Lesson8
完善类的设计

主讲老师：申雪萍

主要内容

- 多态（overview）
- 对象向上映射和向下映射（Upcasting和downcasting）
- Object类
- 操作符“==”与对象的equals()方法
- 内部类
- 匿名类
- 最终方法、最终类
- 复杂软件的管理（package和import的使用）
- 高内聚、低耦合

多态的实现条件（overview）

- Java实现多态有三个必要条件：继承、方法覆盖、向上转型。
  ① 继承：在多态中必须存在有继承关系的子类和父类。
  ② 覆盖：子类对父类中某些方法进行重新定义，在调用这些方法时就会调用子类的方法。
  ③ 向上转型：在多态中需要将子类的引用赋给父类对象，只有这样该引用才能够具备调用父类的方法和子类的方法的能力。

或者进化一下。。

① 多态的前提是必须有子父类关系或者类实现接口关系，否则无法完成多态。

② 使用多态后的父类引用变量调用方法时，会调用子类重写后的方法。

多态的三种形式

- 普通类多态定义的格式
- 抽象类多态定义的格式
- 接口多态定义的格式

1、 普通类多态定义的格式 （继承和覆盖）

- 父类 变量名 = new 子类();

```
class Fu {}
class Zi extends Fu {}

//类的多态使用
Fu f = new Zi();
```

2、 抽象类多态定义的格式 （抽象类，继承和覆盖）

```
abstract class Fu {
    public abstract void method();
}
class Zi extends Fu {
     public void method(){
          System.out.println("重写父类抽象方法");
      }
}

//类的多态使用
Fu fu= new Zi();
```

3、接口多态定义的格式 （接口和实现接口的方法）

```
interface Fu {
    public abstract void method();
}

class Zi implements Fu {
    public void method() {
            System.out.println("重写接口抽象方法");
    }
}

//接口的多态使用
Fu fu = new Zi();
```

注意：

- 同一个父类的方法会被不同的子类重写。在调用方法时，调用的为各个子类重写后的方法。

```
Person p1 = new Student();
Person p2 = new Teacher();

p1.work(); //p1会调用Student类中重写的work方法
p2.work(); //p2会调用Teacher类中重写的work方法
```

多态出现后会导致子父类中的成员变量有微弱的变化。

```
class Fu {
    int num = 4;//没有这句会编译失败
}
class Zi extends Fu {
    int num = 5;
}
class Demo {
    public static void main(String[] args)  {
        Fu f = new Zi();
        System.out.println(f.num);
        Zi z = new Zi();
        System.out.println(z.num);
    }
}
```

```
4
5
```

总之：多态中成员变量

- 当子父类中出现同名的成员变量时，调用该变量时：

  1、编译时期：参考的是引用型变量所属的类中是否有被调用的成员变量。没有，编译失败。

  2、运行时期：也是调用引用型变量所属的类中的成员变量。

  简单记：**编译**和**运行**都参考等号的左边。

  编译运行看左边。

多态出现后会导致子父类中的成员方法有微弱的变化

```
package com.buaa.fuzi;
class Father {
    int num = 4;
    void show() {     //没有这个方法，编译失败
        System.out.println("Fu show num");
    }
}
class Son extends Father {
    int num = 5;
    void show() { //重写父类方法
            System.out.println("Zi show num");
    }
    void show_1(){
            System.out.println("Zi show show_1");
    }
}
class Demo1 {
    public static void main(String[] args)  {
        Father f = new Son();
        f.show();
        //f.show_1();
    }
}
```

```
Zi show num
```

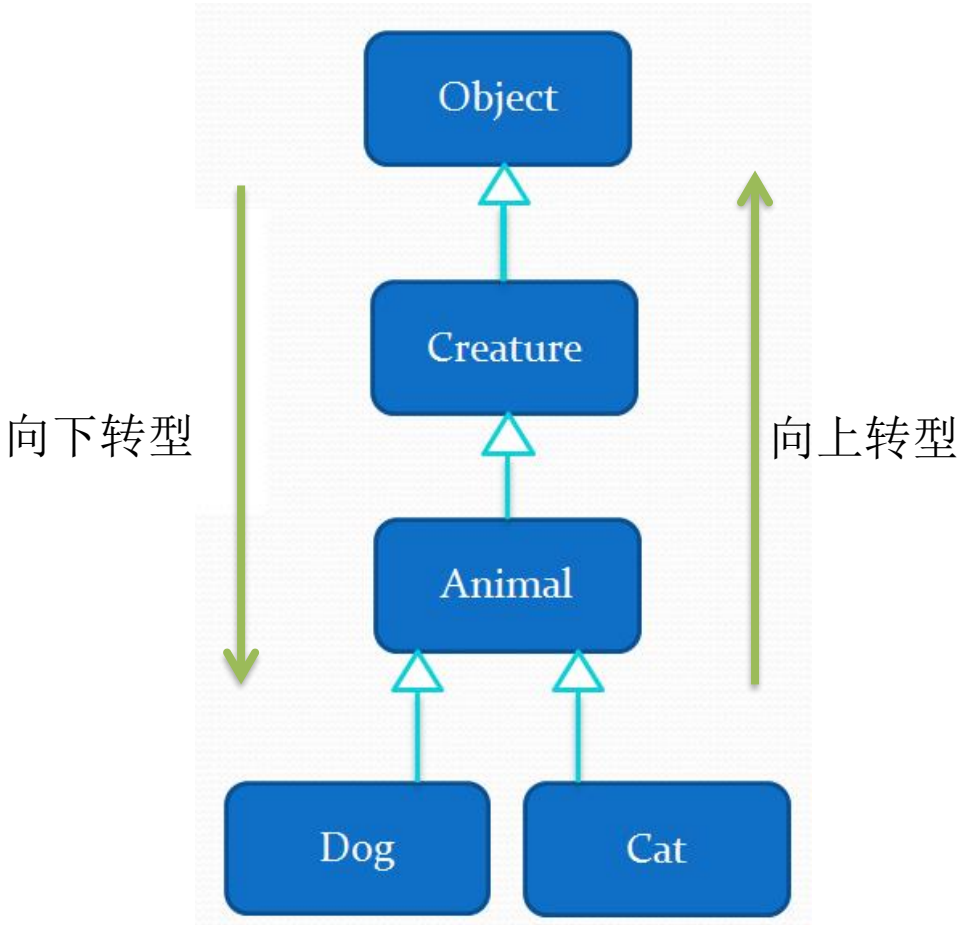
总之：多态中成员方法

- 1、编译时期：参考引用变量所属的类，如果类中没有调用的方法，编译失败
  - 例如，如果把f.show_1()前面的注释打开，则编译失败。
- 2、运行时期：参考引用变量所指的对象所属的类，运行对象所属类中的成员方法
  - 例如把子类重写的show()方法注释掉，那么打印的结果是Fu show num。
- 简而言之：***编译看左边，运行看右边。***

总之

- 上转型对象不能操作子类新增的成员变量；
- **上转型对象不能调用子类新增的成员方法。**

Upcasting和downcasting



父类和子类的对象之间转换

Java允许在父类和子类的对象之间进行转换：

1. 自动转换（向上映射）

2. 强制类型转换（向下映射）

向上转型 upcasting

- 向上转型：当有子类对象赋值给一个父类引用时，便是向上转型，多态本身就是向上转型的过程。
- 使用格式：
  - **父类类型 变量名 = new 子类类型();**
  - 如：Person p = new Student();

测试类：向上映射upcasting

```
public class animalTest {
    public static void main(String[] args) {
        Animal aMouse = new Mouse();
        Animal aGiraffa = new Giraffe();
        Animal aLion = new Lion();
        aMouse.eat();
        aMouse.sleep();
        aLion.eat();
        aLion.sleep();
        aGiraffa.eat();
        aGiraffa.sleep();
        System.out.println("--------------------------");
        Animal[] aArray = new Animal[3];
        aArray[0] = aMouse;
        aArray[1] = aGiraffa;
        aArray[2] = aLion;
        for (Animal i : aArray) {
            i.eat();
            i.sleep();}}
}
```

上转型对象的使用

一．上转型对象可以访问子类继承或隐藏的成员变量，也可以调用子类继承的方法或子类重写的实例方法。

二．如果子类重写了父类的某个实例方法后，当用上转型对象调用这个实例方法时一定是调用了子类重写的实例方法。

三．上转型对象不能操作子类新增的成员变量；不能调用子类新增的方法。

向下转型 downcasting

- 向下转型(映射)：一个已经向上转型的子类对象可以使用强制类型转换的格式，将父类引用转为子类引用，这个过程是向下转型。
- 使用格式：
- **子类类型 变量名 = (子类类型) 父类类型的变量;**
  - 如：Person p = new Student();
    Student stu = **(Student)** p
- ***如果是直接创建父类对象，是无法向下转型的！，会产生运行时异常***
  - 如：Person p = new Peron();
    Student stu = **(Student)** p

对象的向上映射和向下映射

- 对象的向上映射总是安全的，可靠的。
- 对象的向下映射就不一定了，有时可以，有时不可以，*如果不可以转的话，程序是不会报语法错误的，发生的是一个运行时异常。*
- *怎么解决这样的问题，让我们避免这个运行时异常呢？*

instanceof操作符

- instanceof操作符用于判断一个引用类型所引用的对象是否是一个类的实例。instanceof运算符是Java独有的双目运算符
- instanceof操作符左边的操作元**是一个引用类型的对象（可以是null）**，右边的操作元是一个类名或接口名。
- 形式如下：

  obj instanceof ClassName

  或者

  obj instanceof InterfaceName

instanceof操作符

Fish fish=new Fish();

//XXX表示一个类名或接口名

System.out.println(fish instanceof XXX);

- 当“XXX”是以下值时，instanceof表达式的值为**true**：
  - Fish类。
  - Fish类的直接或间接父类。
  - Fish类实现的接口。

instanceof操作符

```
Fish fish=new Fish();
System.out.println(fish instanceof  Fish);  //打印true
System.out.println(fish instanceof  Animal);  //打印true
System.out.println(fish instanceof  Object);  //打印true
System.out.println(fish instanceof  Food);  //打印true
```

```java
class ASuper {
    String s = "class:A";
}

class BSub extends ASuper {   //继承关系
    String s = "class:B"; //变量隐藏
}
```

```java
public class TypeV {
    public static void main(String args[]) {
        BSub b1,b3;
        ASuper a1, a2, a3;
        BSub b2 = new BSub();////
        a1 = b2;  //向上映射，自动转换
        a2 = b2;  //向上映射，自动转换
        System.out.println(a1.s);////
        System.out.println(a2.s);////
        b1 = (BSub) a1;  //向下映射，强制转换
        System.out.println(b1.s);
        a3=new ASuper();//////
//      b3=(BSub)a3;
        if(a3 instanceof BSub)//instanceof的用法
            b3=(BSub)a3;
        else
            System.out.println("can not be transfered!");
    }
}
```

```
class:A
class:A
class:B
can not be transfered!
```

示例代码

```
import java.util.ArrayList;
import java.util.Vector;

public class Main {

public static void main(String[] args) {
    Object testObject = new ArrayList();
        displayObjectClass(testObject);
    }
    public static void displayObjectClass(Object o) {
        if (o instanceof Vector)
        System.out.println("对象是 java.util.Vector 类的实例");
        else if (o instanceof ArrayList)
        System.out.println("对象是 java.util.ArrayList 类的实例");
        else
        System.out.println("对象是 " + o.getClass() + " 类的实例");
    }
}
```

对象是 java.util.ArrayList 类的实例

```
//Teacher和Student继承Person
//Object>String
//Object>Person>Teacher
//Object>Person>Studnet
Object object = new Student();
        System.out.println(object instanceof Student);//true
        System.out.println(object instanceof Person);//true
        System.out.println(object instanceof Object);//true
        System.out.println(object instanceof Teacher);//false
        System.out.println(object instanceof String);//false
        System.out.println("========================");
        Person person = new Student();
        System.out.println(person instanceof Student);//true
        System.out.println(person instanceof Person);//true
        System.out.println(person instanceof Object);//true
        System.out.println(person instanceof Teacher);//false
//          System.out.println(person instanceof String);//编译错误
        System.out.println("========================");
        Student student = new Student();
        System.out.println(student instanceof Student);//true
        System.out.println(student instanceof Person);//true
        System.out.println(student instanceof Object);//true
//          System.out.println(student instanceof Teacher);//编译错误
//          System.out.println(student instanceof String);//编译错误
```

小结

- instanceof是Java中的二元运算符
- 表达式 obj instanceof T，instanceof 运算符的 obj 操作数的类型必须是引用类型或空类型; 否则，会发生编译时错误。
- 如果 obj 强制转换为 T 时发生编译错误，则关系表达式的 instanceof 同样会产生编译时错误。
- 如果 obj 不为 null 并且 (T) obj 不抛 ClassCastException 异常则该表达式值为 true ，否则值为 false 。

示例代码：通过**向下转型**，使用子类特有功能。

```
//描述动物类，并抽取共性eat方法
abstract class Animal {
    abstract void eat();
}

// 描述狗类，继承动物类，重写eat方法，增加lookHome方法
class Dog extends Animal {
    void eat() {
        System.out.println("啃骨头");
    }

    void lookHome() {
        System.out.println("看家");
    }
}
```

示例代码

```
// 描述猫类，继承动物类，重写eat方法，增加catchMouse方法
class Cat extends Animal {
    void eat() {
        System.out.println("吃鱼");
    }

    void catchMouse() {
        System.out.println("抓老鼠");
    }
}
```

示例代码： instanceof操作符

```
public class Test {
    public static void main(String[] args) {
        Animal a = new Dog(); // 多态形式，创建一个狗对象
        a.eat(); // 调用对象中的方法，会执行狗类中的eat方法
        // a.lookHome();//使用Dog类特有的方法，需要向下转型，不能直接使用
        // 为了使用狗类的lookHome方法，需要向下转型
        // 向下转型过程中，可能会发生类型转换的错误，即ClassCastException异常
        // 那么，在转之前需要做健壮性判断
        if (!(a instanceof Dog)) { // 判断当前对象是否是Dog类型
            System.out.println("类型不匹配，不能转换");
            return;
        }
        Dog d = (Dog) a; // 向下转型
        d.lookHome();// 调用狗类的lookHome方法
    }
}
```

示例代码： instanceof操作符

```java
package com.buaa.test;

public interface Electronics{

}
```

```java
package com.buaa.test;
public class Thinkpad implements Electronics {
    // Thinkpad引导方法
    public void boot() {
        System.out.println("welcome,I am Thinkpad");
    }
    // 使用Thinkpad编程
    public void program() {
        System.out.println("using Thinkpad program");
    }

}
```

示例代码： instanceof操作符

```
package com.buaa.test;

public class Mouse implements Electronics {
    // 鼠标移动
    public void move() {
        System.out.println("move the mouse");
    }
    // 鼠标点击
    public void onClick() {
        System.out.println("a click of the mouse");
    }
}
```

示例代码： instanceof操作符

```
package com.buaa.test;

public class Keyboard implements Electronics {
    // 使用键盘输入
    public void input() {
        System.out.println("using Keyboard input");
    }

}
```

示例代码： instanceof操作符

```
package com.buaa.test;
import java.util.ArrayList;
import java.util.List;
public class ShopCar {
    private List<Electronics> mlist = new ArrayList<Electronics>();
    public void add(Electronics electronics) {
        mlist.add(electronics);
    }
    public int getSize() {
        return mlist.size();
    }
    public Electronics getListItem(int position) {
        return mlist.get(position);
    }
}
```

```java
package com.buaa.test;
public class Test {
    public static void main(String[] args) {
        // 添加进购物车
        ShopCar shopcar = new ShopCar();
        shopcar.add(new Thinkpad());
        shopcar.add(new Mouse());
        shopcar.add(new Keyboard());
        // 获取大小
        System.out.println("购物车存放的电子产品数量为 —> " + shopcar.getSize());
        for(int i=0;i<shopcar.getSize();i++){
            if(shopcar.getListItem(i) instanceof Thinkpad){
                Thinkpad thinkpad = (Thinkpad) shopcar.getListItem(i);
                thinkpad.boot();
                thinkpad.program();
                System.out.println("--------------------");
            }
            else if (shopcar.getListItem(i) instanceof Mouse){
                Mouse mouse = (Mouse) shopcar.getListItem(i);
                mouse.move();
                mouse.onClick();
                System.out.println("--------------------");
            }
            else if (shopcar.getListItem(i) instanceof Keyboard){
                Keyboard keyboard = (Keyboard) shopcar.getListItem(i);
                keyboard.input();
                System.out.println("--------------------");
            }}}}
```

示例代码： instanceof操作符

```
购物车存放的电子产品数量为 —> 3
welcome,I am Thinkpad
using Thinkpad program
----------------------
move the mouse
a click of the mouse
----------------------
using Keyboard input
----------------------
```

小结

- 1、什么时候使用向上转型：
  - 如果不需要使用子类特有功能时，使用向上转型，采用动态联编，可以给开发人员带来实际的价值。
- 2、什么时候使用向下转型
  - 当要使用子类特有功能时，就需要使用向下转型，调用子类特有的功能。

小结

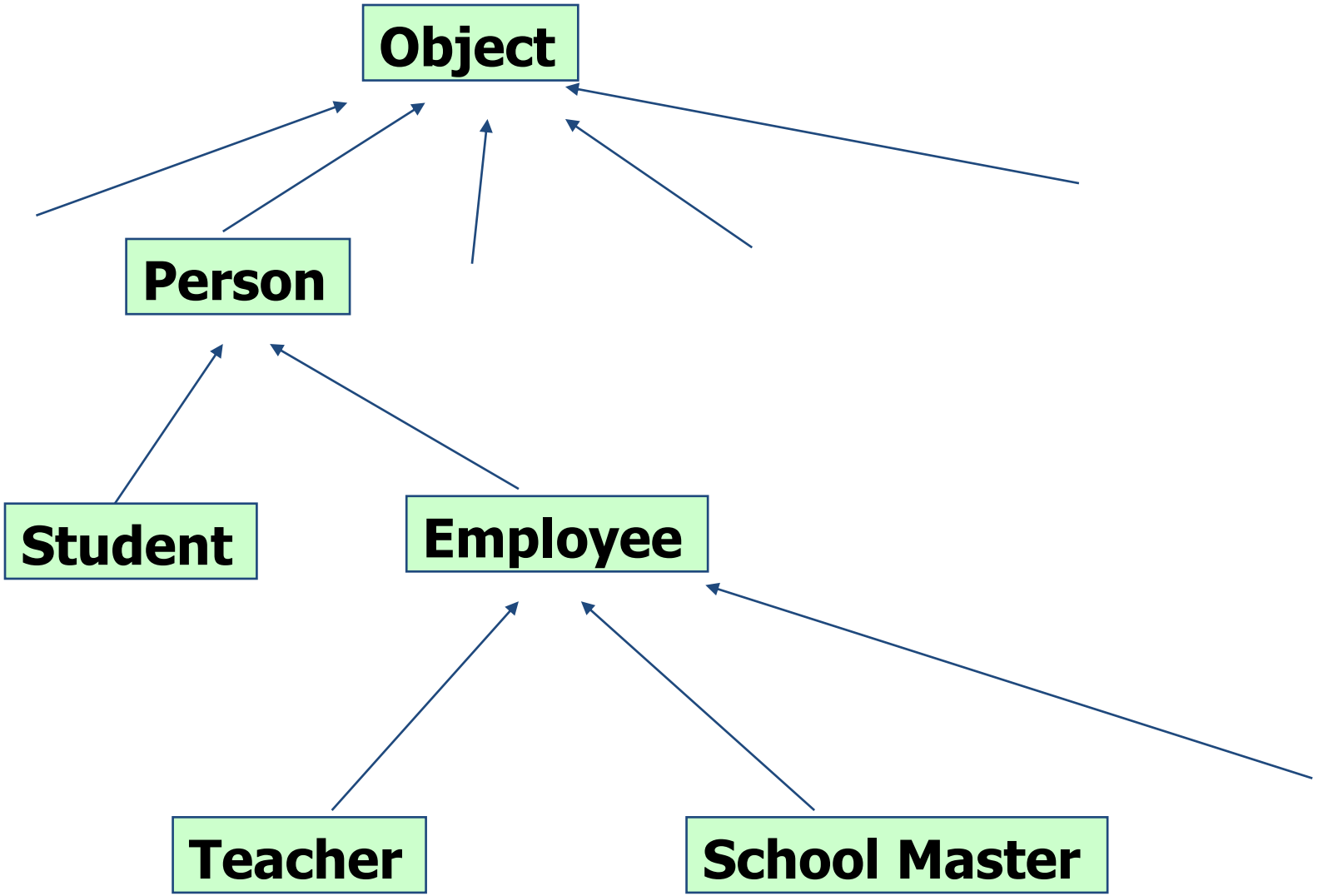
- 3、**向下转型的好处：**可以使用子类特有功能。
- 4、**向下转型的弊端：**向下转型时容易发生ClassCastException类型转换异常。在转换之前必须做类型判断。
  如：`if( !a instanceof Dog){…}`

主要内容

- 对象向上映射和向下映射（Upcasting和downcasting）
- **Object类**
- **操作符“==”与对象的equals()方法**
- 最终方法、最终类
- 内部类
- 匿名类
- 复杂软件的管理（package和import的使用）
- 高内聚、低耦合

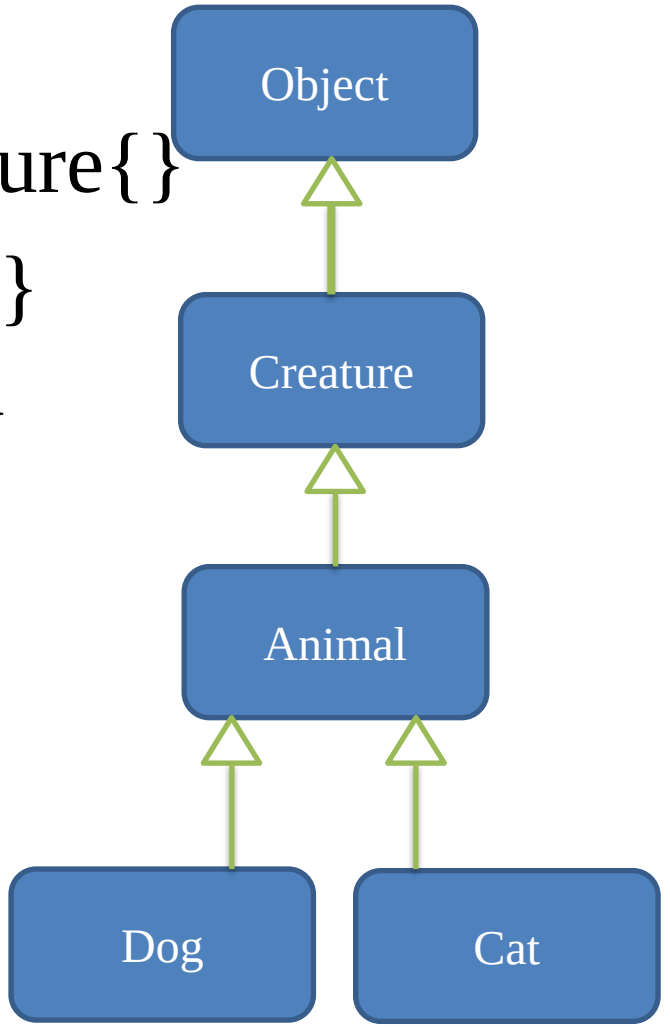
Java类的祖先类:Object类

最根类：Object类



继承树

```
class Creature{}
class Animal extends Creature{}
class Dog extends Animal{}
class Cat extends Animal{}
```



Java类的祖先类:Object类

- 所有的Java类都直接或间接地继承了java.lang.Object类，Object类是所有Java类的祖先，在这个类中定义了所有的Java对象都具有的方法。
  - equals（Object obj）：比较两个对象是否相等。仅当被比较的两个引用变量指向同一对象时，equals()方法返回true。
  - toString()：返回当前对象的字符串表示。

equals()和== 的比较

- The equals method for class Object implements the most discriminating possble equivalence relation on objects; **that is, for any reference values x and y,this method returns true if and only if x and y refer to the same object(x==y) has the value true).**
  - **判断是否为同一引用**
- ObjectTest.java

```java
package com.buaa.edu.cn;
class A {
    public String toString() {
        return "A is for ObjectTest class";
    }
}
class B {
}
public class ObjectTest {
    public static void main(String[] s) {
        A a1 = new A();
        A a2 = new A();
        A a3 = a1;
        B b1 = new B();
        System.out.println("a1.equals(a1) is " + a1.equals(a1));
        System.out.println("a1.equals(a2) is " + a1.equals(a2));
        System.out.println("a1.equals(a3) is " + a1.equals(a3));
        System.out.println(a1.toString());
        System.out.println("a1 is a instance of class "
                + a1.getClass().getName());
        // /默认的toString()输出包名加类名和堆上的首地址
        System.out.println(b1.toString());
        System.out.println("b1 is a instance of class "
                + b1.getClass().getName());
    }
}
```

```
a1.equals(a1) is true
a1.equals(a2) is false
a1.equals(a3) is true
A is for ObjectTest class
a1 is a instance of class com.buaa.edu.cn.A
com.buaa.edu.cn.B@15db9742
b1 is a instance of class com.buaa.edu.cn.B
```

默认的toString()输出包名加类名和堆上的首地址

a1.getClass().getName()输出包名加类名

操作符“==”与对象的equals()方法

- equals()方法是在Object类中定义的方法，它的声明格式如下：

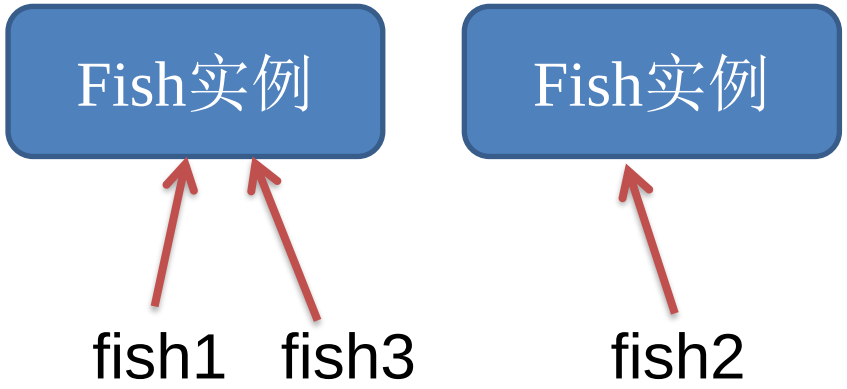
  public boolean equals(Object obj)

- Object类的equals()方法的比较规则为：当参数obj引用的对象与当前对象为同一个对象，就返回true，否则返回false：

```
public boolean equals(Object obj){
  if(this==obj)
        return true;
  else
        return false;
}
```

操作符“==”与对象的equals()方法

- 当操作符“==”两边都是引用类型变量时，这两个引用变量必须都引用同一个对象，结果才为true。




```
Food fish1=new Fish();
Food fish2=new Fish();
Food fish3=fish1;
System.out.println(fish1 ==fish2);  //打印false
System.out.println(fish1==fish3);  //打印true
```

小结：

- instanceof：判断是否为某类的一个实例
- Object类中：
  - 判断是否为同一引用可以使用：操作符“==”与对象的equals()方法，他们是等价的。
- 某些类例外，操作符“==”与对象的equals()方法不等价

操作符“==”与对象的equals()方法

- 在JDK中有一些类覆盖了Object类的equals()方法，它们的比较规则为：**如果两个对象的类型一致，并且内容一致，则返回true。判断的是长的像不像**
- 这些类包括：
  - **java.io.File、**
  - **java.util.Date、**
  - **java.lang.String、**
  - **包装类（如java.lang.Integer和java.lang.Double类等）。**

操作符“==”与对象的equals()方法

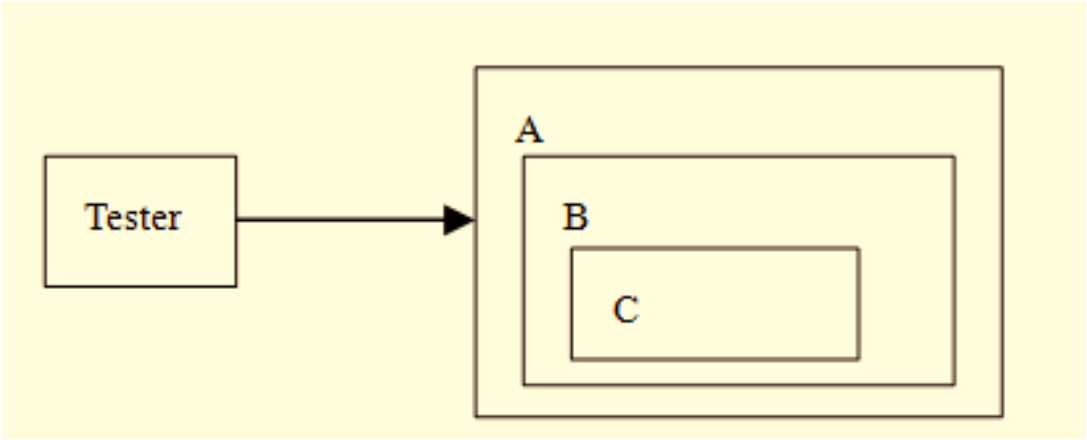
```
String str1=new String("Hello");
String str2=new String("Hello");

System.out.println(str1==str2);  //打印false
System.out.println(str1.equals(str2));  //打印true
```

主要内容

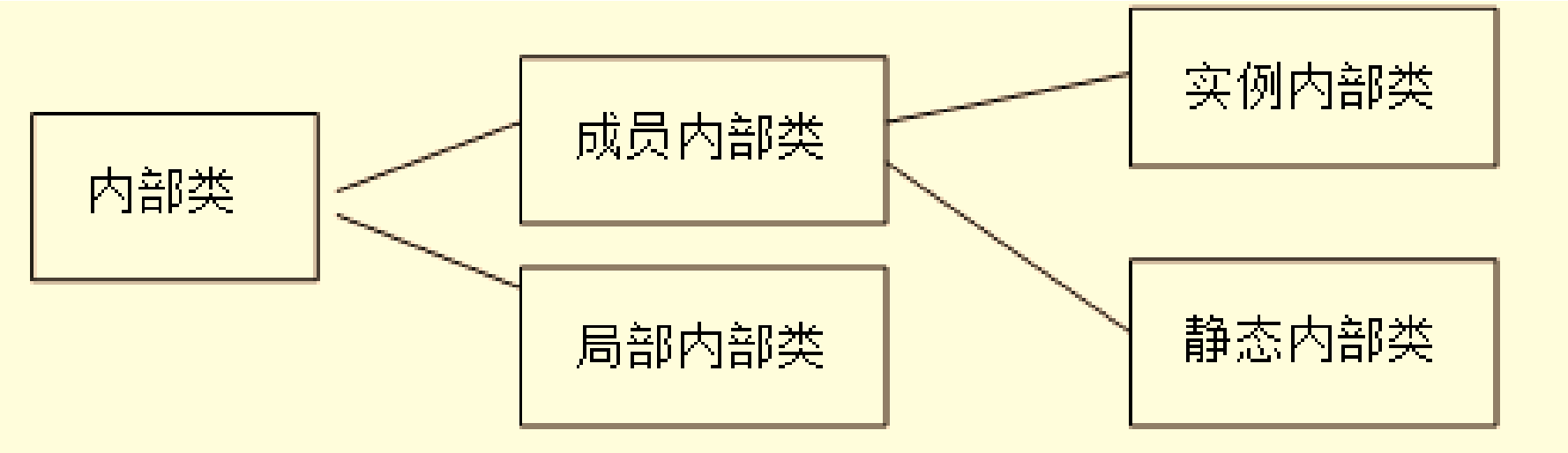
- 对象向上映射和向下映射（Upcasting和downcasting）
- 操作符“==”与对象的equals()方法
- **内部类**
- **匿名类**
- 最终方法、最终类
- 复杂软件的管理（package和import的使用）
- 高内聚、低耦合

主要内容

- 内部类的基本语法
  - 实例内部类
  - 静态内部类
  - 局部内部类
- 匿名类
- 内部类的用途



内部类的分类



示例代码

```
class Outer {
    public class InnerTool { // 内部类
        public int add(int a, int b) {
            return a + b;
        }
    }
    private InnerTool tool = new InnerTool();

    public void add(int a, int b, int c) {
        tool.add(tool.add(a, b), c);
        System.out.println(tool.add(tool.add(a, b), c));
    }
}
```

示例代码

```
public class Tester {
    public static void main(String args[]) {
        Outer o = new Outer();
        o.add(1, 2, 3);
        Outer.InnerTool tool = new Outer().new InnerTool();
    }
}
```

在创建实例内部类的实例时，外部类的实例必须已经存在

创建实例内部类的实例

- 在创建实例内部类的实例时，外部类的实例必须已经存在
  - 例如：要创建InnerTool类的实例，必须先创建Outer外部类的实例
  - `Outer.InnerTool tool=new Outer().new InnerTool();`
  - 以上代码等价于：
  - `Outer outer=new Outer();`
  - `Outer.InnerTool tool =outer.new InnerTool() ;`

创建实例内部类的实例

- 以下代码会导致编译错误：
- `Outer.InnerTool tool=new Outer.InnerTool();`

实例内部类访问外部类的成员

- 在内部类中，可以直接访问外部类的所有成员，包括成员变量和成员方法。

- 实例内部类的实例自动持有外部类的实例的引用。

示例代码

内部类中，可以直接访问外部类的所有成员，包括成员变量和成员方法

```
public class A {
    private int a1;
    public int a2;
    static int a3;
    public A(int a1, int a2) {
        this.a1 = a1;
        this.a2 = a2;
    }
    protected int methodA() {
        return a1 * a2;
    }
    class B{  //内部类
        int b1=a1;  //直接访问private的a1
        int b2=a2;  //直接访问public的a2
        int b3=a3;  //直接访问static的a3
        int b4=new A(3,4).a1;  //访问一个新建的实例A的a1
        int b5=methodA();  //访问methodA()方法
    }
```

示例代码

```
    public static void main(String args[]){
      A.B b=new A(1,2).new B();
      System.out.println("b.b1="+b.b1);   //打印b.b1=1
      System.out.println("b.b2="+b.b2);   //打印b.b2=2
      System.out.println("b.b3="+b.b3);   //打印b.b3=0
      System.out.println("b.b4="+b.b4);   //打印b.b4=3
      System.out.println("b.b5="+b.b5);   //打印b.b5=2
    }
}
```

静态内部类

- 静态内部类的实例不会自动持有外部类的特定实例的引用，在创建内部类的实例时，不必创建外部类的实例。

示例代码

- 例如以下类A有一个静态内部类B，客户类Tester创建类B的实例时不必创建类A的实例：

```
class AA {
    public static class B {
        int v;
    }
}

class Tester {
    public void test() {
        AA.B b = new AA.B();
        b.v = 1;
    }
}
```

静态内部类

- 客户类可以通过完整的类名直接访问静态内部类的静态成员。

```
public class AAA {
    public static class B{
        int v1;
        static int v2;
        public static class C{
            static int v3;
            int v4;
        }
    }
}
```

示例代码

```
public class TesterAAA {
    public void test() {
        AAA.B b = new AAA.B();
        AAA.B.C c = new AAA.B.C();
        b.v1 = 1;
        b.v2 = 1;
        // AAA.B.v1=1; //编译错误
        AAA.B.v2 = 1; // 合法
        AAA.B.C.v3 = 1; // 合法
    }
}
```

局部内部类

- 局部内部类只能在当前方法中使用。
- 局部内部类和实例内部类一样，可以访问外部类的所有成员
- 此外，局部内部类还可以访问所在方法中的符合以下条件之一的参数和变量：
  - 最终变量或参数：用final修饰

局部内部类示例代码

```
public class AAAA {
    int a;
    public void method(final int p1, final int p2) {
        final int localV1 = 1;
        final int localV2 = 2;
        int localV3 = 0;
        localV3 = 1; // 修改局部变量
        class B {
            int b1 = a; // 合法，访问外部类的实例变量
            int b2 = p1; // 合法，访问final类型的参数
            int b3 = p2; // 合法，访问final类型的参数
            int b4 = localV1; // 合法,访问实际上的最终变量
            int b5 = localV2; // 合法，访问最终变量
            // int b6=localV3; //编译错误，localV3不是最终变量或者实际上的最终变量
        }
    }
}
```

匿名类

- 匿名类就是没有名字的类，是将类和类的方法定义在一个表达式范围里。
- 匿名类本身没有构造方法，但是会调用父类的构造方法。
- 匿名内部类将内部类的定义与生成实例的语句合在一起，并省去了类名以及关键字“class”,”extends”和“implements”等。

示例代码

```
public class AAAAA {
    AAAAA(int v) {
        System.out.println("another constructor");
    }
    AAAAA() {
        System.out.println("default constructor");
    }
    void method() {
        System.out.println("from AAAAA");
    };
    public static void main(String args[]) {
        new AAAAA().method(); // 打印from AAAAA
        AAAAA a = new AAAAA() { // 匿名类
            void method() {
                System.out.println("from anonymous");
            }
        };
        a.method(); // 打印from anonymous
    }
}
```

输出结果

```
default constructor
from AAAAA
default constructor
from anonymous
```

匿名类没有构造方法

```
    public static void main(String args[]) {
        int v = 1;
        AAAAA a = new AAAAA(v) { // 匿名类
            void method() {
                System.out.println("from anonymous");
            }
        };
        a.method(); // 打印from anonymous
    }

}
```

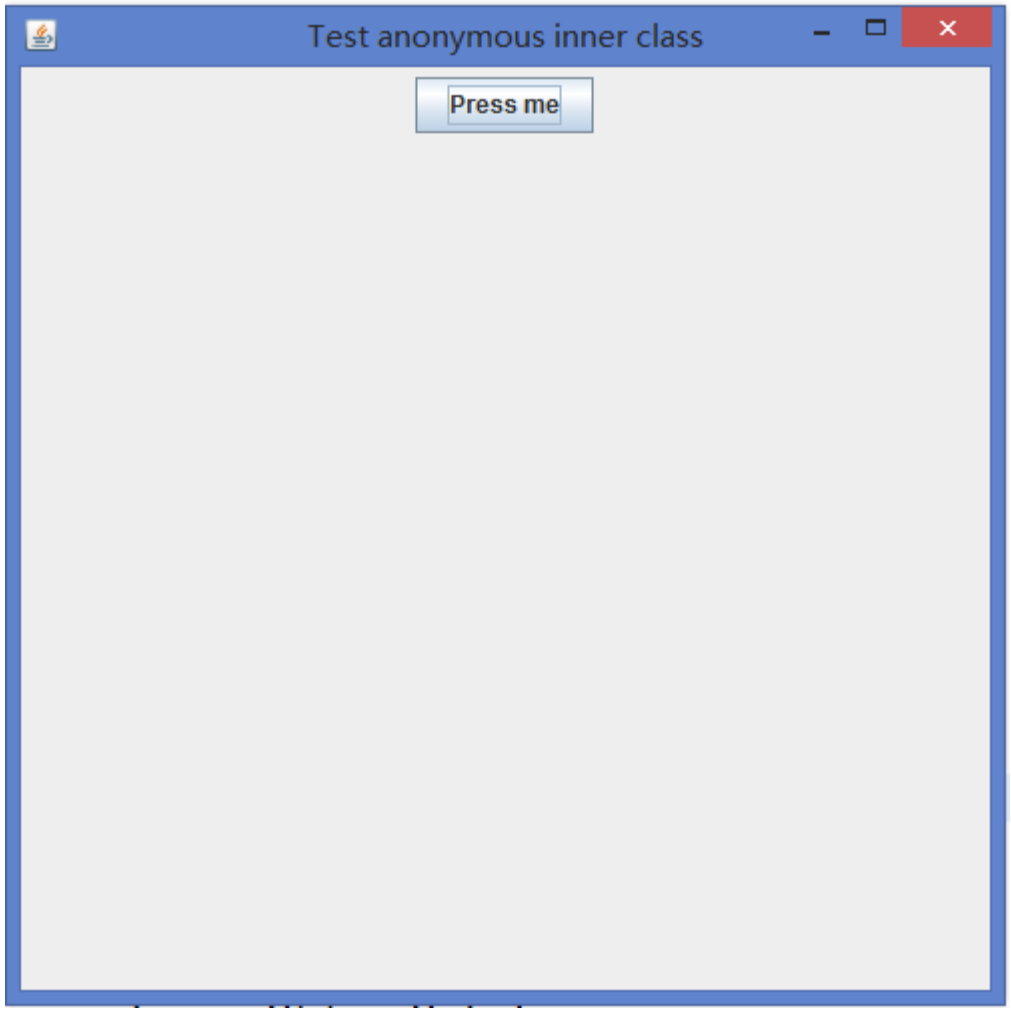
```
another constructor
from anonymous
```

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class J_Test1 extends JFrame{
   public J_Test1(){
      super("Test anonymous inner class");
      Container container=getContentPane();
      container.setLayout(new FlowLayout(FlowLayout.CENTER));
      JButton b=new JButton("Press me");
      container.add(b);
      b.addActionListener(
        new ActionListener(){
            public void actionPerformed(ActionEvent e){
              System.out.println("The button is pressed");
            }
      });     setSize(100,80);   setVisible(true);
   }
   public static void main(String[] args){
      J_Test1 application=new J_Test1();
      application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }}
```

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class J_Test2 extends JFrame{
   public J_Test2(){
      super("Test anonymous inner class");
      Container container=getContentPane();
      container.setLayout(new FlowLayout(FlowLayout.CENTER));
      JButton b=new JButton("Press me");
      container.add(b);
      b.addActionListener(new J_ActionListener());
      setSize(100,80);     setVisible(true);
   }
   public static void main(String[] args){
      J_Test1 application=new J_Test1();
      application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }}
class J_ActionListener implements ActionListener{
      public void actionPerformed(ActionEvent e){
         System.out.println("The button is pressed");
      }}
```

J_Test2.java

输出结果



内部类的用途（价值何在？）

- 封装类型
  - 如果一个类只能有系统中的某一个类访问，可以定义为该类的内部类。
- 直接访问外部类的成员
- 回调外部类的方法

内部类封装类型

- 顶层类只能处于public和默认访问级别
- 而成员内部类可以处于**public、protected、默认和private**四个访问级别。
- 此外，如果一个内部类仅仅为特定的方法提供服务，那么可以把这个内部类定义在方法之内。
- 可见，**内部类是一种封装类型的有效手段。**

内部类封装类型

```
public interface Tool {
    public int add(int a, int b);
}
```

```
public class Outer {
    private class InnerTool implements Tool {
        public int add(int a, int b) {
            return a + b;
        }
    }
    public Tool getTool() {
        return new InnerTool();
    }
    public static void main(String[] args){
        //InnerTool实例向上转型为Tool类型
        Tool tool=new Outer().getTool();
        Tool tool1=new Outer().new InnerTool();
    }
}
```

内部类封装类型（续）

- 在客户类中不能访问Outer.InnerTool类，但是可以通过Outer类的getTool()方法获得InnerTool的实例

内部类访问外部类的成员

- 内部类的一个特点是能够访问外部类的各种访问级别的成员。
- 假定有类A和类B，类B的reset()方法负责重新设置类A的实例变量count的值。
  - 一种实现方式是把类A和类B都定义为外部类：

示例代码

```
class A{
  private int count;
  public int add(){return ++count;}

  public int getCount(){return count;}

  public void setCount(int count){
        this.count=count;}
}
```

示例代码

```
class B{
  A a;   //类B与类A关联
  B(A a){this.a=a;}
  public void reset(){
    if(a.getCount()>0)
      a.setCount(1);
    else
     a.setCount(-1);
  }
}
```

内部类访问外部类的成员

- 假如需求中要求类A的count属性不允许被除类B以外的其他类读取或设置，那么以上实现方式就不能满足这一需求。
- 在这种情况下，把类B定义为内部类就可以解决这一问题，而且会使程序代码更加简洁

示例代码

```
class A{
  private int count;
  public int add(){return ++count;}

  class B{ //定义内部类B
    public void reset(){
      if(count>0)
        count=1;
      else
        count=-1;
    }
  }
}
```

回调

- 在以下Adjustable接口和Base类中都定义了adjust()方法，这两个方法的参数签名相同，但是有着不同的功能。

```
public interface Adjustable{
  /** 调节温度 */
  public void adjust(int temperature);
}
public class Base{
  private int speed;
  /** 调节速度 */
  public void adjust(int speed){
    this.speed=speed;
  }
}
```

回调

- 如果有一个Sub类同时具有调节温度和调节速度的功能，那么Sub类需要继承Base类，并且实现Adjustable接口，但是以下代码并不能满足这一需求：

adjust(int speed)
调节速度

```
public class Sub extends Base implements Adjustable{
  private int temperature;
  public void adjust(int temperature){
    this.temperature=temperature;
  }
}
```

adjust(int temerature)
调节温度

回调

```
public class Sub extends Base {
  private int temperature;
  private void adjustTemperature(int temperature){
    this.temperature=temperature;
  }
  private class Closure implements Adjustable{
    public void adjust(int temperature){
      adjustTemperature(temperature);
    }
  }
  public Adjustable getCallBackReference(){
     return new Closure();
  }
}
```

回调

- 以下代码演示客户类使用Sub类的调节温度的功能：

```
public class TestClass {
    public static void main(String[] args){
        //调节温度
        Sub sub=new Sub();
        Adjustable ad=sub.getCallBackReference();
        ad.adjust(15);

        //调节速度
        sub.adjust(350);
    }
}
```

接口回调

回调

- 回调实质上是指一个类尽管实际上实现了某种功能，但是没有直接提供相应的接口，客户类可以通过这个类的内部类的接口来获得这种功能。而这个内部类本身并没有提供真正的实现，仅仅调用外部类的实现。
- 可见，回调充分发挥了内部类具有访问外部类的实现细节的优势。

```
public class Sub extends Base {
  private int temperature;

  private void adjustTemperature(int temperature){
    this.temperature=temperature;
  }

  private class Closure implements Adjustable{
    public void adjust(int temperature){
      adjustTemperature(temperature);
    }
  }
  public Adjustable getCallBackReference(){
     return new Closure();
  }
}
```

内部类的文件

- 对于每个内部类，Java编译器会生成独立的.class文件。这些类文件的命名规则如下：
  - 成员内部类：外部类的名字$内部类的名字
  - 局部内部类：外部类的名字$数字和内部类的名字
  - 匿名类：外部类的名字$数字

```java
public class AAAAAA {
    static class B {
    } // 成员内部类，对应A$B.class
    class C { // 成员内部类，对应A$C.class
        class D {
        } // 成员内部类，对应A$C$D.class
    }
    public void method1() {
        class E {
        } // 局部内部类1，对应A$1E.class

        B b = new B() {
        }; // 匿名类1，对应A$1.class
        C c = new C() {
        }; // 匿名类2，对应A$2.class
    }
    public void method2() {
        class E {
        } // 局部内部类2，对应A$2E.class
    }
}
```

Java编译器编译以上程序，会生成以下类文件：

A.class

A$B.class

A$C.class

ACD.class

A$1E.class

A$1.class

A$2.class

A$2E.class

小结（思考题，下面的描述是否正确？）

比较方面	实例内部类	静态内部类	局部内部类
主要特征	内部类的实例引用特定的外部类的实例	内部类的实例不与外部类的任何实例关联	可见范围是所在的方法
可用的修饰符	访问控制修饰符，abstract,final	访问控制修饰符，static,abstract,final	abstract,final
可以访问外部类的哪些成员	可以直接访问外部类的所有成员	只能直接访问外部类的静态成员	可以直接访问外部类的所有成员，并且能访问所在方法的最终或实际上的最终变量和参数
拥有成员的类型	只能拥有实例成员	可以拥有静态成员和实例成员	只能拥有实例成员
外部类如何访问内部类的成员	必须通过内部类的实例来访问	对于静态成员，可以通过内部类的完整类名来访问	必须通过内部类的实例来访问

主要内容

- 对象向上映射和向下映射
- Object类
- 操作符“==”与对象的equals()方法
- 内部类
- 匿名类
- **最终方法、最终类**
- 复杂软件的管理（package和import的使用）
- 高内聚、低耦合

讨论：（overview）

- 1、继承带来了哪些好处？
  2、继承是否破坏了类的封装性？

面向对象的三大特征之一：*继承真的很好*（overview）

一. 继承避免了公用代码的重复开发，减少代码的冗余，提高程序的复用性；

二. 支持多态（通过向上映射），提高程序的可扩展性；

三. 继承是类实现可重用性和可扩充性的关键持征。在继承关系下类之间组成网状的层次结构。

四. 通过继承增强一致性，从而减少模块间的接口和界面。

五. 通过向上映射，让我们体验到多态的益处。

继承是否破坏了类的封装性？ （overview）

- 是的。继承破坏了封装性，换句话说，子类依赖于父类的实现细节。**继承很容易改变父类实现的细节(所以父类中能写成final尽量写成final)，**即使父类整体没有问题，也有可能因为子类细节实现不当，而破坏父类的约束。
- 其实这是一个平衡关系，不是绝对关系，一定程度的封装和一定程度的继承，可以提高开发效率，继承破坏了封装，但是有时继承是必须的，为了继承牺牲一定的封装是允许的。不能绝对的为了封装，就不去继承。

谨慎继承 （overview）

- 继承是一种强耦合关系。父类变，子类就必须变。
- 继承破坏了封装，子类可以重写父类的方法，子类可以直接访问保护成员。
- 那么到底要不要使用继承呢？
  - 《Think in java》中提供了解决办法：问一问自己是否需要从子类向父类***进行向上转型***。如果必须向上转型，则继承是必要的，但是如果不需要，则应当好好考虑自己是否需要继承。

继承的使用原则 （overview）

- 合理使用继承，谨慎继承，**继承树的层次不可太多**
- 继承是一种提高程序代码的可重用性、以及提高系统的可扩展性的有效手段
- 但是，如果继承树非常复杂、或者随便扩展本来不是专门为继承而设计的类，反而会削弱系统的可扩展性和可维护性

最终方法，最终类的**必要性**

- 因为继承破坏了封装性，换句话说，子类依赖于父类的实现细节。**继承很容易改变父类实现的细节，**即使父类整体没有问题，也有可能因为子类细节实现不当，而破坏父类的约束。
- **所以父类中能写成final尽量写成final**

最终类

- 如果一个类没有必要再派生子类，通常可以用final关键字修饰，表明它是一个最终类。

```
1. Public final class Math{}
2. Public final class String{}
```

最终方法

- 最终方法：用关键字final修饰的方法称为最终方法。

- 最终方法既不能被覆盖，也不能被重载，它是一个最终方法，其方法的定义永远不能改变

final方法和final类

- final类中的方法可以不声明为final方法，但实际上final类中的方法都是隐式的final方法
- final修饰的方法不一定要存在于final类中。
- **定义类头时，abstract和final不能同时使用**
- 访问权限为private的方法默认为final的

常量

一. 常量：在程序运行过程中，其值不变的量。

二. Java中的常量使用关键字final修饰。

三. final既可以修饰简单数据类型，也可以修饰复合数据类型。

四. final常量可以在声明的同时赋初值，也可以在构造函数中

五. 复合数据类型常量可以是Java类库定义的复合数据类型，也可以是用户自定义的复合数据类型

常量声明格式

格式：final 数据类型 常量名=值

例如：final double PI=3.1415926;

final String str=“Hello World”;

注意事项

一. 简单数据类型常量其值一旦确定，就不能被改变。

二. 复合数据类型常量指的是引用不能被改变，而其具体的值是可以改变的。

三. 常量既可以是局部常量，也可以是类常量和实例常量。如果是类常量，在数据类型前加static修饰（由所有对象共享）。如果是实例常量，就不加static修饰。

四. 常量名一般大写，多个单词之间用下划线连接。

示例代码

```
class PersonA {
    String name;
    String sex;
    int age;
    final static double PAI = 3.1415926; // 静态常量
    final double ID; // 常量，表示每一个人的id不同，但一旦赋值又是不能变化的

    public PersonA(String n, String s, int a, int id) {
        name = n;
        sex = s;
        age = a;
        ID = id; // 构造函数中赋值
    }

    public String toString() {
        String s =
    "姓名：" + name + "," + "性别：" + sex + "," + "年龄：" + age;
        return s;
    }
}
```

示例代码

```
public class FinalTest {
    public static void main(String args[]) {
        final double PAI = 3.1415926; // 局部常量

        // final既可以修饰简单数据类型，也可以修饰符和数据类型
        final PersonA p1 = new PersonA("Tom", "M", 23, 001);
        PersonA p2 = new PersonA("Mary", "F", 20, 002);
        System.out.println("final p1:" + p1.toString());

        // p1=p2 //对final对象重新赋值会产生编译错误
        // 以下对final对象中的成员变量，重新赋指是可以的
        p1.name = p2.name;
        p1.sex = p2.sex;
        p1.age = p2.age;
        System.out.println("final p1:" + p1.toString());
    }
}
```

小结：使用最终方法，最终类增强程序的鲁棒性

- 将方法或者类声明为final型可以有效防止他人覆写该函数，或者继承于该类。**但是或许更重要的是，这么做可以“关闭”动态绑定。**

- 或者说，这么做便是告诉编译器：动态绑定是不需要的。于是编译器可以产生效率较佳的程序代码。

思考题

```
package com.buaa.edu;
public class EduBackground {

    String primarySchool;
    String secondarySchool;
    String juniorHSchool;
    String seniorHSchool;
    String university;

    public EduBackground() {

    }
}
```

```java
package com.buaa.edu;
public class Person {
    private String name;
    private int age;
    private String gender;
    private final EduBackground edu = new EduBackground();
    public Person() {
    }
    // final修饰局部变量、修饰成员方法、修饰方法的参数
    // 修饰局部变量时，局部变量的值不能改变
    public void finalLocal() {
        final int i;
        final EduBackground edu = new EduBackground();
        i = 1;
        System.out.println("finalLocal: i = " + i);
    }
    // 修饰方法的参数时(简单数据类型)，参数i不能被修改
    public void finalArgs(final int i) {
        // i = 3;
        System.out.println("finalArgs: i = " + i);
    }
    // 修饰方法的参数时(复合数据类型)，不能指向新的位置
    public void finalArgs(final EduBackground edu) {
        // edu = new EduBackground();
        System.out.println("finalArgs: edu");
    }

    // 修饰成员方法时，成员方法不能被子类重写
    public final void finalMethod() {
        int i = 2;
        System.out.println("finalMethod: i = " + i);
    }

    private final void priFinalMethod() {
        System.out.println("Person:priFinalMethod");
    }

    public static void main(String[] args) {
        Person per = new Person();
        Student stu = new Student();
        Person per1 = stu;

        per.priFinalMethod();
        stu.priFinalMethod();
        per1.priFinalMethod();
    }
}
```

```java
package com.buaa.edu;
public class Student extends Person {
    private final int stuNumber;
    private int score;
    private static final int BAN_JI=20210001;
    public Student() {
        stuNumber=(int)Math.random()*500;
        score=(int)Math.random()*100;
    }
//子类不能重写父类被final修饰的方法
//  public final void finalMethod() {
//          int i = 2;
//          System.out.println("finalMethod: i = " + i);
//  }
    public final void priFinalMethod() {
        System.out.println("Student:priFinalMethod");
    }
}
```

- 茶歇