа.
Затем машинные коды заменили **мнемоническими сокращениями**, т.е командами на английском языке из понятных человеку слов. Так появился язык ASSEMBLER. Команды этого языка по-прежнему управляли напрямую микропроцессором, заставляя выполнить его одно из допустимых действий, или периферийными устройствами. ASSEMBLER считается ЯП низкого уровня.
Программа на ассемблере состояла из команд для *конкретного* микропроцессора и могла выполняться только на нём. Программу без изменений (иногда очень существенных) нельзя было запустить на другом микропроцессоре (например, на более современных), т.е программа была *непереносимой*, и успех – только при совместимости процессоров и идентичном наборе команд.
Программу перед запуском из понятных человеку слов всё равно нужно было переводить в машинные коды.
В 1956 году появился язык FORTRAN. Он также обладал мнемоническими командами, но они были ориентированы на предметную область решаемых задач (например,
команда могла сохранять что-то в файл или вычислять какую-нибудь математическую функцию). Происходит переход от работы с железом к работе с практическими задачами. Обучаться программированию стало проще. Исходный текст программ стал более понятным для неопытных программистов. Фортран был первым ЯП *высокого уровня*, то есть не привязанному к конкретному процессору.
Программы на Фортране перед запуском также надо было транслировать в машинные коды. Этим занималась специальная программа, называемая **компилятором**. Компиляторы могли создаваться для разных платформ (процессоров и ОС0, поэтому написанную на Фортране программу можно было без изменений (или с небольшими изменениями) компилировать для каждой платформы, т.е программы стали *переносимыми на уровне исходного кода*.
В большинстве случаев программы на Фортране занимали больше памяти, работали медленнее, не могли использовать все особенности процессоров и периферии, в отличие от программ на Ассемблере. После любого изменения в исходном коде перед запуском программу нужно было откомпилировать, что для больших программ требовало много времени.
В конце 60-х – начале 70-х появился ЯП С. Он был ЯП высокого уровня, но при необходимости позволял обращаться к аппаратному обеспечению с помощью собственных

команд и включения кода на Ассемблере. С завоевал огромную популярность и позволил создать целые новые ОС. На базе С был создан ЯП С++, он обратно совместим с С (т.е программы, написанные на С, являются корректными с точки зрения С++), но при этом включает новые возможности и позволяет создавать объектно-ориентированные программы.
С++ до сих пор самый популярный язык системного программирования (низкоуровневого). Но для создания прикладных программ (высокого уровня) он уже не так популярен в силу сложной переносимости (тоже только на уровне исходного кода) и необходимости компилировать программу после всякого изменения и под всякую платформу.
В 1995 году появилась первая версия ЯП java, которая была лишена некоторых названных недостатков.

Подходы к программированию.

Первые программы представляли собой линейный алгоритм, где все действия выполнялись от начала до конца в строгой последовательности.

Из-за того, что в сложных программах было много однотипных и повторяющихся действий, их стали выделять в отдельные подпрограммы, которые назвали **процедурами**. Процедуру можно было многократно вызывать в разных местах программы, при этом передавая ей разные параметры. Таким образом, создание сложной программы сводилось к описанию нужных процедур и их вызову в нужном порядке и с подходящими параметрами. Такой подход называется **процедурным программированием**.

Объектно-ориентированное программирование (ООП) – подход к созданию программ, основанный на использовании классов и объектов, взаимодействующих между собой.

Класс описывает устройство и поведение объектов. Устройство описывается через набор характеристик (**свойств**), а поведение – через набор доступных для объектов операций (**методов**).

Классы можно создавать на основе уже имеющихся, добавляя или переопределяя свойства и методы.

Классы можно представлять себе как шаблоны, по которым строятся объекты.

Объекты – это элементы программы, обладающие схожим набором характеристик и поведением (т.е это элементы, построенные на основе одного класса).

Каждый объект имеет некоторое состояние, оно определяется значением всех его свойств.

В одной программе могут существовать несколько классов, и объекты разных классов могут взаимодействовать между собой (через методы).

Создание программы в рамках ООП происходит так:

1. Из предметной области задачи выделяются *существенные* характеристики, с учётом которых создаются классы.
2. От классов порождаются объекты, обладающие некоторым начальным состоянием (значениями свойств).
3. Объекты начинают взаимодействовать между собой с помощью методов, изменяя своё состояние.
4. Так, мы получаем модель некоторого явления или процесса. Чтобы получить

полезный результат, надо оценить состояние этой модели в нужный момент.

Платформа Java.

ЯП является лишь частью платформы, в состав которой также входят:

1) Набор классов, собранных в библиотеку для решения многих стандартных задач. (библиотека класса Java),
2) Виртуальная машина Java. Которая исполняет байт-код, полученный в результате компиляции программ.

Существует несколько редакций платформы:

1) Java SE (Standard Edition) для настольных приложений.
2) Java ME (Micro Edition) для приложений под мобильные и встраиваемые устройства.
3) Java EE (Enterprise Edition) для клиент-серверных приложений корпоративного назначения.

Процесс создания простейшей программы на Java.
В общем случае новую программу можно начать писать в любом текстовом редакторе, но удобнее это делать в редакторе с подсветкой синтаксиса языка или в интегрированной среде разработки (IDE), которая, кроме редактора кода, содержит различные заготовки и полезные функции для ускоренного создания программы.

После того, как исходный код программы написан, она сохраняется с именем *имя_программы.java*, где имя_программы – имя одного из классов, представленных в исходном коде.

Далее программу необходимо откомпилировать. Это можно сделать командой

javac имя_програмы.java

После запуска этой команды происходят 2 важных процесса:

1) Специальный отладчик ищет возможные ошибки в программе. Например, он выявляет синтаксические ошибки, но почти не анализирует сам алгоритм.
2) Если отладчик не выявил ошибок, то запускается компилятор, он превращает исходный код программы в так называемый байт-код – это двоичные данные, непонятные человеку, но готовые к выполнению на JVM.

Если корректно прошли отладка и компиляция, то появляется файл *имя_программы.class*

Теперь мы можем запустить полученный байт-код на JVM с помощью команды

java имя_программы

JVM – это программа, которая запускается внутри ОС и создана специально под конкретную архитектуру (например, для Win и 32-разрядного процессора или для Linux и 64-разрядного процессора или для мобильного телефона).

В JVM происходят запуск и выполнение байт-кода. Указанная выше команда заставит JVM искать в текущей директории файл с именем *имя_программы.class*, в этом файле – класс имя_программы, а в этом классе – метод *Main*. Дальше будут выполняться все инструкции, указанные в методе *Main*. Если класс был назван неправильно или метод *Main* в нём

отсутствует, то программа выполняться не сможет.

Если какие-то изменения были внесены в исходный код, то перед запуском программы её код необходимо сохранить и перекомпилировать, иначе выполняться будет старая версия байт-кода.

Переменные и типы данных.

Переменной называется некоторая область памяти, которой присвоена уникальное и понятное для программиста имя и в которых могут быть записаны данные определённого рода и объёма. Данные из переменной могут быть также прочитаны, и старые данные могут заменяться новыми.

В ЯП Java до того, как переменная будет использована (до записи или чтения данных) необходимо определить её тип – один из возможных вариантов, описывающих, какие данные могут помещаться в переменную, какова область допустимых значений и какие опреации смогут применяться к этим данным.

Тип каждой переменной Java задаётся однажды и не может быть изменён по ходу программы. Такой набор характеристик и называется **статической строгой типизацией**. В других ЯП тип переменной может определяться теми значениями, которые в неё записываются, и даже изменяться по ходу программы (PHP).

Для того. Чтобы объявить переменную, надо указать существующий тип данных и после – имя данной переменной. Имя может состоять из латинских букв, цифр и некоторых символов, но начинаться должно с буквы. По традиции имена переменных и методов начинают с маленькой буквы, а имена классов – с большой. В одной части программы не может существовать двух и более переменных с одинаковыми именами (в одном блоке). Имена Primer, primer, PRIMER различны. Имена переменных рекомендуется придумывать так, чтобы из имени следовало назначение переменной.

Sum – сумма, например, а не произведение.

Логический тип.

Тип может хранить 2 значения в своих переменных: истину (*true)* и ложь (*false*). Для объявления переменной логического типа требуется

boolean a;

После этого можно присваивать и читать значения из переменных.

a=true ;

boolean b;

b=a;

Для данного типа применима следующая операция:

a && b – операция пересечения (логическая И), даёт истинный результат <=> оба аргумента истинны.

a || b — логическое ИЛИ, даёт истинный результат тогда, когда хотя бы один из операндов

истинен.

!a – отрицание, возвращает результат, противоположный значению переменной a.

a^b – исключающее ИЛИ, возвращает истинный результат <=> один из аргументов истинен, а другой ложен.

С участием описанных операций, а также операция присваивания, которая обозначается операцией =, можно составлять и вычислять сложные операции.

boolean a, b, c ;

a = true;

b = !a;

c = a || b && !a;

Приоритет операций таков:

!, &&, ||. но приоритет можно регулировать с помощью (). Выражение в скобках выполняется в первую очередь. () можно вкладывать друг в друга, первым будет выполняться выражение во внутренних ().

Операция присваивания (=) работает однозначно по следующему принципу:

Сначала вычисляется выражение, стоящее справа от =, потом оно присваивается переменной, стоящей слева от =.

Числовые целые типы данных.

Тип	Объём	Значения по умолчанию	Диапазон
byte	1 байт	0	[-128, 127]
char	2 байт	\u 0000	\u 0000... \u FFFF
int	2 байт	0	[-2^15;2^15-1]
long	4 байта	0L	[-2^31, 2^31-1]

Тип *char* хранит не просто целые числа, а номера символов по кодовой таблице Unicode, которая содержит специальные символы, символы псевдографики и буквы различных национальных алфавитов, в т.ч и кириллицей, а также популярные иероглифы.

Условные операторы

Для организации ветвления в программах на Java моет также использоваться оператор *switch*. Он попал в Java по наследству из более старых языков программирования, и любая конструкция с его использованием может быть заменена с помощью набора связок *else if*. Более того, оператор *switch* позволяет работать только с одной переменной, проверяя её точное совпадение с некоторым набором констант и потому является менее универсальным, чем *if*.

switch (n) {

case 1:

```
        //n — единица
    break;
    case 2:
    case 8:
        //n – не единица
    break;
    default:
        //n – ни 1, ни 2, ни 8
}
```

Операторы организации цикла.

Циклом в ЯП Java называется некоторый фрагмент программного кода, который повторяется многократно. При этом цикл может состоять даже из одного выражения или вообще быть пустым.

Выделяют 2 типа циклов: Цикл типа «пока» и цикл типа «n раз».

Цикл типа «n раз».

Данный вид циклов используется в том случае, когда заранее известно, какое количество повторений потребуется. Обычно цикл типа «n раз» связан с каким-то счётчиком, который и отмеряет количество повторений.

Общая схема цикла:

```
for (инициализация; условие повторения; итерация) {
    //Тело цикла
}
```

Инициализация -- создание некоторого счётчика и указание его начального значения. Для одного цикла при необходимости может быть создано сразу несколько счётчиков. Инициализация выполняется один раз до самого первого шага цикла.

Итерация – некоторое выражение, описывающее, каким образом будет изменяться счётчик/счётчики цикла после каждого его шага.

Условия повторения – некоторое логическое выражение, переменная или константа, значение которой будет проверяться перед началом каждого шага цикла (в том числе и самого первого). Если условие истинно, то очередной шаг цикла будет выполняться, иначе цикл остановится.

Тело цикла – набор каких-то операций ЯП Java, который и будет повторяться на каждом шаге цикла. Все значения переменных, полученных на текущем шаге, будут переданы в следующий шаг.

Пример 1:

```
for (int i=1; i <=10; i++) {
```

```
        System.out.println («!»);
}
```

Пример 2:

```
for (int j=99; j > 0; j =j-2) {
        System.out.println (j + « »);
}
//99 97 95 …
```

Пример 3:

```
for (int a = 5; b = 5; a-b>=0; a--, b++) {
         System.out.println (a*b);
}
// -25
-16
-9
-4
-1
0
```

Пример 4 (вычисление факториала)

```
int n = 6, f = 1;
for (int i = 2; i <= n, i++) {
        f = f*i;
}
System.out.println (f);
```

Пример 5 (все делители некоторого целого положительного числа)

```
int n = 24;
for (int i = 1; i<=n; i++) {
        if (n%i == 0) {
        System.out.println (i+» «);
        }
// 1 2 3 4 6 8 12 24
```

Цикл типа «пока».

Данный вид циклов используется в том случае, когда набор некоторых действий нужно повторять до тех пор, пока выполняется определённое условие, при этом заранее можно не знать, сколько раз выполнится цикл.

Общая схема:

```
while (условие) {
        // тело цикла
}
```

Условие – некоторое логическая переменная, выражение или константа, истинность которой проверяется перед каждым шагом цикла, включая первый)

Если условие истинно, то выполняется очередной шаг цикла, иначе происходит выход из цикла и выполняется та часто программы, которая расположена после него.

Тело цикла – набор операций, повторяемых на каждом шаге цикла.

Таким образом, цикл типа «пока» может не выполняться ни разу. Как правило, условие повторения цикла составляется таким образом, чтобы после очередного шага цикла оно всё-таки стало ложным. Если этого не сделать, то получится цикл, повторяющийся бесконечное число раз.

Существует разновидность цикла типа «пока», в которой условие выполнения следующего шага цикла проверяется не перед ним, а после него.

Схема:

```
do {
        // тело цикла
} while (условие);
```

Цикл *do while*, в отличие от цикла *while*, выполняется по крайней мере один раз.

Примеры:

Пример 1.

```
int s = 1;
while (s<11) {
                System.out.print(«!»);
                s++;
}
```

Любой цикл *for* можно свести к циклу *while*, выполняющему аналогичные действия. Более того, для цикла *for* может не указываться первый и последний параметр. Тогда он будет полностью аналогичен циклу *while*.

Пример 2.

```
int s = 1;
```

```
do {
        System.out.print («!»);
        s++;
} while (s<10);
// !!!!!!!!!!
```

Пример 3.

```
int s = 0;
do {
        s++;
        System.out.print («!»);
} while (s<11);
```

Пример 4.

```
Scanner inp = new Scanner(System.in);
double u = inp.nextDouble();
while (u<=0 || u != Math.Floor(u)) {
        u = input.nextDouble();
}
```

Данный фрагмент кода будет читать значение, введённое пользователем с клавиатуры, до тех пор, пока не будет введено целое положительное число.

Пример 5.

```
Scanner inp = new Scanner(System.in);
double u;
do {
        u = inp.nextDouble();
} while (!(u>0 && Math.floor(u) == u));
```

Пример 6.

```
int kol=0;
int n = 1875;
while (n>0) {
        n = n/10 ;
        kol++ ;
}
System.out.print(kol);
// 4
```

Данная программа считает количество цифр в положительном целом числе.

Случайные числа.

Math.random() [0,1)

А надо: [-10, 5]

[-10; 5] ← [-10,6) ← [0;16) ← [0;1)

*Math.floor(Math.random()*16-10)*

или же

a = Math.random();

*a = a*16*

a=a-10;

a = Math.floor(a);

Массивы.

Массив – конечная последовательность элементов *одного типа*.

1. Int [ ] a; – описание массива: описываем, какого типа будет массив, даём описание *double arr [ ]*

   тип [ ] имя;

   тип имя [ ];

2. Выделение памяти под этот массив.

   a = new int [5];

   arr = new double [10];

В случае, когда мы не знаем наверняка длину массива, то мы можем описать переменную, присвоить ей какое-либо значение к моменту выделения памяти для массива.

int n = 15;

a = new int [n];

arr = {3.14; 5; 2.16};

Можно и совместить описание массива и выделение памяти под его элементы в одной строке.

Тип [ ] имя= *new* тип [размер];

тип [ ] имя = { элемент 1, элемент 2, … , элемент 15 }

float [ ] d = newfloat[5];

byte [ ] mas = { 1, 3, 8 };

a.length – получить размер массива.

Циклы в массивах.

Для работы с массивами больше всего подходит цикл типа *for*.

```
for (int ; i=0; i < a.length-1 ; i++)
{ a[i]=10;
}
```

Заполнение массива случайными числами от 0 до 9.

```
int arr[ ];
arr = new int [10];
for (int i=0; i<arr.length; i++)
{
arr[i] = Math.int (Math.random () *10);
System.out.print (arr[i] + «  »
```

Сортировка массива.

Сортировка массива – преобразование массива таким образом, что все элементы располагаются по убыванию или по возрастанию.

mas

[9] [2] [1] [4] [6]

0 1 2 3 4

min

```
for (int i=0; i<mas.length; i++) {
int min = mas [i];
int indmin = i
// пусть минимальным является первый элемент из рассматриваемой части массива.
for (int j=i+1; j<mas.length; j++) {
        if (mas [j]<min) {
                min = mas [j];
                indmin = j; }
}
if (i!=indmin) {
        int temp=mas[i];
```

```
                mas[i] = mas[indmin]
mas[indmin]=temp;
                }
}
```

Метод «Пузырька»

2 9 1 4 3 2

2 9 1 4 3 2

2 1 9 4 3 2

2 1 4 9 3 2

2 1 4 3 9 2

2 1 4 3 2 9

```
for (int i = a.length -1; i>=2; i  - -)
boolean sorted = true;
FOR (int j=0; j<i; j++ {
        if (a[j] > a[j+1]) {
                int temp = a[j];
                a[j]=a[j+1];
                a[j+1]=temp;
                sorted = false;
                        }
                }
if (sorted) {break;}
```

Статические методы.

Из математики известно понятие функции. Похожим элементом в программировании являются методы.

Методом называется фрагмент программы, которому присвоено некоторое уникальное имя и который по этому имени можно обращаться из остальных частей программы. В момент, когда происходит такое обращение, выполняются действия, перечисленные внутри метода.

В объектно-ориентированном программировании основная задача методов заключается в том, чтобы изменять текущее состояние объекта, но до тех пор, когда в программе объекты ещё не используются, методы уже могут вводиться. Метод, который описан внутри некоторого класса, но вызывается без приложения к конкретному объекту этого класса, называется **статическим.**

Кроме имени и описания, о которых сказано выше, у метода есть ряд других характеристик:

1) Набор модификаторов,
2) Тип возвращаемого значения,
3) Набор аргументов (параметров)

Модификаторы метода.

Для того, чтобы создать статический метод, перед ним надо указать модификатор *static*. Если этого не сделать, то метод можно будет вызывать только в приложении к конкретному объекту данного класса.

Модификатор *public* отвечает за уровень доступа к описываемому методу. Вместо *public* – *private (protect)*, а также может не указываться ничего, тогда будет действовать уровень доступа по умолчанию.

С модификаторами познакомимся подробнее, когда будем создавать свои классы, а пока отметим, что доступ по умолчанию разрешает обращаться к методу из любой части того пакета, в котором метод описан. А уровень *public* открывает доступ к методу откуда угодно. В том числе и из других пакетов.

Метод *Main* обязан иметь уровень доступа *public* как раз потому, что к нему обращается ВМ Java, не являющаяся частью какого-либо пакета.

Кроме этого, существуют другие модификаторы, которые, например, позволяют регулировать работу методов в процессе параллельных вычислений. Или модификатор *native*, позволяющий в java-программе использовать методы, описанные в других ЯП.

Тип возвращаемого значения.

Методы java условно можно разделить на 2 группы: функции и процедуры. К первой группе относятся методы, очень похожие на функции в математическом смысле. В результате своей работы такие методы возвращают в то место программы, из которого они были вызваны, некоторый конкретный результат существующего типа, то есть это может быть целое или вещественное число, логическое значение (*int, double, boolean...*), массив (ссылку на него), объект (ссылку на него). Это значение должно присваиваться переменной подходящего типа или же передаваться какому-либо другому методу в роли аргумента.

В отличие от функций, методы процедурного типа производят какие-либо полезные действия, но не дают законченного результата, который мог бы выражаться в одном конкретном значении или объекте.

Примеры:

double r = Math.random();

/ относится к функциям */*

System.out.println(r);

/ процедура */*

Если бы мы создали метод, который так же, как и *println*, печатал бы текст на экран, но при этом подсчитывал бы количество пробелов в тексте, и возвращал бы этот результат, то мы получили бы функцию. При этом функция продолжала бы выполнять полезные действия,

характерные для процедуры *println*. Соответственно, функция более универсальна, чем процедура.

При создании метода в первую очередь надо определить, будет ли он функцией или процедурой. Для промежуточных вычислений, как правило, используются функции. Для сокращения однотипных кусов кода могут подходить и процедуры.

Все команды, указанные в описании метода после *return*, выполняться уже не будут.

Return без аргумента а можно использовать внутри процедур. Он будет просто досрочно завершать процедуру (аналог *break* для цикла).

Аргументы (параметры)

При вызове метода в него из основной программы может передаваться набор некоторых значений. Для того, чтобы научить метод их обрабатывать, в круглых скобках после имени метода в его описаниях должны быть перечислены пары вида: *тип_аргумента имя_аргумента* через запятую.

Тогда при вызове метода можно будет указать набор значений, соответствующих по типам, описанным аргументом.

Значение, которые передаются методу в момент вызова, называются **фактическими параметрами**, а имена аргументов, которые фигурируют в описании метода – **формальными параметрами**.

Каждый формальный параметр является внутри метода локальной переменной, то есть он недоступен за пределами метода (вне блока его описания). В момент вызова метода фактическое значение копируется в формальный параметр.

В частности, это означает, что, передавая какую-либо переменную базового типа как параметр метода при его вызове, мы не сможем изменить значение этой переменной в основной программе. Если в метод через аргумент передаётся какого-либо объекта или массива, то внутрь метода копируется ссылка на объект или массив. Его адрес в памяти компьютера. Действия, которые мы совершим с массивом или объектом внутри метода, отразятся на состоянии этого массива или объекта в основной программе даже после того, как метод завершит свою работу. Внутри метода мы обращались по тому же адресу и работали с теми же данными в памяти. Если имя фактического параметра совпадает с именем формального параметра, то это не влечёт никакой проблемы: внутри метода имеется локальная переменная, в которую при вызове было скопировано значение одноимённой глобальной. Обращаясь по имени, будем попадать на локальную переменную и никак не сможем добраться до глобальной.

Описание метода.

Метод должен описываться внутри класса, но при этом один метод не описывают внутри другого, то есть метод должен вкладываться непосредственно в блок класса.

Общая схема описания метода.

Модификаторы тип_возвращаемого_значения имя_метода (формальные аргументы) {

//действия, выполняемые методом

//возможно, return

}

Имя метода по традиции должно начинаться с маленькой буквы. Если оно состоит из нескольких слов, их не разделяют пробелом, но каждое следующее слово начинают с заглавной. Имя для метода выбирают так, чтобы было понятно, что он делает.

Примеры:

poiskMax

vyvodMassiva

Рассмотрим несколько примеров:

Пример 1.

```
public static double kvadk (double) {
        double t;
        t=Math.pow(a, 0.5);
        return t;
}
```

Теперь внутри метода *Main* мы сможем использовать наш метод. Например, так:

```
int a = 25;
System.out.println(kvadk(a));
// 5.0
System.out.println(a)
// 25
```

При передаче значений в метод действует **автоприведение**. Если аргумент фактический не соответствует типу формального, то Java пробует привести фактический аргумент к более универсальному типу (в данном случае *int* был приведён к *double*).

Перегрузка методов.

Сигнатурой метода называются его имя и набор параметров.

Java позволяет создавать несколько методов с одинаковыми именами, но разными сигнатурами. Создание метода с тем же именем, но с другим набором параметров называется **перегрузкой**. Какой из перегруженных методов должен выполняться при вызове, Java определяет на основе фактических параметров.

```
void pr( double a) {
        System.out.println(a);
}


void pr (String a) {
```

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


pr (5); pr («5»)
void pr(int[ ] a) {
        for (int i=0; i<a.length; i++) {
System.out.print(a[i]+»»)
        }
}
```

Пример использования метода.

```
Int a = 5;
int [ ] m = {1, 2, 8, 3}
String s = «Мир»;
pr (a) //работает исходный метод
pr (a+s); // 5 мир, работает 1 перегрузка
pr (m); // 1 2 8 3
pr (m+a); // ошибка
```

Переменная *a* не относится к типу *double*, но её обрабатывает исходный метод, поскольку возможно автоприведение из *int* в *double*. В обратном направлении оно невозможно. Если бы метод имел аргумент типа *int*, то с его помощью вещественные числа выводить не получилось бы.

Перегрузка методов реализует такое важное свойство в программировании, как полиморфизм.

Полиморфным называется программный код, который связан с одним общим именем, но имеет разные реализации. Какая из реализаций будет работать, выбирается на основе контекста, в котором имя было упомянуто. Конкретно для методов полиморфными являются их перегрузки, а выбор исполняемой перегрузки происходит по параметрам.

Полиморфизм: одно имя, много форм.

Рекурсия.

Рекурсией называется метод (функция), которая внутри своего тела вызывает сама себя.

Рассмотрим пример – вычисление факториала. Для того, чтобы вычислить n!, достаточно знать и перемножить между собой (n-1)! и n.

Создадим метод, реализующий описанный способ.

```
Int fact (int n) {
        if (n==1) return 1,
```

```
    if (n==2) return 2;
    if (n==3) return 6;
return fact (n-1)*n,
}
```

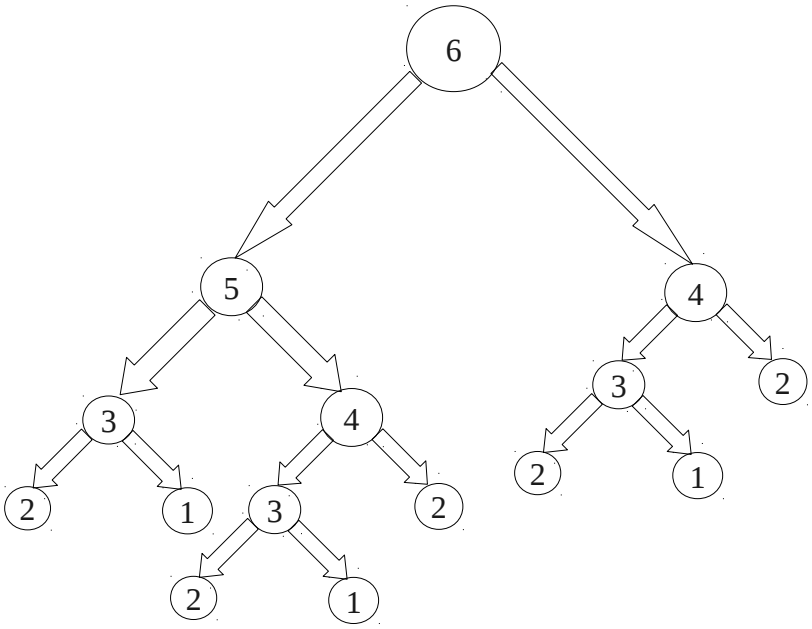
Указанный метод вычисляет факториал натурального числа.

Рассмотрим пример, вычисляющий через рекурсию ***n-ое число Фибоначчи***.

1 1 2 3 5 8 13

```
int fib (int n) {
    if (n==1 || n == 2) return 1;
    return fib (n-2) + fib (n-1);
}
```

Рассмотрим пример работы метода с аргументом 6.

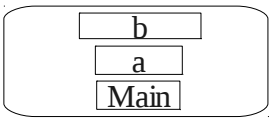


Рекурсивные вычисления часто приводят к необходимости повторять одни и те же действия, что существенно замедляет работу программы. Например, алгоритм вычисления числа Фибоначчи через цикл с хранением 2 предыдущих элементов последовательности намного эффективнее.

Стек вызовов.

В общем случае в текущий момент времени может исполняться только один единственный метод из всей программы. Это значит, что, если метод a устроен таким образом, что в своём теле он вызывает метод *b*, а сам *a* вызывается в *Main*, то при запуске программы управление сначала будет передано методу *Main*, затем методу *a*, затем методу *b*. Метод *b* вернёт результат *b* управление в *a*, *a* вернет результат управления в *Main*, им только потом будут

выполняться основные инструкции, указанные в методе *Main*. Вся иерархия (кто кого вызывал) хранится в специальной области памяти, называемой **стек вызовов**. Элементы в этот фрагмент памяти добавляются по следующему принципу: последний добавленный элемент должен быть извлечён первым. Когда работает метод *b*, имеем:



В связи с этим в процессе рекурсии существует опасность переполнения стека вызовов.

Существует так называемая **сложная рекурсия**, при которой функция *a* вызывает функцию *b*, *b* вызывает *c*, а *c* вызывает *a*.

Объектно – ориентированное программирование.

Объектно – ориентированный подход (парадигма) заключается в следующем: проанализировав условие задачи, необходимо понять, какие объекты в рамках этой задачи должны существовать, какой набор характеристик, существенно для решения реальных объектов и как объекты должны взаимодействовать между собой. После того, как эти вопросы ясны, создаётся программа, в которой порождаются виртуальные объекты. Эти объекты начинают взаимодействовать, изменяя свои характеристики и в нужный момент, остановив взаимодействие и оценив состояние всей системы, можно получить искомый результат.

ООП похоже на моделирование.

Классы.

Класс – шаблон для создания объектов. В классе описывается набор характеристик, которыми смогут обладать объекты данного класса. Эти характеристики называются полями или свойствами. В классе указывается тип и имя каждого свойства, но не его значение (значение у свойств появится в тот момент, когда на основании данного класса будут создаваться объекты.

Также в классе описываются **методы**. Эти методы смогут применяться к объектам данного класса. В теле каждого метода доступны свойства того объекта, для которого метод был вызван. Среди всех методов класса выделяются специальные под названием конструкторы. Они имеют строго то же имя, что и сам класс, перед ними не указывается тип возвращаемого значения или *void*, эти методы могут вызываться только при создании объекта данного класса и не могут применяться к уже существующим объектам. Задача конструктора – дать начальное значение свойствам объекта.

Внутри каждого метода класса доступно ключевое слово *this*. Оно заменяет имя того объекта, для которого метод был вызван. Чтобы для объекта вызвать метод или обратиться к свойству объекта, надо указать его имя, поставить точку и далее указать название метода или свойства.

Рассмотрим ***пример создания и использования класса точек на координатной плоскости.***

```
package tochki;
import java.util.Scanner;
class Point {
/* точку на плоскости можно описать 2 действительными числами, поэтому создадим 2 свойства типа double */
    double x; // абсцисса
    double y;// ордината
    void printPoint () {
/* метод будет печатать на экран координаты точки, ему не потребуются аргументы, поскольку к свойству того объекта, для которого метод вызван, можно обращаться напрямую. */
    System.out.println (« (« + x + «;» +y+») «);
/* Создадим метод, сравнивающий по координатам 2 точки. Он должен отвечать на вопрос, совпадают ли они, а поэтому будет возвращать логическое значение. При этом координаты одной точки мы получим, прямо обращаясь к свойствам того объекта, для которого метод вызван,  а вторую точку мы должны передать через аргумент.*/
    boolean isSame (Point a) {
        if (x == a.x && y == a.y) {
            return true;
            } else {
            return false;
        }
// создадим метод, вычисляющий расстояние между двумя точками.
    double getDistance (Point a) {
        return Math.sqrt ((x-a.x)* (x-a.x)+ (y-a.y)* (y-a.y));
}
/* создадим конструкторы, первый из них не будет иметь аргументов, такой конструктор называется конструктором по умолчанию. Его можно не описывать явно, тогда он будет создан автоматически самой Java, а свойства создаваемого им объекта будут получать те значения, что предусмотрены для данного типа данных по умолчанию: 0 – int, 0.0– double */
Point() {
x = 0;
y = 0;
}
/* Второй конструктор будет иметь 2 параметра и позволит создавать точку с указанными координатами.*/
Point (double a, double u) {
x =a;
```

```
y = u;
}
// Опишем метод, позволяющий вычислить расстояние от данной точки до начала координат.
double getRadius () {
        Point a = new Point (0,0);
        return a.getDistance (this);
        }
}
```

Основные свойства ООП.

Наследование.

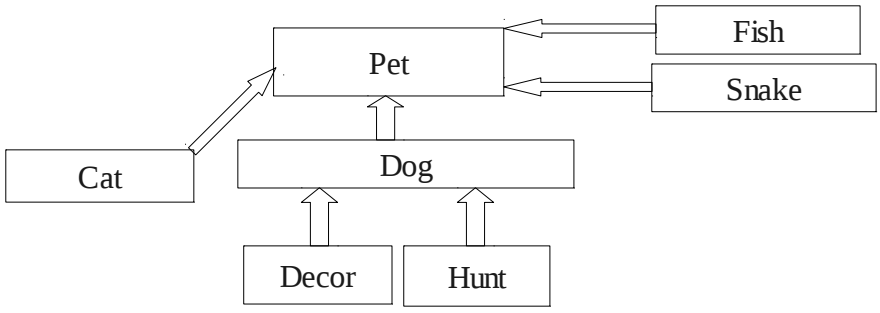
Наследованием называется процесс построения нового класса на основе уже существующего. При этом новый класс называю **подклассом** или **классом потомков**, а старый класс – **суперклассом** или **родителем**. Все свойства и методы, которые были доступны в родителе, переходят к потомку. Дополнительно в потомке можно добавить новые свойства, новые методы, переопределить (перегрузить) методы, доставшиеся из класса родителя.

В Java не допускается множественное наследование, то есть *у любого потомка может быть только один родитель*. Но от одного класса может быть построено сколько угодно много потомков.

Класс, который получен в результате наследования (является чьим-то потомком), может быть унаследован далее (то есть стать и чьим-то родителем). Соответственно, в процессе наследования может быть построена целая иерархия классов, её принято изображать с помощью **UML — диаграмм**: каждый класс изображается прямоугольником с названием класса и классы связываются стрелками в направлении от потомка к родителю.

Наследование позволяет сначала создавать классы с небольшим количеством свойств, описывающих самые общие объекты, а затем строить наследников, где количество свойств будет увеличиваться и, соответственно, описываться смогут более конкретные объекты.

Рассмотрим иерархию классов домашних животных. Её диаграмма:



Для того, чтобы создать новый класс как наследника другого класса, при объявлении нового

указывается слова *extends* и далее имя класса-родителя.

```
Class Pet {
        string name;
        int age;
        double weight;
        boolean hungry;
        void voice () {
                System.ouT.ptint («»):
        }
        void food () {
                hungry = faulse;
        }
}
class Cat extends Pet {
        String poroda;:
        void voice () {
                System.out.print («Mew»)
        }
}
class Fish extends Pet {
}
class Dog extends Pet [
        String poroda;
        void voice () {
                System.out.print («Gaf»);
        }
}
class Decor extends Dog {
        String color;
}
class Hunt extends Dog {
        void voice () {
                System.out.ptint («Rrr»)
        }
}
```

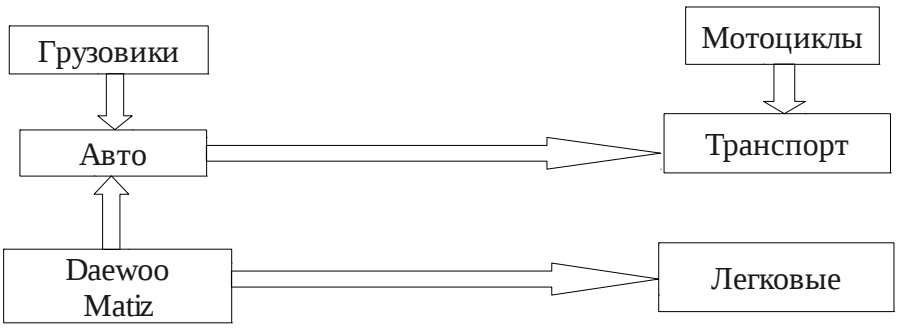
```
class Snake extends Pet
        double length;
        void voice () {
                System.out.ptint («Shh»);
        }
}
```

Свойства *name*, *age* и *weight* будут доступны в любом классе представленной иерархии. Свойство *poroda* существует в 2 классах: *Cat* и *Dog*, но, хоть эти свойства названы одинаковы, между собой они никак не связаны. Свойство *color* существует только в классе Decor и о нём не известно ни в родителе *Dog*, ни в исходном классе *Pet.*

Пример:

```
Dog Sharik = new Dog ();
Sharik.name = «Шарик»;
Cat Murzik = new Cat ();
Decor JuJu = new Decor ();
JuJu.name = «Жу-Жу»;
JuJu.color = «Чёрная»;
Sharik.color = «Белый»; // ошибка
Murzik.voice (); // Mew
```

В процессе наследования нужно сначала нужно разобраться с тем, какие классы являются самыми общими (содержат меньше свойств). Они и станут родителями.



Членами класса называются его свойства и методы вместе. Перед каждым членом в момент его объявления может быть указан один из модификаторов *private, protected, public.*

Эти модификаторы управляют доступом с свойствам и методам, то есть разрешают и запрещают обращаться к ним из разных частей программы. Рассмотрим пример:

```
class A {
        public int x;
```

```
        protected int y;
        private int z;
        void m1() {
                x = 1;
                y = 1;
                z=1;
        }
}
class B extends A {
        void m2() {
                x=2;
                y=2;
                z=2; //ошибка
        }
}
class Main {
        public static void main(String [] args) {
                A a = new A();
                B b = new B ();
                a.x=3;
                a.y=3; // Err
                a.z=3;// Err
                b.x =28;
                n.y=28; // Err
                b.z=28; // Err
        }
}
```

Уровни доступа.

Уровень доступа ***public*** разрешает обращаться к свойству и методу другого класса, в том числе и из другого пакета. Когда перед объявлением мы не указываем никакого модификатора, свойству или классу присваиваются ***уровень доступа по умолчанию***. Он обеспечивает почти такие же широкие возможности, как *Public*, за исключением того, что к свойству или методу нельзя будут обратиться из другого пакета. В представленном примере уровень доступа по умолчанию имеют методы *m1* и *m2*.

Уровень ***protected*** разрешает обращаться к свойству или методу в самом классе, где он был объявлен, а также в его потомках (в том числе в непрямых потомках), но запрещает обращаться к данному члену из остальной части программы (например, из метода *Main).*

Уровень ***private*** разрешает обращаться к члену только в том классе, где он был объявлен.

	Класс	Потомок	Другой класс	Другой пакет
public	+	+	+	+
По умолчанию	+	+	+	-
protected	+	+	-	-
private	+	-	-	-

Инкапсуляция.

Инкапсуляция – принцип проектирования классов в ООП, в соответствии с которым все свойства и методы имеющие промежуточную или служебную роль и не предназначены для непосредственного использования в остальных частях программы, скрываются от внешней программы с помощью ограничения доступа.

Инкапсуляция позволяет пользователю класса ничего не знать о его внутреннем устройстве и при том защищает от нечаянных или умышленных изменений закрытый класс.

Например, мы использовали многие метод класса *Math*, но не имели понятия о том, как они устроены и какие алгоритмы там используются. Нам достаточно было знать имена методов и их аргументы.

В современной практике программирования принято все свойства класса делать закрытыми (private), а для обращения к ним создавать специальные методы: **сеттеры** и **геттеры**.

Пример:

```
class A {
        private int x;
        void setX(int a) {
                x=a;
        }
        int getX() {
                return x;
        }
}
…
A a = new A();
a.x=0: // Err
a.setX(10);
System.out.ptint(a.x); // Err
```

System.out.ptint(a.getX());

Сеттеры и геттеры нужны для того, чтобы контролировать доступ к свойству. С помощью них мы получаем возможность отслеживать каждый момент обращения к свойству и это обращение нужным образом контролировать.

Представим, что мы создаём класс натуральных чисел.

```
class N {
    private int n;
    void setN(int a) {
        if (a>=0) {
            n = a;
        } else {
            n = -a;
        }
    }
}
```

Все свойства и методы с уровнем доступа *по умолчанию* или с уровнем *public* называются открытой частью класса. В соответствии с принципом инкапсуляции для эффективного использования класса достаточно иметь представление о его открытой части (как называются методы, что они делают, аргументы).

Полиморфизм.

С полиморфизмом мы уже сталкивались, когда рассматривали перегрузку методов. В ООП полиморфизм реализуется за счёт переопределения методов в классах-потомках в процессе наследования.

Суть полиморфизма заключается в том, что *под одним и тем же именем скрываются разные варианты программного кода*. Какой из вариантов будет выполняться в момент обращения к имени – выбирается, исходя из контекста этого обращения. Для перегруженных методов контекстом был набор аргументов, а теперь контекстом является класс того объекта, для которого метод вызван.

Пример про домашних животных см. выше. Метод *voice* имеет одно название, но даёт разные результаты в зависимости от того, к какому объекту он применяется.

Основная цель полиморфизма: « если метод называется *print*, то он должен печатать что угодно: для животного – кличку, для окружности – радиус».