Лекция 2

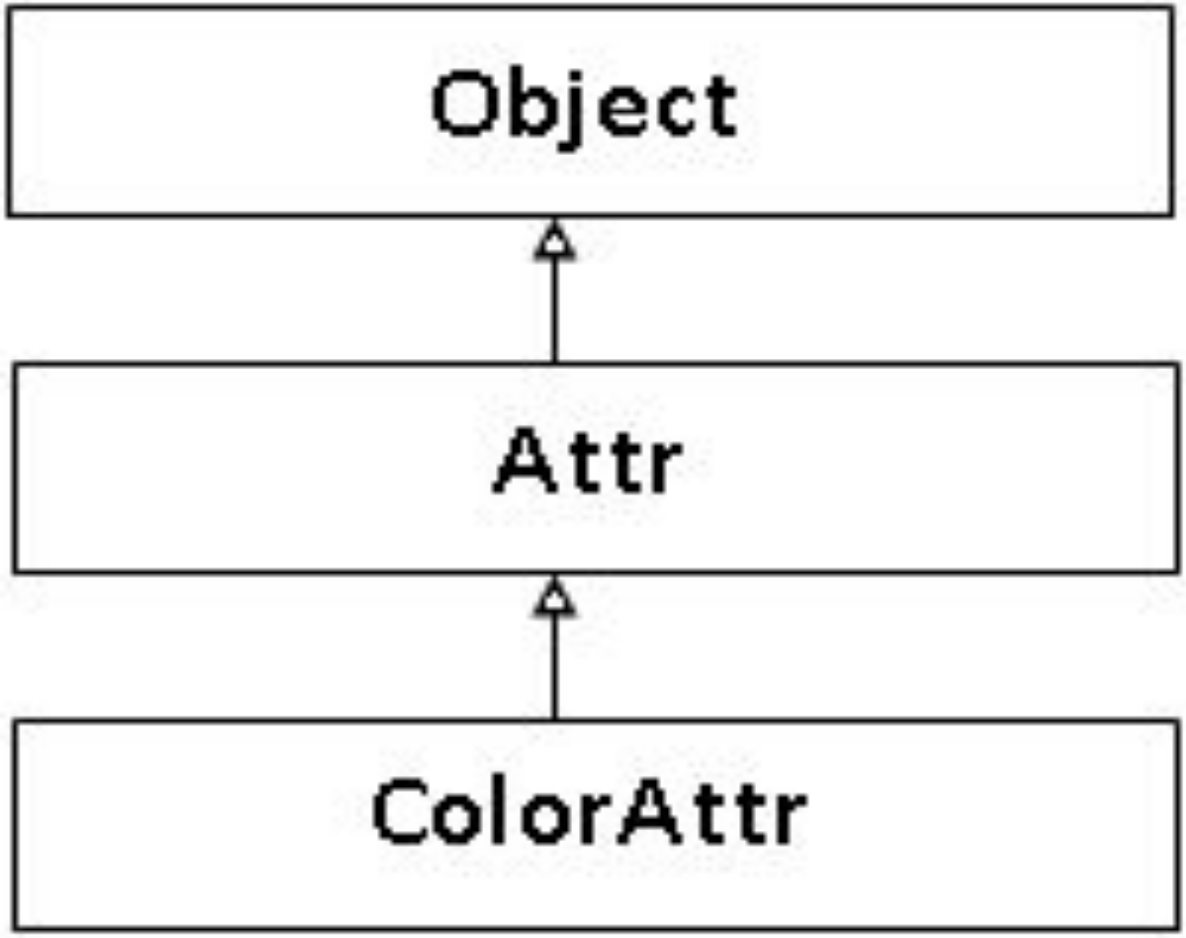
Наследование

Наследование в Java имеет тот же смысл, что и в C++. Однако наследование в Java осуществляется при помощи ключевого слова extends. Рассмотрим пример:

```
class Attr {
  private String name;
  private Object value = null;
  public Attr(String nameOf) { name = nameOf; }
  public Attr(String nameOf, Object valueOf){
    name = nameOf;
    value = valueOf;
  }
public String nameOf() { return name; }
public Object valueOf() { return value; }
……………………………………}
```

```
class ColorAttr extends Attr {
   private ScreenColor myColor;
   public ColorAttr(String name, Object value) {
      super(name, value); //вызов конструктора суперкласса
      decodeColor();
    }
   public ColorAttr(String name) {
     this(name, “transparent”); // вызов первого конструктора
   }
   public ColorAttr(String name, ScreenColor value) {
    super(name, value.toString());
    myColor = value;
   }
   public Object valueOf(Object newValue) {
    // сначала выполнить метод valueOf() суперкласса
    Object retval = super.valueOf(newValue);
    decodeColor();
    return retval;
   }
   ……….
}
```

Иерархия классов для нашего примера выглядит следующим образом(класс Object присуствует в любой иерархии):



Порядок вызова конструкторов в производных классах

При создании объекта всем его полям присваиваются исходные значения по умолчанию в зависимости от их типа. Затем происходит вызов конструктора. Каждый конструктор выполняется за три фазы:

1. Вызов конструктора суперкласса.
2. Присвоение значений полям при помощи инициализаторов.
3. Выполнение тела конструктора.

Рассмотрим пример:

```
class X {
   protected int xMask = 0x00ff;
   protected int fullMask;
 public X() { fullMask = xMask; }
 public int mask(int orig){
     return (orig & fullMask);
 }
}
class Y extends X {
   protected int yMask = 0xff00;
    public Y() {yMask |= fullMask; }
}
```

Рассмотрим значения полей на каждом этапе:

1. Присвоение значений полям по умолчанию.
   xMask=0, fullMask=0, yMask=0.

2. Вызов конструктора класса Y
xMask=0, fullMask=0, yMask=0.
3. Вызов конструктора X
xMask=0, fullMask=0, yMask=0.
4. Инициализация полей X
xMask= 0x00ff, fullMask=0, yMask=0.
5. Выполнение конструктора X
xMask= 0x00ff, fullMask= 0x00ff, yMask=0.
6. Инициализация полей Y
xMask= 0x00ff, fullMask= 0x00ff, yMask= 0xff00.
7. Выполнение конструктора Y
xMask= 0x00ff, fullMask= 0x00ff, yMask= 0xffff

Переопределение методов и скрытие полей

Методы в Java как и в C++ можно переопределять. Расширенный класс может изменить права доступа к методам, унаследованным из суперкласса, но лишь в том случае, если он расширяет их.

Метод, объявленный в суперклассе как protected, может быть повторно заявлен как protected (вполне обычная ситуация) или public, но не как private.

Ограничивать доступ к методам по сравнению с суперклассом бессмысленно, поскольку такое ограничение очень легко обойти: достаточно преобразовать ссылку в супертип с большими правами доступа и использовать ее для вызова метода.

При вызове метода для некоторого объекта его реализация выбирается в зависимости от фактического типа объекта. Рассмотрим следующий пример:

```
class SuperShow {
  public String str = “SuperStr”;
  public void show() {
    System.out.println(“Super.show: ” + str);
  }
}

class ExtendShow extends SuperShow {
  public String str = “ExtendStr”;
  public void show() {
      System.out.println(“Extend.show: ” + str); }

  public static void main(String[] args) {
  ExtendShow ext = new ExtendShow();
  SuperShow sup = ext;
  sup.show(); // вызывается show из класса ExtendShow
  ext.show(); // вызывается show из класса ExtendShow
  System.out.println(“sup.str = ” + sup.str);//печатается SuperStr
  System.out.println(“ext.str = ” + ext.str); // печатается ExtendStr
  }
}
```

Если существующий метод получает параметр типа SuperShow и обращается к str через ссылку на объект-параметр, он всегда будет получать SuperShow.str, даже если методу на самом деле был передан объект типа ExtendShow.

Ключевое слово super

Ключевое слово super может использоваться во всех нестатических методах класса. При доступе к полям или вызове методов ключевое слово super представляет собой ссылку на текущий объект как экземпляр суперкласса.

В вызове вида **super.method()** всегда используется реализация method из суперкласса, а не его переопределенная реализация, которая находится где-то ниже в иерархии классов.

Рассмотрим пример:

```
class That {
  /** вернуть имя класса */
   protected String nm() { return “That”; }
 }

class More extends That {
   protected String nm() { return “More”; }
   protected void printNM() {
        That sref =(That)this;
        System.out.println(“this.nm() = ” + this.nm());
       //Вызывается nm() из More
       System.out.println(“sref.nm() = ” + sref.nm());
       //Вызывается nm() из More
      System.out.println(“super.nm() = ”+super.nm());
     //Вызывается nm() из That
   }
 }
```

Объявление методов и классов с ключевым словом final

Если метод объявлен с атрибутом final, это означает, что ни один расширенный класс не сможет переопределить данный метод с целью изменить его поведение.

Другими словами, данная версия метода является окончательной. Пример:

```
 class A{
   final void f(){……};
}
class B extends A{
  void f(){……..}; //error
}
```

Для класса тоже можно применять final:

final class NoExtending {...}

Класс, помеченный с атрибутом final, не может иметь наследников, а все его методы также неявно являются final

Класс Object

Все классы являются явными или неявными расширениями класса Object и, таким образом, наследуют его методы. Рассмотрим методы класса Object:

1.public boolean equals(Object obj)

Сравнивает объект-получатель с объектом, на который указывает ссылка obj; возвращает true, если объекты равны между собой, и false в противном случае. Если нужно выяснить, указывают ли две ссылки на один и тот же объект, можно сравнить их с помощью операторов == и !=, а метод equals предназначен для сравнения значений.

Реализация метода equals, принятая в Object по умолчанию, предполагает, что объект равен лишь самому себе.

При этом super.equals(this) – вернет true.

2. public int hashCode()

Возвращает хеш-код для данного объекта. Каждому объекту может быть присвоен некоторый хеш-код, используемый при работе с хеш-таблицами.

По умолчанию возвращается значение, которое является уникальным для каждого объекта. Оно используется при сохранении объектов в таблицах Hashtable.

3. **protected Object clone() throws**

CloneNotSupportedException

Возвращает дубликат объекта. Дубликатом является новый объект- копия объекта, для которого вызывался метод clone.

4. public final Class getClass()

Возвращает объект типа Class, который соответствует классу данного объекта.

5. protected void finalize() throws Throwable

Завершающие операции с объектом, осуществляемые во время сборки мусора.

Методы hashCode и equals должны переопределяться, если необходима другая концепция равенства объектов, отличающуюся от принятой в классе Object.

По умолчанию считается, что два различных объекта не равны между собой, а их хеш-коды не должны совпадать.

Абстрактные классы и методы

Абстрактный класс необходим, когда некоторое поведение характерно для большинства или всех объектов данного класса, но некоторые аспекты имеют смысл лишь для ограниченного круга объектов, не составляющих суперкласса.

В Java такие классы объявляются с ключевым словом abstract, и каждый метод, не реализованный в классе, также объявляется abstract.

Рассмотрим пример:

```
abstract class Benchmark {
  abstract void benchmark();
  public long repeat(int count) {
    long start = System.currentTimeMillis();
    for (int i = 0; i < count; i++) {
                    benchmark(); }
    return (System.currentTimeMillis() — start); }}

class MethodBenchmark extends Benchmark {
  public void benchmark() { ….} //Метод должен быть
   переопределен обязательно, т.к. в Benchmark он
   абстрактный.
  public static void main(String[] args) {
    int count = Integer.parseInt(args[0]);
    long time = new MethodBenchmark().repeat(count);
    System.out.println(count + “ methods in ” + time
                                          + “milliseconds”);
}}
```

Дублирование объектов

Метод Object.clone помогает производить дублирование объектов.

При дублировании возвращается новый объект, исходное состояние которого копирует состояние объекта, для которого был вызван метод clone.

Все последующие изменения, вносимые в объект-дубль, не изменяют состояния исходного объекта.

Самая простая возможность создать дублируемый класс — объявить о реализации в нем интерфейса Cloneable:

class MyClass extends AnotherClass implements Cloneable {...}

Метод clone в интерфейсе Cloneable имеет атрибут public, следовательно, метод MyClass.clone, унаследованный от Object, также будет public.

Рассмотрим пример:

```
public class IntegerStack implements Cloneable {
   private int[] buffer;
   private int top;
   public IntegerStack(int maxContents) {
      buffer = new int[maxContents];
      top = -1;  }
   public void push(int val) {
    buffer[++top] = val; }  }
```

Если копировать объект класса IntegerStack прямо, например:

```
IntegerStack first = new IntegerStack(2);
first.push(2);
first.push(9);
IntegerStack second =
                    (IntegerStack)first.clone();
```

то изменяя second мы меняем first.

Правильное копирование – это переопределение метода clone в классе IntegerStack.

Т.е. в классе IntegerStack определяем метод clone:

```
public Object clone() {
 try {
  IntegerStack nObj =(IntegerStack)super.clone();
  nObj.buffer = (int[])buffer.clone();
   return nObj; }
 catch (CloneNotSupportedExeption e) {
// Не может произойти - метод clone()
//поддерживается классом IntegerStack, так и
                                        массивами
  throw new InternalError(e.toString());
  }
}
```

Можно потребовать, чтобы метод clone поддерживался во всех наследниках данного класса, — для этого следует переопределить метод clone так, чтобы в его сигнатуру не входило объявление о возбуждении исключения CloneNotSupportedException.

В результате подклассы, в которых реализуется метод clone, не смогут возбуждать исключение CloneNotSupported Exception, поскольку методы подкласса не могут вводить новые исключения.

Аналогично, если метод clone будет всегда возбуждать исключение CloneNotSupportedException, то в этом случае это эквивалентно тому, что класс не поддерживает клонирование.