Все является объектом

Язык Java является на 100% объектно-ориентированным и основными понятиями для этого языка являются класс и объект.

Класс - модель ещё не существующей сущности (объекта), описанная на языке исходного кода. Фактически он описывает устройство объекта, являясь своего рода чертежом.

Объект - сущность в адресном пространстве вычислительной системы, появляющаяся при создании экземпляра класса. Объект обладает состоянием, поведением и индивидуальностью.

Плюсы ООП

- Возможность повторного использования кода
- Возможность дорабатывать внутреннюю структуру, предоставляя только интерфейс
- Скрытость реализации
- Возможность построения автономных и логически законченных структур данных

Ключевые моменты разработки

- Правильный выбор того, что будет считаться классом.
- Выбор предоставляемого объектом интерфейса
- Выбор механизма повторного использования кода:
  a) Наследование
  b) Композиция

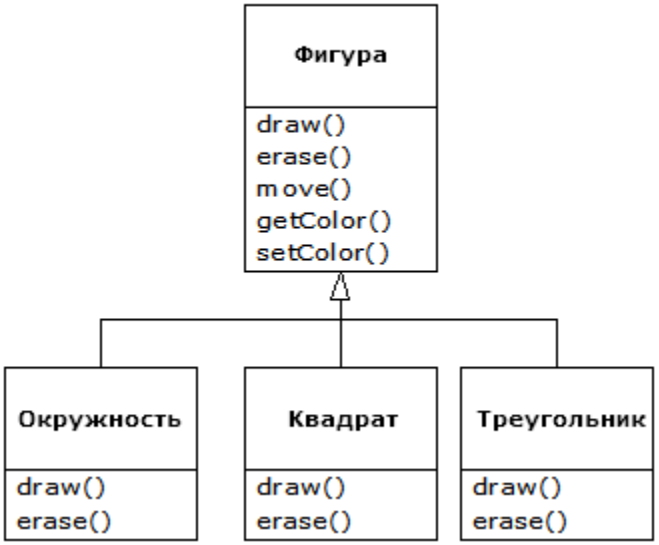
Композиция

Композицию часто называют отношением типа «имеет». Объекты-члены нового класса обычно объявляются закрытыми (***private***), что делает их недоступными для программистов-клиентов, использующих класс. Это позволяет вносить изменения в эти объекты-члены без модификации уже существующего клиентского кода. Вы можете также изменять эти члены во время исполнения программы, чтобы динамически управлять поведением вашей программы.

Наследование

Основная идея наследования - взять готовый класс, «клонировать» его, а затем внести добавления и обновления в полученный клон. Это именно то, что вы получаете в результате наследования, с одним исключением — если изначальный класс (называемый также базовым классом, суперклассом или родительским классом) изменяется, то все изменения отражаются и на его «клоне» (называемом производным классом, унаследованным классом, подклассом или дочерним классом). *Одной из предпосылок использования наследования является необходимость работы с объектом производного класса как с объектом базового.*

Полиморфизм



Возможность обращения к объектам производного типа как к объектам базового позволяет писать код, не зависящий от конкретных типов. Подобный код не зависит от добавления новых типов, а добавление новых типов является наиболее распространенным способом расширения объектно-ориентированных программ для обработки новых ситуаций.

```
void doSomething(Shape
shape)
{
   shape.erase();
  //...
  shape.draw();
}
```

```
Circle circle = new Circle();
Triangle triangle=new Triangle();
Line line = new Line();
doSomething(circle);
doSomething(triangle);
doSomething(line);
```

Полиморфизм и позднее связывание

Имеются два термина, часто используемых, когда речь заходит об объектно-ориентированных языках программирования: раннее и позднее связывание.

В терминах объектно-ориентированного программирования **раннее связывание** означает, что объект и вызов функции связываются между собой на этапе компиляции. Это означает, что вся необходимая информация для того, чтобы определить, какая именно функция будет вызвана, известна на этапе компиляции программы.

Позднее связывание означает, что объект связывается с вызовом функции только во время исполнения программы, а не раньше.

В *Java* позднее связывание производится по умолчанию, и не нужно помнить о необходимости добавления каких-либо ключевых слов для обеспечения полиморфизма.

Работа с памятью

В *Java* используется динамическое создание объектов в области памяти, называемой «кучей» (*heap*). В таком случае количество объектов, их точные типы и время жизни остаются неизвестными до момента запуска программы. Все это определяется «на ходу» во время работы программы. Если понадобится новый объект, он просто создается в «куче» тогда, когда потребуется.

Каждый раз при создании объекта используется ключевое слово ***new*** для построения динамического экземпляра. При создании объекта в куче компилятор не имеет представления о сроках жизни объекта.

При этом возникает проблема освобождения памяти из под объекта, который уже не используется. Проблема заключается в том, что на один объект может быть несколько ссылок, поэтому неизвестно когда нужно память вычищать.

В *Java* существует сборщик мусора, который спроектирован так, чтобы он мог самостоятельно решать проблему освобождения памяти (это не касается других аспектов завершения жизни объекта). Сборщик мусора «знает», когда объект перестает ис-пользоваться, и применяет свои знания для автоматического освобождения памяти.

Объекты
в
Java

Основные моменты

При работе с объектами используются ссылки:

```
Shape sh; // sh –ссылка на объект(но не сам объект)
Sh = new Shape(); // Создаем сам объект
```

Для основных небольших типов Java использует обычный механизм, а именно создает переменную для них и размещает их в стеке.(числовые типы, логический тип, символы). Стоит особо обратить внимание на классы обретки, позволяющее создать в куче объекты типа примитивов. Благодаря автоматической упаковке автоматически преобразуются примитивы к оберткам и наоборот.

```
Character ch = 'x';
```

Область действия определяет как видимость, так и срок жизни имен, определенных внутри нее. Переменная, определенная внутри области действия, доступна только в пределах этой области.

Объекты *Java* имеют другое время жизни в сравнении с примитивами.

```
String s;
  {
    String a="asfsdf";
    s=a;
  }
System.out.println(s);
```

ссылка ***a*** исчезнет в конце области действия. Однако объект типа ***String***, на который указывала ***a***, все еще будет занимать память.

Создание новых классов. Поля

`class ATypeName { /* some code */ }` – описание нового класса

```
ATypeName a = new ATypeName();
```

В класс можно включить две разновидности элементов: поля (*fields*) (иногда называемые переменными класса) и методы (*methods*) (еще называемые функциями класса). Поле представляет собой объект любого типа, с которым можно работать по ссылке, или объект примитивного типа.

Каждый объект использует собственный блок памяти для своих полей данных; совместное использование обычных полей разными объектами класса невозможно.

```
class DataOnly {
  int i;
  double d;
  boolean b;
}
```

```
DataOnly data = new DataOnly();
data.i = 47;
data.d = 1.1;
data.b = false;
```

```
DataOnly data1 = new DataOnly();
data1.i = 53;
data1.d = 22.3;
data1.b = true;
```

Если поле данных относится к примитивному типу, ему гарантированно присваивается значение по умолчанию, даже если оно не было инициализировано явно

Создание новых классов. Методы

Методы в *Java* определяют сообщения, принимаемые объектом.

Методы в Java создаются только как части класса и вызываются для конкретного объекта (кроме static, которые вызываются для класса)

Основные части метода — имя, аргументы, возвращаемый тип и тело.

Список аргументов определяет, какая информация передается методу.

В языке Java все значения передаются по значению, что означает, что метод не может модифицировать значение ни одного параметра передаваемое ему:

```
public static void triple(int x)
{
  x=x*3;
}
int myint=10;
Triple(myint);
```

При вызове метода параметр x инициализируется значением переменной myint. Далее значение записанное в x утраивается но при завершении метода это значение не возвращается, а теряется вместе с x.

Создание новых классов. Методы

```
public static void triple(DataOnly x)
{
  x.i=x.i*3;
}
DataOnly myData = new DataOnly();
myData.i=10;
Triple(myData);
```

При вызове x инициализируется копией значения переменной myData т.е. ссылкой на объект DataOnly. В теле метода идет обращение к полю объекта через ссылку и при этом происходит увеличение значения этого поля. После завершения метода myData ссылается на измененный объект.

```
public static void swap(DataOnly x, DataOnly y)
{
  DataOnly temp=x;
  x=y;
  y=temp;
}
```

Метод может возвращать любой тип, но, если вы не хотите пользоваться этой возможностью, следует указать, что метод возвращает ***void***.

static поля и методы

Ключевое слово **static** используется для того, чтобы сделать элемент класса статическим. Когда что-либо объявляется как ***static***, это означает, что данные или метод не привязаны к определенному экземпляру этого класса. Поэтому, даже если не создаются объекты класса, можно вызвать статический метод или получить доступ к статическим данным.

```
class DataOnly {
  int i;
  double d;
  boolean b;
  static int statVar;
}
```

```
System.out.println(DataOnly.statVar);
DataOnly d1 = new DataOnly();
DataOnly d2 = new DataOnly();
d1.statVar=5;
System.out.println(d2.statVar);
```

```
class MyComparator {
  static int compare(int x,int y){
  if (x>y) return 1;
     else if (x<y) return -1;
        else return 0;}}
```

```
System.out.println(
 MyComparator.compare(3,4));
MyComporator mc = new
   MyComparator();
int i mc.compare(1,2);
```

Применительно к полям ключевое слово ***static*** радикально меняет способ определения данных: статические данные существуют на уровне класса, в то время как нестатические данные существуют на уровне объектов, но в отношении изменения не столь принципиальны. Одним из важных применений ***static*** является определение методов, которые могут вызываться без объектов.

Простейшая программа

```
import java.util.*;
public class MyFirstProgramm {
  public static void main(String[] args) {
  System.out.println(“Hello World");
  }
}
```

Именование классов, методов и полей:

- Имена классов должны записываться с прописной буквы. Если имя состоит из нескольких слов, они объединяются (то есть символы подчеркивания не используются для разделения), и каждое слово в имени начинается с большой буквы
- Для всего остального: методов, полей и ссылок на объекты — используется такой же способ записи, за одним исключением — первая буква идентификатора записывается строчной.

Основные операторы

- Присвоение: int x = 1;
- Математические операции: % - остаток от деления; Также существуют укороченные виды записей, например x*=10;
- Операции инкремента и декремента ++ и --. Они существуют в двух вариантах: постфиксный (a++ сначала результат потом операция) и префиксный (++a сначала операция потом результат).

```
int i1=10;
 int i2=10;
 int b=5;
 int c1 = (++i1)+b;
 int c2 = (i2++)+b;
 System.out.println(c1);
 System.out.println(c2);
 System.out.println(i1);
 System.out.println(i2);
```

- Операции сравнения и равенства/неравенства. Проверка на равенство объектов оператором == не имеет смысла.
- Логические операции && (И) , || (Или), !(not)

Побитовые операции

- Поразрядные операции И(&), ИЛИ(|), НЕ(~), исключающее или (^)

```
int i =10; //1010
 System.out.println(Integer.toBinaryString(i));
 long l =123456789; // 11101011011110011010010101
 System.out.println(Long.toBinaryString(l));
 long p = i&l;
 System.out.println(p);//0
 System.out.println(Long.toBinaryString(p));//0
```

- **Знаковый оператор сдвига влево <<**

Все биты смещаются влево. Число справа дополняется нулем. Операция используется для быстрого умножения на 2. Если оператор применяется к числу, умножение на 2 которого будет больше максимального значения int (2147483647), то в результате будет отрицательное число. Это происходит потому, что крайний левый бит, который отвечает за знак числа, выставляется в единицу, что соответствует отрицательным числам.

- **Знаковый оператор сдвига вправо >>**

Все биты смещаются вправо. Число слева дополняется нулем, если число положительное и единицей, если отрицательное. Операция используется для быстрого деления на 2. Если делится нечетное число, то остаток отбрасывается для положительных чисел и сохраняется для отрицательных.

Побитовые операции

Беззнаковый оператор сдвига >>>

Все биты смещаются вправо, число слева дополняется нулем, даже если операция выполняется с отрицательными числами. Отсюда и название оператора — беззнаковый. В результате применения оператора всегда получается положительное число, т.к. в Java левый бит отвечает за знак числа. Операция так же, как и знаковый оператор сдвига вправо, соответствует делению числа на два за исключением первого сдвига в отрицательном числе.

```
int i = 192;
i       00000000 00000000 00000000 11000000 (192)
i<<1    00000000 00000000 00000001 10000000 (384)
i>>1    00000000 00000000 00000000 01100000 (96)
i>>>1   00000000 00000000 00000000 01100000 (96)
int i = -192;
i       11111111 11111111 11111111 01000000 (-192)
i<<1    11111111 11111111 11111110 10000000 (-384)
i>>1    11111111 11111111 11111111 10100000 (-96)
i>>>1   01111111 11111111 11111111 10100000 (2147483552)
```

Хранение в одной целочисленной переменной нескольких значений

```
int age, height, weight, combined, mask;
age = 28; //00011100
height = 185; //10111001
weight = 80; //01010000
combined = (age) | (height << 8) | (weight << 16);
mask = 255;
System.out.printf("Age: %d, height: %d, weight: %d", mask &
combined, mask & combined >>> 8, mask & combined >>> 16);
```

Простейший ввод-вывод

Для вывода информации в консоль можно воспользоваться методом System.out.println(). Для чтения информации нужно создать объект класса Scanner и связать его со стандартным входным потоком

```
Scanner sc = new Scanner(System.in);
String st= sc.nextLine();//чтение всей строки целиком
String st1 = sc.next(); //чтение одного слова
int i = sc.nextInt(); // чтение целого числа
double d = sc.nextDouble();
```

If – then -else

if(логическое выражение) команда

if(логическое выражение)

команда

else

команда

```
class MyComparator {
  static int compare(int x,int y){
  if (x>y) return 1;
     else if (x<y) return -1;
        else return 0;}}
```

Циклы

Цикл с предусловием:

while(логическое выражение) команда

```
public class Echo {
  public static void main(String[] args)throws IOException {
  BufferedReader stdin = new BufferedReader(new
       InputStreamReader(System.in));
    String s;
    while((s = stdin.readLine()) != null && s.length()!= 0)
     { System.out.println(s);}
}}
```

Цикл с постусловием

do команда while(логическое выражение);

```
public class Echo {
 public static void main(String[] args) throws IOException {
  BufferedReader stdin = new BufferedReader(
   new InputStreamReader(System.in));
  String s="";
  do
  {  System.out.println("Contnue?"); }
  while((s = stdin.readLine()) != null && !s.equals("N"));
}}
```

Циклы

- **Цикл for**

for(инициализация; логическое выражение; шаг) команда;

Теоретически любую из трех частей можно пропустить

```
for(int i =0; ;i<10; i++;)
 {
    System.out.print(i);

}
```

```
int i =0;
for(;i<10;)
 {
     i++;  System.out.print(i);
}
```

```
Int i=0;
for(;;)
 {
    i++; System.out.print(i);
}
```

Массивы

Массив в *Java* гарантированно инициализируется, к нему невозможен доступ за пределами его границ.

```
random rand = new Random(47);
float f[] = new float[10];
for(int i = 0; i < 10; i++)
  f[i] = rand.nextFloat();
```

```
random rand = new Random(47);
float f [][] = new float[10][10];
for(int i = 0; i < 10; i++)
    f[i][i] = rand.nextFloat();
```

f – это ссылка на массив со всеми вытекающими отсюда последствиями

Циклы

Цикл **foreach**

for(тип переменная: множество элементов этого типа)

```
for(float x: f)
  System.out.println(x);
```

```
for(char c: “Hello".toCharArray() )
   System.out.print(c + " ");
```

Важно то, что переменная инициализируется значением из множества а не является ссылкой на элемент. Таким образом при помощи foreach нельзя менять

```
public static int[]
                 range(int from,int to){
  if (from>to)
  return null;
  else
  {
  int[] temp = new int[to-from+1];
  for(int i=from;i<=to;i++)
  temp[i-from]=i;
  return temp;
   }
}
```

```
for(int x:range(1,5))
{
 System.out.println(x);
}
```

Операторы безусловного перехода

return – выход из метода

```
class MyComparator {
  static int compare(int x,int y){
  if (x>y) return 1;
  if (x<y) return -1;
  return 0;}}
```

break и continue – операторы управления циклом. **break –** прерывает выполнение цикла. **continue –** прерывает выполнение текущей итерации

```
for(int i : range(100))
{
  if(i == 74) break; // Выход из цикла
  if(i % 9 != 0) continue; // Следующая итерация
  System.out.print(i + " ");
}
```

```
while(true)
{
   i++;
   int j = i * 7;
   if(j == 777) break;
}
```

Метки

Метка представляет собой идентификатор с последующим двоеточием:

```
label1:
```

- Обычная команда ***continue*** переводит исполнение к началу текущего внутреннего цикла, программа продолжает работу.
- Команда ***continue*** с меткой вызывает переход к метке и повторный вход в цикл, следующий прямо за этой меткой.
- Команда ***break*** завершает выполнение текущего цикла.
- Команда ***break*** с меткой завершает выполнение внутреннего цикла и цикла, который находится после указанной метки.

Метку можно связать с любым оператором или блоком операторов.

```
myLabel:{
  i++;
  System.out.print(i);
  if (i<10) break myLabel;
  System.out.print("aaa");
  }
 System.out.print("bbb");
```

Используя метки, можно выйти из любого блока, но нельзя войти в него. **Использование меток крайне нежелательно.**

Метки

```
int i = 0;
outer: for(; true ;) {
  inner: for(; i < 10; i++)
   {
     print("i = " + i);
     if(i == 2) { print("continue"); continue; }
     if(i == 3) { print("break"); i++; break; }
     if(i == 7) { print("continue outer"); i++; continue outer;}
     if(i == 8) { print("break outer"); break outer; }
     for(int k = 0; k < 5; k++)
     {
          if(k == 3) { print("continue inner"); continue inner;}
     }
   }
}
```

Оператор Switch

switch(целочисленное-выражение)
{ case целое-значение1 : команда; break;
 case целое-значение2 : команда; break;
  case целое-значение3 : команда; break;
  case целое-значение4 : команда; break;
  case целое-значениеб : команда; break; // ..
  default: оператор;
}

```
switch(c) {
 case 'a':
 case 'e':
 case 'i':
 case 'o':
 case 'u': print(“Гласная"); break;
 case 'y':
 case 'w': print(“Иногда гласная"); break;
 default: print(«Согласная");
}
```

В Java SE 7 появилась возможность использовать объект **String** в операторе **switch**

Литература

1. Брюс Эккель Философия Java. 4-е издание
2. Хорстманн К. С., Корнелл Г. -- Java 2. Том 1. Основы
3. Habrahabr.ru
4. Sql.ru
5. http://grepcode.com/project/repository.grepcode.com/java/root/jdk/openjdk/
6. !!! Google.com