Полиморфизм

```
class Stringed extends Instrument {
   public void play(Note n) {
       print("Stringed.play() " + n); } }
class Brass extends Instrument {
   public void play(Note n) {
       print("Brass.play() " + n); } }
public class Music2 {
  public static void tune(Wind i) {
      i.play(Note.MIDDLE_C); }
  public static void tune(Stringed i) {
      i.play(Note.MIDDLE_C); }
  public static void tune(Brass i) {
      i.play(Note.MIDDLE_C); }
  public static void main(String[] args) {
      Wind flute = new Wind();
      Stringed violin = new Stringed();
      Brass frenchHorn = new Brass();
      tune(flute);
      tune(violin);
      tune(frenchHorn); } }
```

Полиморфизм

полиморфизм – свойство языка программирования, позволяющее единообразно обрабатывать данные разных типов.

```
public enum Note { MIDDLE_C, C_SHARP, B_FLAT;}
class Instrument { public void play(Note n) {
    print("Instrument.play()"); } }
public class Wind extends Instrument {
   public void play(Note n) {
      System.out.println("Wind.play() " + n); } }
public class Music {
    public static void tune(Instrument i) {
            i.play(Note.MIDDLE_C); }
    public static void main(String[] args) {
       Wind flute = new Wind(); tune(flute);} }
```

«переопределение» закрытых методов

```
public class PrivateOverride {
    private void f() { print("private f()"); }
    public static void main(String[] args) {
         PrivateOverride po = new Derived(); po.f(); } }
class Derived extends PrivateOverride {
     public void f() { print("public f()"); } }
```

Наследование и поля классов

```
class ClassA{
  private int i=0;
  public int j;
  public int getI(){return i;}
  public void setI(int i){this.i=i;}}
class ClassB extends ClassA{
  public int i=5;
  public String j;
  public int getI(){ return i;}
/*public void setI(int i){
this.i=i;
}*/
public int returnJ(){return super.j;}}
public class TestHiding {
public static void main(String[] arg){
   ClassB b = new ClassB();
   b.setI(10);
   ((ClassA)b).j=0;
   System.out.println(b.getI());
}}
```

Инициализация и завершение при наследовании

конструкторы для сложного объекта вызываются в следующей по-следовательности:

- Сначала вызывается конструктор базового класса. Этот шаг повторяется рекурсивно: сначала конструируется корень иерархии, затем следующий за ним класс, затем следующий за этим классом класс и т. д., пока не достигается «низший» производный класс.
- Проводится инициализация членов класса в порядке их объявления.
- Вызывается тело конструктора производного класса.

Очередность завершения должна быть обратной порядку инициализации в том случае, если объекты зависят друг от друга. Для полей это означает порядок, обратный последовательности объявления полей в классе (инициализация соответствует порядку объявления).

Инициализация при наследовании

```
class Glyph {
    void draw() { print("Glyph.draw()"); }
    Glyph() { print("Glyph() before draw()");
        draw(); print("Glyph() after draw()"); } }
class RoundGlyph extends Glyph {
    private int radius = 1;
    RoundGlyph(int r) { radius = r;
      print("RoundGlyph.RoundGlyph(),radius=" + radius); }
   void draw() {
      print("RoundGlyph.draw(), radius=" + radius); } }
public class PolyConstructors {
    public static void main(String[] args) {
          new RoundGlyph(5);}}
```

При написании конструктора руководствуйтесь следующим правилом: не пытайтесь сделать больше для того, чтобы привести объект в нужное состояние, и по возможности избегайте вызова каких-либо методов. Единственные методы, которые можно вызывать в конструкторе без опаски — неизменные (***final***) методы базового класса.

Клонирование

Абстрактные классы

Абстрактный класс — базовый класс, который не предполагает создания экземпляров.

Абстрактный класс создается для работы с набором классов через общий интерфейс. Классы, содержащие абстрактные методы, тоже должны помечаться ключевым словом ***abstract*** (в противном случае компилятор выдает сообщение об ошибке).

```
abstract class Instrument {
   private int i;
   public abstract void play(Note n);
   public String what() { return "Instrument"; }
  public abstract void adjust();
}
class Wind extends Instrument{
  @Override
  public void play(int n) {…}
  @Override
  public void adjust() {…}
}
```

Интерфейсы

Интерфейс это конструкция языка программирования Java, в рамках которой могут описываться только абстрактные публичные (abstract public) методы и статические константы свойства (final static). То есть также, как и на основе абстрактных классов, на основе интерфейсов нельзя порождать объекты.

```
interface Instruments {
    final static String key = "До мажор";
    public void play();
}
class Wind implements Instrument{
@Override
public void play() {
}
}
…..
static void tune(Instrument i) {i.play(); }
……
Instrument myInstr = new Wind();
tune(myInstr)
```

Абстрактный класс VS Интерфейс

```
abstract class SomethingDoer{
abstract void doSomething(Object obj);
}
class StringDoer extends SomethingDoer{
@Override
void doSomething(Object obj) {
System.out.println((String)(obj));
}
}
class IntDoer extends SomethingDoer{
@Override
void doSomething(Object obj) {
System.out.println((Integer)(obj));
}
}
class DoerUser{
public void UseDoer(SomethingDoer doer, Object obj){
doer.doSomething(obj);
}}
```

Абстрактный класс VS Интерфейс

```
abstract class OtherDoer{
abstract void doSomething(Object obj);}
class DoubleDoer extends OtherDoer{
@Override
void doSomething(Object obj) {
System.out.println((String)(obj));
}}
```

```
interface SomethingDoer{
void doSomething(Object obj);}
class StringDoer implements SomethingDoer{
public void doSomething(Object obj) {
System.out.println((String)(obj));}}

class DoubleDoer extends OtherDoer implements
SomethingDoer{
@Override
public void doSomething(Object obj) {
System.out.println((String)(obj));
}}
```

Абстрактный класс VS Интерфейс

Критерий сравнения	**Абстрактный класс**	**Интерфейс**
Наследование	Любой класс может наследовать только один абстрактный класс.	Любой класс может реализовывать (имплементировать, наследовать) множество интерфейсов.
Модификаторы доступа методов	К неабстактным членам класса применимы любые модификаторы. НО! Абстрактные методы (имеющие модификатор **abstract**) могут иметь модификатор либо **public**, либо **protected**.	Методы в интерфейсе могут иметь модификаторы только **public** и **abstract**. По умолчанию они уже **public abstract**.

Абстрактный класс VS Интерфейс

Данные	Абстрактный класса может содержать любые поля: статические и экземплярные, константы, **private/protected/public**.	Интерфейс может содержать только общедоступные константы (**public final static** int NOT_PI_CONST = -1);
Наличие реализации	Абстактный класс допускает реализацию методов.	Интерфейс не может содержать никакой реализации методов.
Возможность описать конструктор	В абстрактом классе можно описать конструктор (или несколько конструкторов).	В интерфейсе нельзя описать конструктор.

Адаптеры

Довольно часто модификация тех классов, которые нужно использовать, невозможна. В таких ситуациях применяется паттерн «адаптер»: вы пишете код, который получает имеющийся интерфейс, и создаете тот интерфейс, который вам нужен:

```
abstract class OneMoreDoer{
  abstract void doSomething(DataOnly obj);}
class WriteDataDoer extends OneMoreDoer{
  void doSomething(DataOnly obj) {
  System.out.println(DataOnly.i);}}
class WriteDoerAdapter implements SomethingDoer{
  WriteDataDoer wd;
  WriteDoerAdapter(WriteDataDoer d){wd=d;}
  public void doSomething(Object obj) {
      wd.doSomething((DataOnly)obj);}}
……..
DoerUser doer = new DoerUser();
WriteDataDoer wd = new WriteDataDoer();
doer.UseDoer(new WriteDoerAdapter(wd), new DataOnly());
```

Множественное наследование

В Java поддерживается множественное наследование интерфейсов. Интерфейсов может быть сколько угодно, причем к ним можно проводить восходящее преобразование.

```
interface CanFight {void fight();}
interface CanSwim {void swim();}
interface CanFly {void fly();}
class ActionCharacter {public void fight() {}}
class Hero extends ActionCharacter
    implements CanFight, CanSwim, CanFly {
  public void swim() {}
  public void fly() {}
}
class Adventure {
  public static void t(CanFight x) { x.fight(); }
  public static void u(CanSwim x) { x.swim(); }
  public static void v(CanFly x) { x.fly(); }
  public static void w(ActionCharacter x) { x.fight(); }
  public static void main(String[] args) {
    Hero h = new Hero();
    t(h);  u(h); v(h); w(h); }}
```

Расширение интерфейсов

Наследование позволяет легко добавить в интерфейс объявления новых методов, а также совместить несколько интерфейсов в одном.

```
interface Monster {void menace();}
interface DangerousMonster extends Monster {
  void destroy();}
interface Lethal {
  void kill();}
class DragonZilla implements DangerousMonster {
  public void menace() {}
  public void destroy() {}}
interface Vampire extends DangerousMonster, Lethal {
  void drinkBlood();}
class VeryBadVampire implements Vampire {
  public void menace() {}
  public void destroy() {}
  public void kill() {}
  public void drinkBlood() {}
}
```

Конфликты имен

```
interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
class C { public int f() { return 1; } }
class C2 implements I1, I2 {
  public void f() {}
  public int f(int i) { return 1; }}
class C3 extends C implements I2 {
  public int f(int i) { return 1; }}
class C4 extends C implements I3 {
  public int f() { return 1; }}
//! class C5 extends C implements I1 {}
//! interface I4 extends I1, I3 {}
```

Поля в интерфейсах

```
public interface RandVals {
  Random RAND = new Random(47);
  int RANDOM_INT = RAND.nextInt(10);
  long RANDOM_LONG = RAND.nextLong() * 10;
}
```

Внутренние классы

Определение класса может размещаться внутри определения другого класса. Такие классы называются внутренними (*inner class*).

```
class MainClass{
  class InnerA{
    int i;
    InnerA(int i){
      this.i=i;
      System.out.print("A");}}
  class InnerB{
    String st="B";
    InnerB(){
    System.out.print("B");}}
   public void doSomething(){
     InnerA a = new InnerA(1);
     InnerB b = new InnerB();
}}
```

```
class InnerA{
 int i;
 InnerA(int i){
  this.i=i;
  System.out.print("A");}}
class InnerB{
 String st="B";
 InnerB(){
 System.out.print("B");}}
class MainClass{
public void doSomething(){
 InnerA a = new InnerA(1);
 InnerB b = new InnerB();
}}
```

Внутренние классы

```
interface Selector {
  boolean end();
  Object current();
  void next();}
class Sequence {
  private Object[] items;
  private int next = 0;
  public Sequence(int size) { items = new Object[size]; }
  public void add(Object x) {
    if(next < items.length) items[next++] = x;}
  private class SequenceSelector implements Selector {
    private int i = 0;
    public boolean end() { return i == items.length; }
    public Object current() { return items[i]; }
    public void next() { if(i < items.length) i++; }
  }
  public Selector selector() {
    return new SequenceSelector();
  }}
```

Внутренние классы

```
public class DotThis {
  void f() {…}
  public class Inner {
    public DotThis outer() {
      return DotThis.this;}}
  public Inner inner() {
     return new Inner(); }
  …
  DotThis dt = new DotThis();
  DotThis.Inner dti = dt.inner();
  dti.outer().f();}}
```

```
public class DotNew {
 public class Inner {}
}

…….

 DotNew dn =
          new DotNew();
 DotNew.Inner dni =
        dn.new Inner();
```

При создании объекта внутреннего класса указывается не имя внешнего класса ***DotNew***, как можно было бы ожидать, а имя объекта внешнего класса.

Локальные классы

Декларируются внутри методов основного класса. Могут быть использованы только внутри этих методов. Имеют доступ к членам внешнего класса. Имеют доступ как к локальным переменным, так и к параметрам метода при одном условии - переменные и параметры используемые локальным классом должны быть задекларированы final. Не могут содержать определение (но могут наследовать) статических полей, методов и классов (кроме констант). Для создания внутренних классов можно выделить две причины:

- как было показано ранее, вы реализуете некоторый интерфейс, чтобы затем создавать и возвращать ссылку его типа;
- вы создаете вспомогательный класс для решения сложной задачи, но при этом не хотите, чтобы этот класс был открыт для посторонних.

Локальные классы

```
class OuterClass{
  public OuterClass(){}
  private int outerField;
  //InnerClass inner; // Error
  void methodWithLocalClass (final int parameter)
  {
    int notFinal = 0;
    class InnerClass {
       final static int i =5;
       int getOuterField()
       {
         return OuterClass.this.outerField;
        }
        // notFinal++; // Error
         int getParameter()
         {
            return parameter;
         }
     };
     InnerClass innerInsideMehod; }}
```

Локальные классы

```
interface MyInter{
int getOuterField();
int getParameter();}
class OuterClass{
  public OuterClass(){}
  private int outerField;
  MyInter methodWithLocalClass (final int parameter)
  {
     class InnerClass implements MyInter{
       final static int i =5;
       public int getOuterField(){
           return OuterClass.this.outerField;
       }
       public int getParameter(){return parameter; }
     }
     return new InnerClass();
   }}
OuterClass o = new OuterClass();
System.out.print(o.methodWithLocalClass(1).getParameter())
```

Анонимные классы

```
class MBase {
 void method1() {
    println("Base");
  }
  void method2() {}
}
```

```
class A {
  void g() {
   MBase bref = new MBase() {
      void method1()
         {println("Anonim");} };
    bref.method1();}}
```

Анонимные (безымянные) классы декларируются внутри методов основного класса. Могут быть использованы только внутри этих методов. В отличие от локальных классов, анонимные классы не имеют названия. Главное требование к анонимному классу - он должен наследовать существующий класс или реализовывать существующий интерфейс. Не могут содержать определение (но могут наследовать) статических полей, методов и классов (кроме констант).

Анонимные классы

```
abstract class MBase {
  int x;
  public MBase(int i) {
    System.out.println("Base constructor, i = " + i);
  }
  public abstract void f();}
public class AnonymousConstructor {
  public static MBase getBase(final int i) {
    return new MBase(i) {
      int b;
      { System.out.println("Inside instance initializer");
        b= i; }
      public void f() {
      System.out.println("In anonymous f()"+b);
      }};
  }
  public static void main(String[] args) {
    MBase base = getBase(47);
    base.f();
  }}
```

Анонимные классы

```
interface Sender{void send();}
class SenderFactory{
  public static Sender getMailSender(final String toDest){
    return new Sender(){
      public void send() {
         String Destination = toDest;
         String From = "xxx@yandex.ru";
         //Some code to Send Mail
         System.out.println("mail send");}};}
  public static Sender gePhoneSender(final int number){
    return new Sender(){
    public void send() {
    int MyNumber = number;
    //Some code to send by sms
    System.out.println("SMS send");}};}}
public class TestAnonim {
   public static void main(String[] args) {
      SenderFactory.getMailSender("yyy@gmail.com").send();
      SenderFactory.gePhoneSender(123456).send();
}}
```

Вложенные(nested) классы

Декларируются внутри основного класса и обозначаются ключевым словом static. Не имеют доступа к членам внешнего класса за исключением статических. Может содержать статические поля, методы и классы, в отличие от других типов внутренних классов.
Применение статического внутреннего класса означает следующее:

- для создания объекта статического внутреннего класса не нужен объект внешнего класса;
- из объекта вложенного класса нельзя обращаться к нестатическим членам внешнего класса.

```
class OuterClass {
  public OuterClass() {}
  private int outerField;
  static int staticOuterField;
  static class InnerClass { int getOuterField() {
     return OuterClass.this.outerField;//Error }
  int getStaticOuterField() {
    return     OuterClass.staticOuterField; } } }
```

Использование внутренних классов

•У внутреннего класса может существовать произвольное количество экземпляров, каждый из которых обладает собственной информацией состояния, не зависящей от состояния объекта внешнего класса.

•Один внешний класс может содержать несколько внутренних классов, по-разному реализующих один и тот же интерфейс или наследующих от единого базового класса.

•Место создания объекта внутреннего класса не привязано к месту и времени создания объекта внешнего класса.

•Внутренний класс не использует тип отношений классов «является тем-то», способных вызвать недоразумения; он представляет собой отдельную сущность.

```
interface Selector {
  boolean end();
  Object current();
  void next();}
```

Использование внутренних классов

```
public class Sequence {
  private Object[] items;
  private int next = 0;
  public Sequence(int size) { items = new Object[size]; }
  public void add(Object x) {
    if(next < items.length)
      items[next++] = x;}
  private class SequenceSelector implements Selector {
    private int i = 0;
    public boolean end() { return i == items.length; }
    public Object current() { return items[i]; }
    public void next() { if(i < items.length) i++; }}
  private class ReverseSelector implements Selector {
        private int i = items.length-1;
        public boolean end() {return i<0;}
        public Object current() { return items[i];}
        public void next() {if (i>=0) i--;}}
 public Selector selector(){return new SequenceSelector()}
  public Selector reverseSelector(){
       return new ReverseSelector();}
```

Использование внутренних классов

```
public static void main(String[] args) {
    Sequence sequence = new Sequence(10);
    for(int i = 0; i < 10; i++)
      sequence.add(Integer.toString(i));
    Selector selector = sequence.reverseSelector();
    while(!selector.end()) {
      System.out.print(selector.current() + " ");
      selector.next();
    }
  }
```

Обратный вызов

```
Button btn = (Button)findViewById(R.id.btnCancel);
            btn.setOnClickListener(new OnClickListener() {
                  public void onClick(View v) {
                        setResult(RESULT_CANCELED);
                        finish();

                  }
            });
```

Обратный вызов

Фу́нкция обра́тного вы́зова— передача исполняемого кода в качестве одного из параметров другого кода. Обратный вызов позволяет в функции исполнять код, который задаётся в аргументах при её вызове.

```
interface IntDoer{
  int DoSomething(int i);}
public class TestCallBack{
  public void DoSomeWithInt(IntDoer doer){
  for(int i=1;i<10;i++)
  System.out.println(doer.DoSomething(i));}
  public static void main(String args[]){
     TestCallBack cb = new TestCallBack();
     cb.DoSomeWithInt(new IntDoer() {
         public int DoSomething(int i) {return i+1;}
     });
     cb.DoSomeWithInt(new IntDoer() {
        public int DoSomething(int i) {return i*2;}
     });
}}
```

Замыкание

Замыкание (англ. *closure*)— функция, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции и не в качестве её параметров (а в окружающем коде).

```
interface SomeDoer{void DoSomething(int i);}
class HardDoer{
  void DoSomeWork(SomeDoer doer){//SomeWorkHere
      int i=5;
      doer.DoSomething(i);}}
public class TestClosure {
  private int a;
  private void testDoer(){
  HardDoer hd = new HardDoer();
  hd.DoSomeWork(new SomeDoer() {
     public void DoSomething(int i) {//Somecode
         a=i;}});
  System.out.print(a);}
  public static void main(String[] args){
     TestClosure t = new TestClosure();
     t.testDoer();
}}
```

Обратный вызов

```
interface InterestingEvent{
     public void interestingEvent();}
public  class EventNotifier{
    private InterestingEvent ie;
    private boolean somethingHappened;
    public EventNotifier (InterestingEvent event){
      ie = event; // Save the event object for later use.
      somethingHappened = false;}
    public void doWork (){
      if (somethingHappened){ie.interestingEvent ();}}
    public static void main(String[] args) {
      CallMe me = new CallMe();
      EventNotifier event = new EventNotifier(me);
      event.doWork();}}
class CallMe implements InterestingEvent{
     private EventNotifier en;
     public CallMe (){}
     public void interestingEvent (){
      // Do something...
     }}
```

Наследование внутренних классов

Так как конструктор внутреннего класса связывается со ссылкой на окружающий внешний объект, наследование от внутреннего класса получается чуть сложнее, чем обычное. Проблема состоит в том, что «скрытая» ссылка на объект объемлющего внешнего класса должна быть инициализирована, а в производном классе больше не существует объемлющего объекта по умолчанию. Для явного указания объемлющего внешнего объекта применяется специальный синтаксис:

```
class WithInner {class Inner {}}
public class InheritInner extends WithInner.Inner {
  //! InheritInner() {} // Не компилируется
  InheritInner(WithInner wi) {wi.super(); }

public static void main(String[] args) {
    WithInner wi = new WithInner();
    InheritInner ii = new InheritInner(wi); }}
```

Переопределение внутренних классов

```
class A{
  class Inner{
   Inner(){System.out.println("InnerA");}
   void print(){System.out.println("InnerA is printing");}
  }
  A(){System.out.println("ClassA");Inner i = new Inner();}
  void DoSomething(){(new Inner()).print();}}
class B extends A{
  class Inner{
   Inner(){System.out.println("InnerB");}
   void print(){System.out.println("InnerB is printing");}
  }}
public class InheritInner{
  public static void main(String[] args){
    B b = new B();
    b.DoSomething();
  }}
```

«переопределение» внутреннего класса, как если бы он был еще одним методом внешнего класса, фактически не имеет никакого эффекта:

Переопределение внутренних классов

```
class A{
  class Inner{
   Inner(){System.out.println("InnerA");}
   void print(){System.out.println("InnerA is printing");}
  }
  private Inner i=new Inner();
  A(){System.out.println("ClassA"); }
  void doSomething(){i.print();}
  void setInner(Inner myI){i=myI;}
}
class B extends A{
  class Inner extends A.Inner{
   Inner(){System.out.println("InnerB");}
   void print(){System.out.println("InnerB is printing");}
  }
  B(){setInner(new Inner());}
}
public class InheritInner{
public static void main(String[] args){
    B b = new B();b.doSomething();}}
```

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