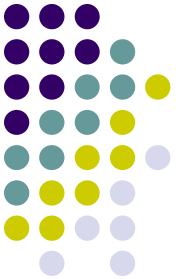
第三章 Java面向对象程序设计 – 3



3.12 接口

3.13 塑型

3.14 多态

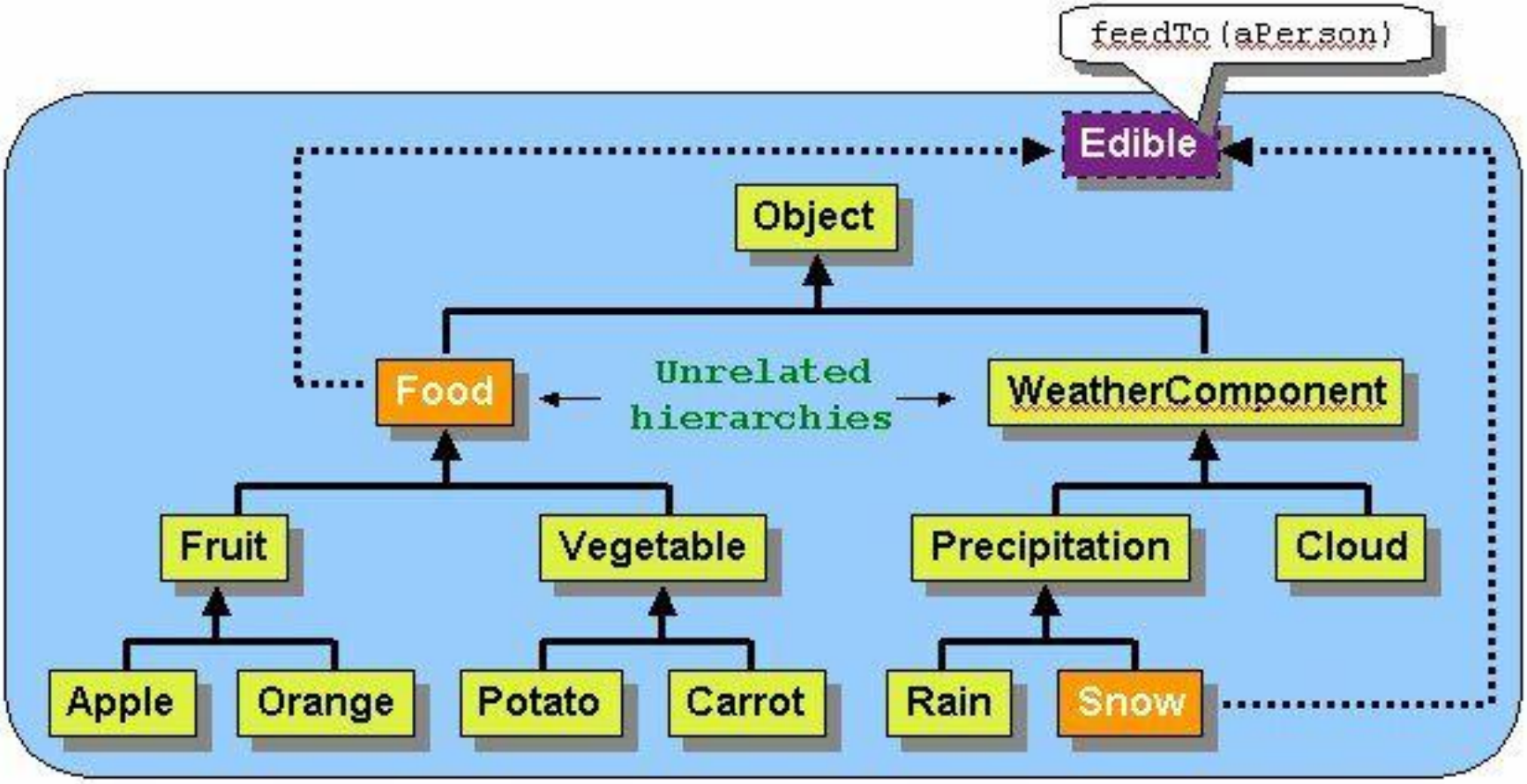
3.15 内部类

3.12 接口

- 接口(interface)
  - 与抽象类一样，定义多个类的共同属性
  - 使抽象的概念更深入了一层，是一个“纯”抽象类，它只提供一种形式，并不提供实现
  - 允许创建者规定方法的基本形式：方法名、参数列表以及返回类型，但不规定方法主体
  - 也可以包含基本数据类型的数据成员，但它们都默认为static和final

3.12 接口

- 与抽象类的差异
  - 接口允许我们在看起来不相干的对象之间定义共同行为

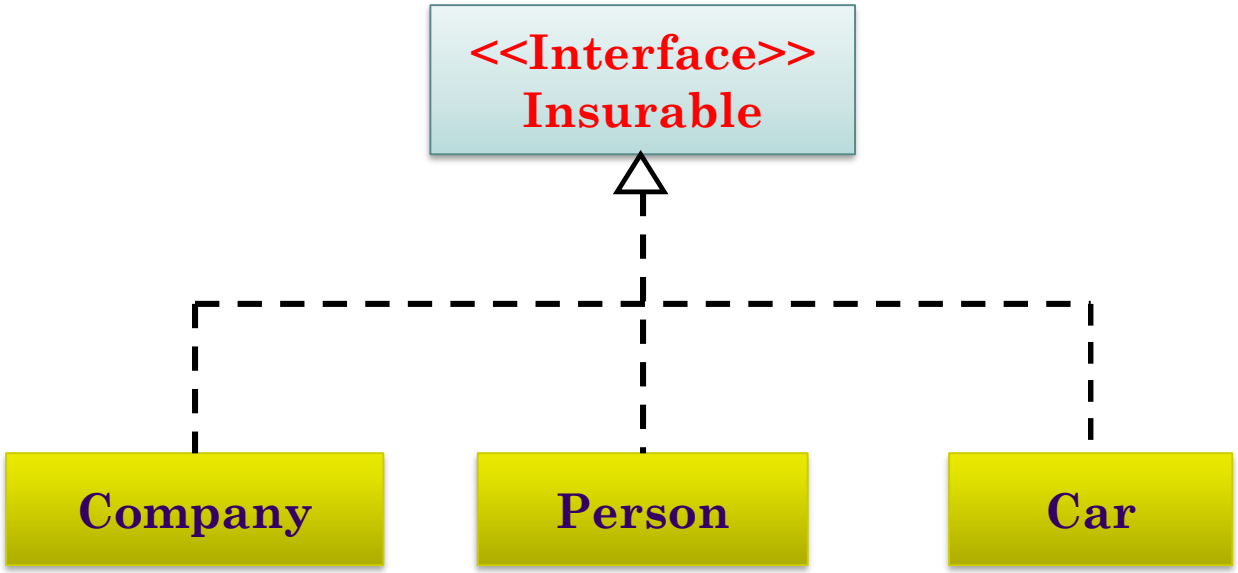


3.12 接口

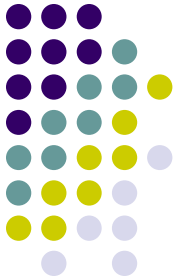
- 与抽象类的差异
  - 实现多继承，同时免除C++中的多继承那样的复杂性
  - 建立类和类之间的“协议”
    - 把类根据其实现的功能来分别代表，而不必顾虑它所在的类继承层次；
    - 这样可以最大限度地利用动态绑定，隐藏实现细节
    - 实现不同类之间的常量共享

3.12.1 接口的语法

- 举例
  - 具有车辆保险、人员保险、公司保险等多种保险业务，在对外提供服务方面具有相似性，如都需要计算保险费(premium)等，因此可声明一个Insurable 接口
  - 在UML图中，实现接口用带有空三角形的虚线表示



3.12.1 接口的语法

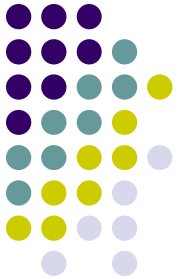


- 声明格式

```
[接口修饰符] interface 接口名称 [extends 父接口名]
{
    …//方法的原型声明或静态常量
}
```

- **接口的数据成员一定要赋初值，且此值将不能再更改，允许省略final、static关键字**
- **接口中的方法必须是“抽象方法”，不能有方法体，允许省略public及abstract关键字**

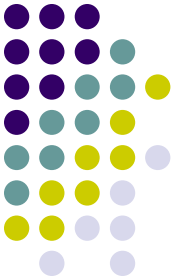
3.12.1 接口的语法



- 上例实现
  - 具有车辆保险、人员保险、公司保险等多种保险业务，在对外提供服务方面具有相似性，如都需要计算保险费(premium)等
  - Insurable 接口声明如下：

```
public interface Insurable   {
    public int getNumber(); //省略abstract关键字
    public int getCoverageAmount();
    public double calculatePremium();
    public Date getExpiryDate();
 }
```

3.12.1 接口的语法



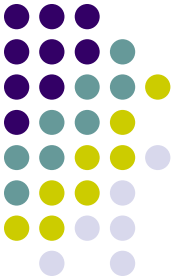
- 举例
- 声明一个接口Shape2D，可利用它来实现二维的几何形状类Circle和Rectangle
  - 把计算面积的方法声明在接口里
  - pi值是常量，把它声明在接口的数据成员里

```
interface Shape2D{                  //声明Shape2D接口
    final  double pi=3.14;          //数据成员一定要初始化
    public  abstract double area();   //抽象方法
}
```

  - 在接口的声明中，允许省略一些关键字，也可声明如下

```
interface Shape2D{          //声明Shape2D接口
    double pi=3.14;         //数据成员一定要初始化
    double area();          //抽象方法
}
```

3.12.2 实现接口

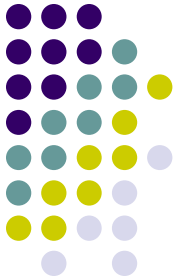


- 接口的实现
  - 接口不能用new运算符直接产生对象，必须利用其特性设计新的类，再用新类来创建对象
  - 利用接口设计类的过程，称为接口的实现，使用implements关键字
  - 语法如下

```
public class 类名称 implements 接口名称 {
        /* Bodies for the interface methods */
        /* Own data and methods. */
}
```

    - 必须实现接口中的所有方法
    - 来自接口的方法必须声明成public

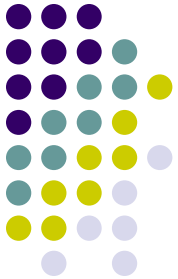
3.12.2 实现接口



- 举例
  - 声明汽车类实现例子中的Insurable接口，实现接口中的所有抽象方法

```
public class Car implements Insurable {
        public int getPolicyNumber() {
          // write code here
        }
        public double calculatePremium() {
          // write code here
         }
        public Date getExpiryDate() {
          // write code here
         }
        public int getCoverageAmount() {
          // write code here
         }
         public int getMileage() {  //新添加的方法
           //write code here
         }
  }
```

3.12.2 实现接口



- 对象转型
  - 对象可以被转型为其所属类实现的接口类型

```
Car  jetta = new Car();
Insurable  item = (Insurable)jetta;   //对象转型为接口类型
item.getPolicyNumber();
item.calculatePremium();
item.getMileage();          // 接口中没有声明此方法，不可以
jetta.getMileage();         // 类中有此方法，可以
((Car)item).getMileage(); // 转型回原类，可调用此方法了
```

  - getPolicyNumber、calculatePremium是Insurable接口中声明的方法
  - getMileage是Car类新添加的方法，Insurable接口中没有声明此方法

3.12.2 实现接口

- 声明Circle与Rectangle两个类实现Shape2D接口

```
interface Shape2D{                  //声明Shape2D接口
    final  double pi=3.14;          //数据成员一定要初始化
    public  abstract double area();   //抽象方法
}
```

```
class Circle implements Shape2D
{
    double radius;
    public Circle(double r)
    {
       radius=r;
    }
    public double area()
   {
       return (pi * radius * radius);
   }
}
```

```
class Rectangle implements Shape2D
{
     int width, height;
     public Rectangle(int w, int h)
     {
        width=w;
        height=h;
      }
      public double area()
      {
        return (width * height);
       }
}
```

3.12.2 实现接口

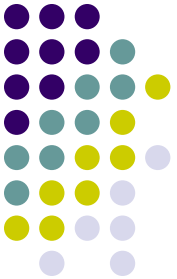
- 测试上例

```
public class InterfaceTester {
   public static void main(String  args[]){
      Rectangle rect = new Rectangle(5,6);
      System.out.println("Area of rect = " + rect.area());
      Circle cir = new Circle(2.0);
      System.out.println("Area of circle = " + cir.area());
   }
}
```

- 运行结果

Area of rect = 30.0
Area of circle = 12.56

3.12.3 接口的扩展



- 接口可通过扩展的技术派生出新的接口
  - 原来的接口称为基本接口(base interface)或父接口(super interface)
  - 派生出的接口称为派生接口(derived interface)或子接口(sub interface)
- 派生接口不仅可以保有父接口的成员，同时也可加入新成员以满足实际问题的需要
- 实现接口的类也必须实现此接口的父接口
- 接口扩展的语法

  interface 子接口的名称 extends 父接口1，父接口2，…

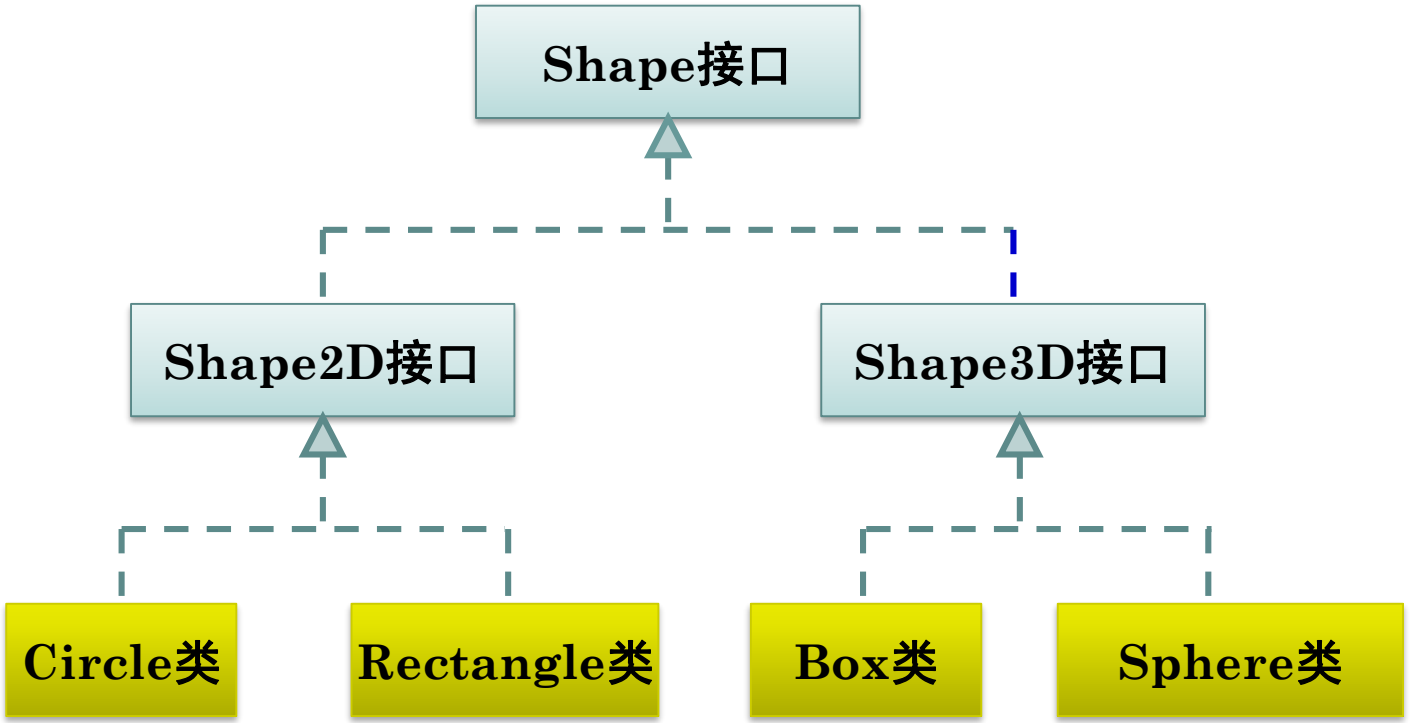
  {

  …… …

  }

3.12.3 接口的扩展

- 举例
- Shape是父接口，Shape2D与Shape3D是其子接口。Circle类及Rectangle类实现接口Shape2D，而Box类及Sphere类实现接口Shape3D



3.12.3 接口的扩展

- 部分代码如下

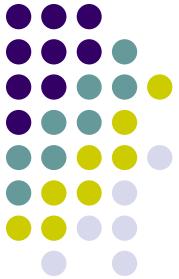
```
// 声明Shape接口
interface Shape{
    double pi=3.14;
    void setColor(String str);
}
```

```
//声明Shape2D接口扩展Shape接口
interface Shape2D extends Shape {
    double area();
}
```

```
class Circle implements Shape2D {
    double radius;
    String color;
    public Circle(double r) {   radius=r;  }
    public double area() {
      return (pi*radius*radius);
     }
    public void setColor(String str){
      color=str;
      System.out.println(“Color="+color);
    }
}
```

```
//测试类
public class ExtendsInterfaceTester{
  public static void main(String []args) {
     Circle cir;
     cir=new Circle(2.0);
     cir.setColor("blue");
     System.out.println("Area = " + cir.area());
  }
}
```

3.12.3 接口的扩展



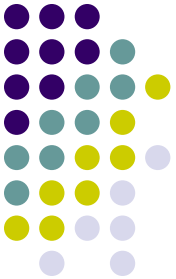
- 运行结果

  Color=blue

  Area = 12.56

- 说明
  - 首先声明了父接口Shape，然后声明其子接口Shape2D
  - 之后声明类Circle实现Shape2D子接口，因而在此类内必须明确定义setColor( )与area( )方法的处理方式
  - 最后在主类中我们声明了Circle类型的变量cir，并创建新对象，最后通过cir对象调用setColor( )与area( )方法

3.12.4 多重继承



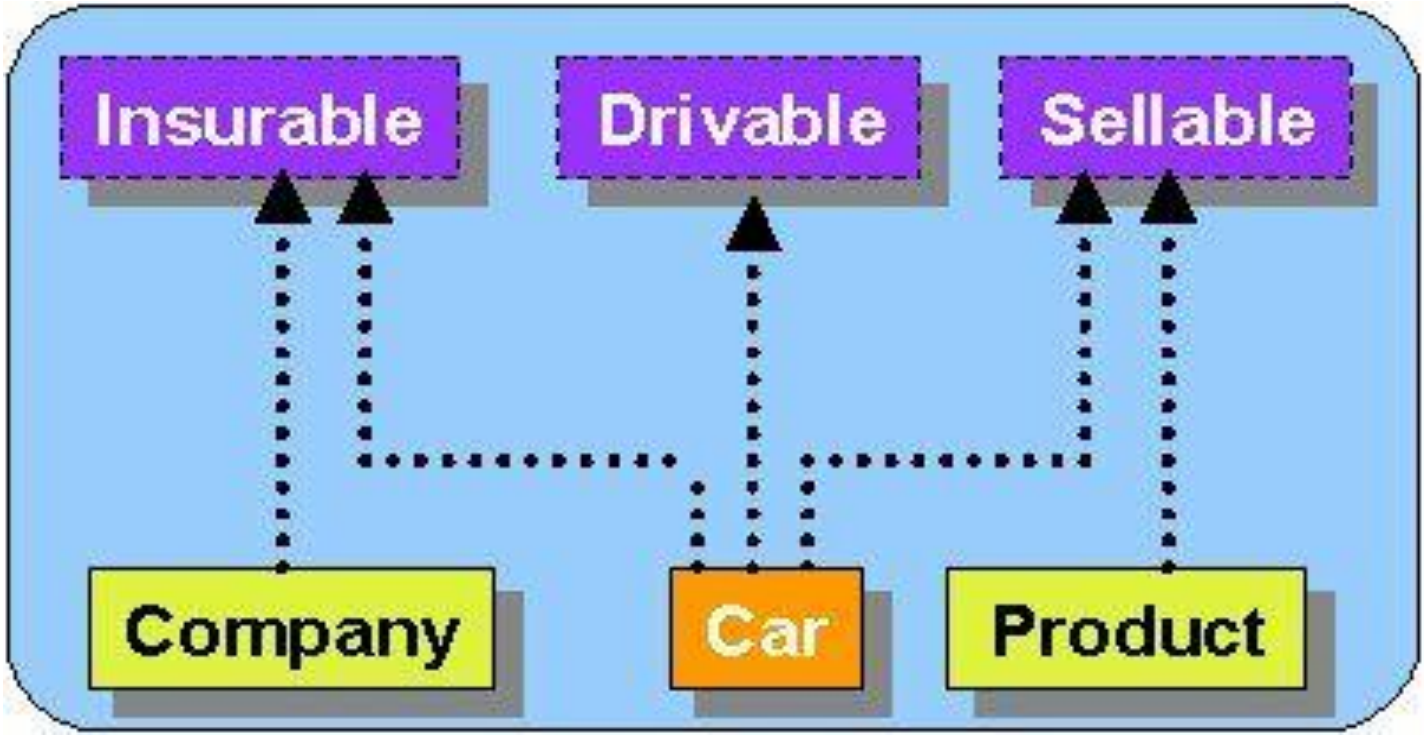
- 多重继承
  - Java的设计以简单实用为导向，不允许一个类有多个父类
  - 但允许一个类可以实现多个接口，通过这种机制可实现多重继承
  - 一个类实现多个接口的语法如下

```
[类修饰符] class  类名称  implements 接口1,接口2, …
{
      …… …
}
```

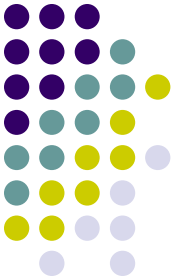
3.12.4 多重继承

- 举例
  - Car类可以实现接口Insurable，Drivable，Sellable

```
public class Car implements Insurable, Drivable, Sellable
{
    ....
}
```



3.12.4 多重继承



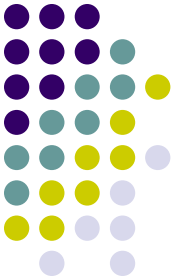
- 举例
- 声明Circle类实现接口Shape2D和Color
  - Shape2D具有pi与area()方法，用来计算面积
  - Color则具有setColor方法，可用来赋值颜色
  - 通过实现这两个接口，Circle类得以同时拥有这两个接口的成员，达到了多重继承的目的

```
interface Shape2D{          //声明Shape2D接口
    final double pi=3.14;    //数据成员一定要初始化
    public abstract double area();   //抽象方法
}
interface Color{
   void setColor(String str);  //抽象方法
}
```

3.12.4 多重继承

```
class Circle implements Shape2D, Color   // 实现Circle类
{
   double radius;
   String color;
   public Circle(double r)            //构造方法
   {
      radius = r;
   }
   public double area()               //定义area()的处理方式
   {
      return (pi*radius*radius);
   }
   public void setColor(String str)  //定义setColor()的处理方式
   {
      color = str;
      System.out.println("Color="+color);
   }
}
```

3.12.4 多重继承



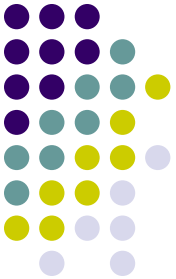
- 测试类

```
public class MultiInterfaceTester{
    public static void main(String args[]) {
        Circle cir;
        cir=new Circle(2.0);
        cir.setColor("blue");
        System.out.println("Area = " + cir.area());
    }
}
```

- 输出结果

Color = blue
Area = 12.56

第三章 Java面向对象程序设计 – 3



3.12 接口

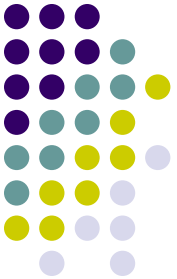
3.13 塑型

3.14 多态

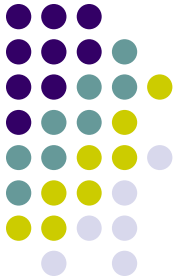
3.15 内部类

3.13 塑型

- 塑型 (type-casting)
  - 又称为类型转换
  - 方式
    - 隐式(自动)的类型转换
    - 显式(强制)的类型转换



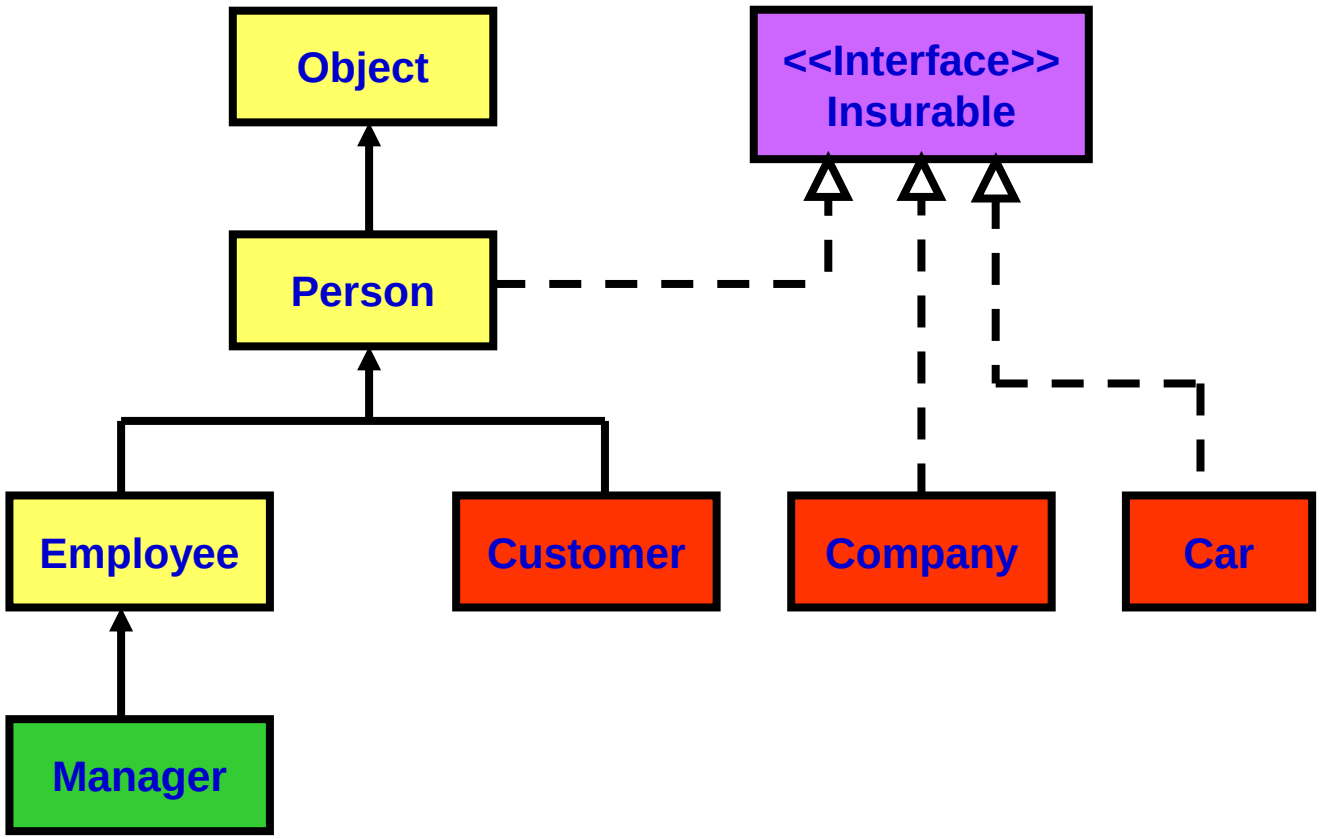
3.13.1 塑型的概念



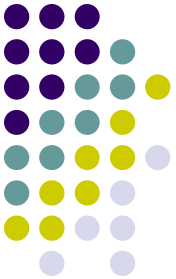
- 塑型的对象包括
  - 基本数据类型
    - 将值从一种形式转换成另一种形式
  - 引用变量
    - 将对象暂时当成更一般的对象来对待，并不改变其类型
    - 只能被塑型为
      - 任何一个父类类型
      - 对象所属的类实现的一个接口
      - 被塑型为父类或接口后，再被塑型回其本身所在的类

3.13.1 塑型的概念

- Manager对象
  - 可以被塑型为Employee、Person、Object或Insurable，
  - 不能被塑型为Customer、Company或Car



3.13.1 塑型的概念



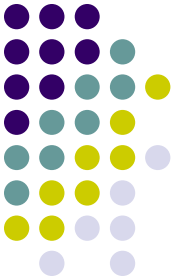
- 隐式(自动)的类型转换
- 基本数据类型
  - 相容类型之间存储容量低的自动向存储容量高的类型转换
- 引用变量
  - 被塑型成更一般的类

```
Employee  emp;
emp = new Manager( );  //将Manager类型的对象直接赋给
                       //Employee类的引用变量，系统会
                       //自动将Manager对象塑型为Employee类
```

  - 被塑型为对象所属类实现的接口类型

```
Car  jetta = new Car( );
Insurable  item = jetta;
```

3.13.1 塑型的概念



- 显式(强制)的类型转换

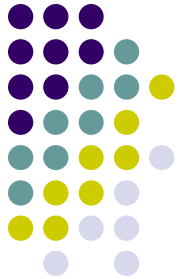
- 基本数据类型

```
(int)871.34354;    // 结果为 871
(char)65;          // 结果为 'A'
(long)453;         // 结果为453L
```

- 引用变量：还原为本来的类型

```
Employee  emp;
Manager   man;
emp = new Manager();
man = (Manager) emp; //将emp强制塑型为本来的类型
```

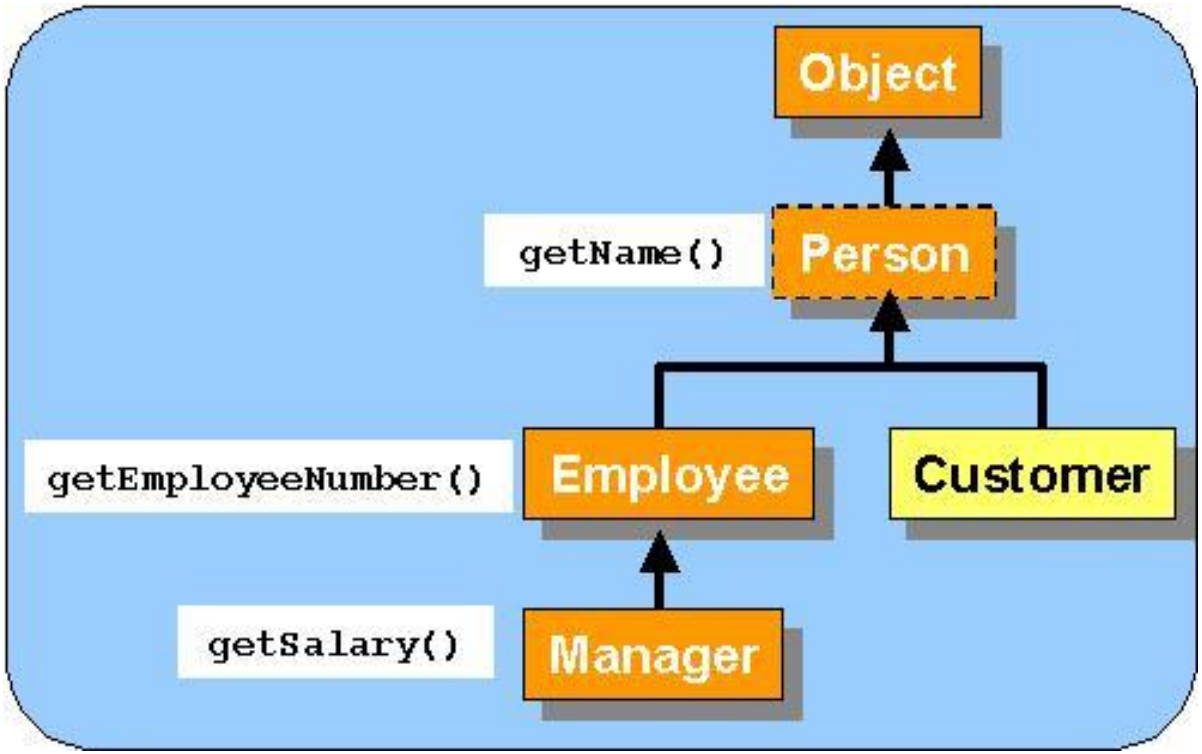
3.13.1 塑型的概念



- 塑型应用的场合
  - 赋值转换
    - 赋值号右边的表达式类型或对象转换为左边的类型
  - 方法调用转换
    - 实参的类型转换为形参的类型
  - 算数表达式转换
    - 算数混合运算时，不同类型的项转换为相同的类型再进行运算
  - 字符串转换
    - 字符串连接运算时，如果一个操作数为字符串，一个操作数为数值型，则会自动将数值型转换为字符串

3.13.2 塑型的应用

- 当一个类对象被塑型为其父类后，它提供的方法会减少
  - 当Manager对象被塑型为Employee之后，它只能接收getName()及getEmployeeNumber()方法，不能接收getSalary()方法
  - 将其塑型为本来的类型后，又能接收getSalary()方法了



3.13.2 塑型的应用

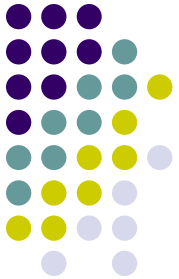
- 声明Circle与Rectangle两个类实现Shape2D接口

```
interface Shape2D{                  //声明Shape2D接口
    final  double pi=3.14;          //数据成员一定要初始化
    public  abstract double area();   //抽象方法
}
```

```
class Circle implements Shape2D
{
    double radius;
    public Circle(double r)
    {
       radius=r;
    }
    public double area()
   {
       return (pi * radius * radius);
   }
}
```

```
class Rectangle implements Shape2D
{
     int width, height;
     public Rectangle(int w, int h)
     {
        width=w;
        height=h;
      }
      public double area()
      {
        return (width * height);
       }
}
```

3.13.2 塑型的应用



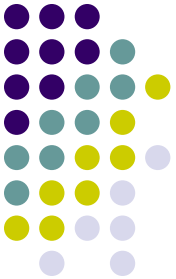
- 声明接口类型的变量，并用它来访问对象

```
public class VariableTester {
        public static void main(String []args)
        {
                Shape2D var1,var2;
                var1 = new  Rectangle(5,6);  //对象塑型
                System.out.println("Area of var1 = " + var1.area());
                var2 = new Circle(2.0);   //对象塑型
                System.out.println("Area of var2 = " + var2.area());
        }
}
```

- 运行结果

Area of var1 = 30.0
Area of var2 = 12.56

第三章 Java面向对象程序设计 – 3



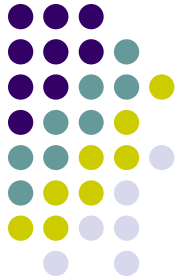
3.12 接口

3.13 塑型

3.14 多态

3.15 内部类

3.14 多态



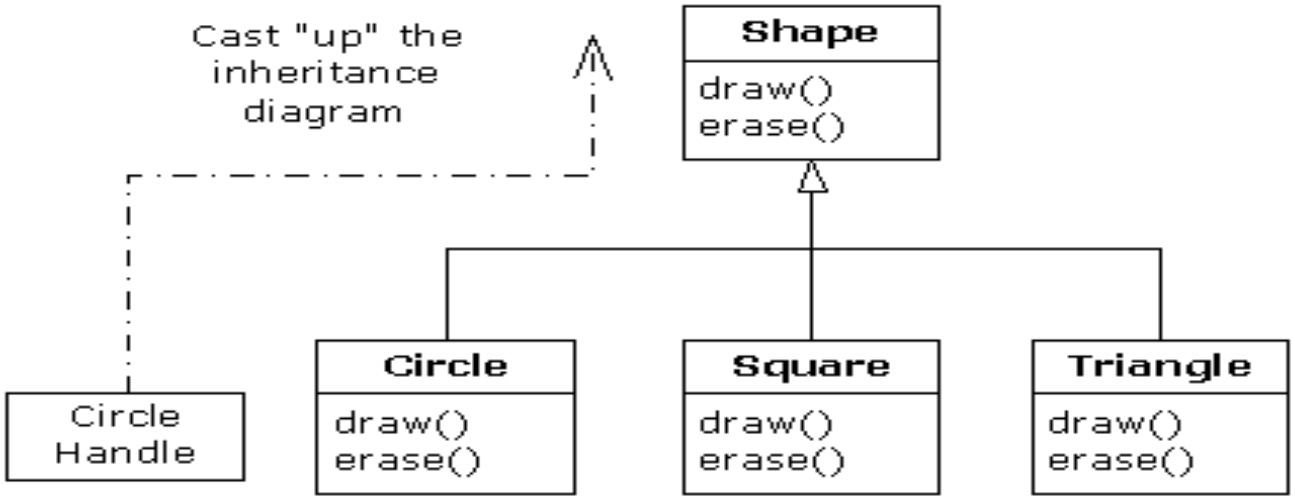
- 多态 (polymorphism)
  - “多态性”一词最早用于生物学，指同一种族的生物体具有相同的特性。
  - 在面向对象理论中，多态性的定义是：同一操作用于不同的类的实例，不同的类将进行不同的解释，最后产生不同的结果。
  - 简单地说：同样一条语句，在不同的情况下可能产生不同的运行结果。

3.14.1 多态的概念

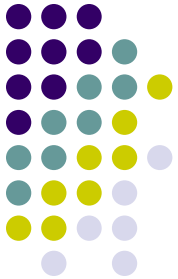
- 例子：绘图——更好的方式
  - 在每个子类中都声明同名的draw()方法
  - 以后绘图可如下进行

```
Shape s = new Circle();
s.draw();
```

  - Circle属于Shape的一种，系统会执行自动塑型
  - 当调用方法draw时，实际调用的是Circle.draw( )
  - 在程序运行时才进行绑定，接下来介绍绑定的概念



3.14.1 多态的概念



- **多态**
  - 是指不同类型的对象可以响应相同的消息
  - 从相同的基类派生出来的多个类型可被当作同一种类型对待，这些不同派生类对象响应同一方法时的行为是有所差别的
  - 例如
    - 所有的Object类的对象都响应toString()方法
    - 所有的BankAccount类的对象都响应deposit()方法
- **多态的目的**
  - 所有的对象都可被塑型为相同的类型，响应相同的消息
  - 使代码变得简单且容易理解
  - 使程序具有很好的“扩展性”

3.14.2 绑定的概念

- 仍以绘图为例，所有类都放在binding包中
  - 基类Shape

```
class Shape {
    void draw()   { }
    void erase()  { }
}
```

  - 派生类覆盖了draw方法，为每种特殊的几何形状都提供独一无二的行为

```
class Circle extends Shape {
    void draw()
      { System.out.println("Circle.draw()"); }
    void erase()
      { System.out.println("Circle.erase()"); }
}
```

3.14.2 绑定的概念

```
class Square extends Shape {
     void draw()
          { System.out.println("Square.draw()"); }
     void erase()
          { System.out.println("Square.erase()"); }
}
```

```
class Triangle extends Shape {
    void draw()
          { System.out.println("Triangle.draw()"); }
    void erase()
          { System.out.println("Triangle.erase()"); }
 }
```

3.14.2 绑定的概念

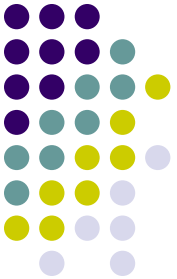
- 对动态绑定进行测试如下

- 运行结果

```
Square.draw()
Triangle.draw()
Circle.draw()
Triangle.draw()
Triangle.draw()
Circle.draw()
Square.draw()
Circle.draw()
Triangle.draw()
```

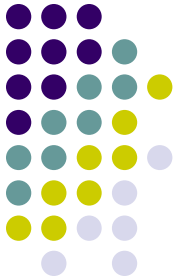
```
public class BindingTester{
    public static void main(String[] args) {
        Shape[] s = new Shape[9];
        int n;
        for(int i = 0; i < s.length; i++) {
            n = (int)(Math.random() * 3);
            switch(n) {
                case 0: s[i] =  new Circle(); break;
                case 1: s[i] =  new Square(); break;
                case 2: s[i] =  new Triangle();
            }
        }
        for(int i = 0; i < s.length; i++)
                s[i].draw();
    }
}
```

3.14.2 绑定的概念



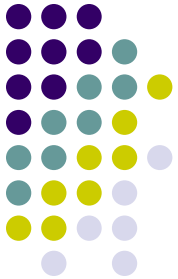
- 说明
  - 编译时无法知道s数组元素的具体类型，运行时才能确定类型，所以是动态绑定
  - 在主方法的循环体中，每次随机生成指向一个Circle、Square或者Triangle的引用

3.14.2 绑定的概念



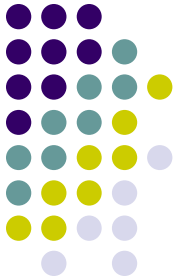
- 绑定(binding)
  - 指将一个方法调用同一个方法主体连接到一起
  - 根据绑定时期的不同，可分为
    - 早期绑定（静态绑定）
      - 程序运行之前执行绑定
    - 晚期绑定（动态绑定）
      - 也叫作“动态绑定”或“运行期绑定
      - 基于对象的类别，在程序运行时执行绑定

3.14.3 多态的应用



- 多态的应用
  - 技术基础
    - 向上塑型技术：一个父类的引用变量可以指向不同的子类对象
    - 动态绑定技术：运行时根据父类引用变量所指对象的实际类型执行相应的子类方法，从而实现多态性

3.14.3 多态的应用



- 例子
  - 声明一个抽象类Driver及两个子类FemaleDriver及MaleDriver
  - 在Diver类中声明了抽象方法drives，在两个子类中对这个方法进行了重写

```
public abstract class Driver
{
    public Driver( ) { }
    public abstract void drives( );
}
```

3.14.3 多态的应用

```
public class FemaleDriver extends Driver {
      public FemaleDriver( ) { }
      public void drives( ) {
          System.out.println("A Female driver drives a vehicle.");
    }
}
```

```
public class MaleDriver extends Driver {
     public MaleDriver( ) { }
     public void drives( ) {
          System.out.println("A male driver drives a vehicle.");
   }
}
```

3.14.3 多态的应用

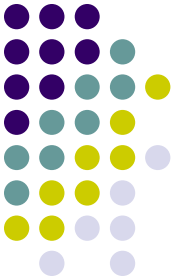
```
public class PolymorphismTester
{
     static public void main(String[ ] args)
    {
         Driver a = new FemaleDriver( );
         Driver b = new MaleDriver( );
         a.drives( );
         b.drives( );
     }
 }
```

- 运行结果

A Female driver drives a vehicle.

A male driver drives a vehicle.

3.14.4 构造方法与多态



- **构造方法与多态**
  - 构造方法与其他方法是有区别的
  - 构造方法并不具有多态性，但仍然非常有必要理解构造方法如何在复杂的分级结构中随同多态性一同使用的情况
- **构造方法的调用顺序**
  - 调用基类的构造方法。
    - 这个步骤会不断重复下去，首先得到构建的是分级结构的根部，然后是下一个派生类，直到抵达最深一层的派生类。
  - 按声明顺序调用成员初始化模块。
  - 调用派生构造方法。

3.14.4 构造方法与多态

- 构建一个点类Point，一个球类Ball，一个运动的球类MovingBall继承自Ball

```
public class Point {
    private double xCoordinate;
    private double yCoordinate;
    public Point ( ) { }
    public Point(double x, double y) {
        xCoordinate = x;   yCoordinate = y;
     }
     public String toString( ){
        return "(" + Double.toString(xCoordinate) + ", “
                   + Double.toString(yCoordinate) + ")";
      }
}
```

```
public class Ball {
        private Point center;     //中心点
        private double radius;    //半径
        private String colour;    //颜色
        public Ball( ) { }
        public Ball(double xValue, double yValue, double r) {
              center = new Point(xValue, yValue); //调用Point中的构造方法
              radius = r;
         }
         public Ball(double xValue, double  yValue, double r, String c) {
              this(xValue, yValue, r); // 调用三个参数的构造方法
              colour = c;
          }
          public String toString( ) {
              return "A ball with center " + center.toString( ) + ", radius "
                  + Double.toString(radius) + ", colour " + colour;
           }
}
```

```
public class MovingBall extends Ball {
    private double speed;
    public MovingBall( ) { }
    public MovingBall(double xValue, double yValue,
                      double r, String c, double s) {
        super(xValue, yValue, r, c);
        speed = s;
    }
    public String toString( ) {
        return super.toString( ) + ", speed " + Double.toString(speed);
    }
}
```

- 子类不能直接存取父类中声明的私有数据成员，super.toString调用父类Ball的toString方法输出类Ball中声明的属性值

3.14.4 构造方法与多态

```
public class Tester{
    public static void main(String args[]){
    MovingBall mb = new MovingBall(10,20,40,"green",25);
    System.out.println(mb);
    }
}
```

- 运行结果

A ball with center (10.0, 20.0), radius 40.0, colour green, speed 25.0

3.14.4 构造方法与多态

- 上面的代码中，构造方法调用的顺序为

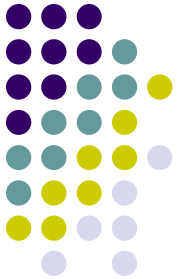
MovingBall(double xValue, double yValue, double r, String c, double s)

Ball(double xValue, double yValue, double r, String c)

Ball(double xValue, double yValue, double r)

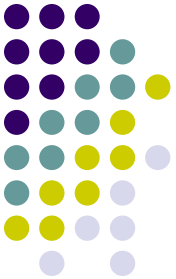
Point(double x, double y)

3.14.4 构造方法与多态



- 构造方法中的多态方法
  - 在构造方法内调用准备构造的那个对象的动态绑定方法
    - 会调用位于派生类里的一个方法
    - 被调用方法要操纵的成员可能尚未得到正确的初始化
    - 可能造成一些难于发现的程序错误

第三章 Java面向对象程序设计 – 3



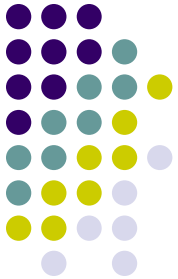
3.12 接口

3.13 塑型

3.14 多态

3.15 内部类

3.15 内部类 (inner class)



- 内部类
  - 在另一个类或方法的定义中定义的类
  - 可访问其外部类中的所有数据成员和方法成员
  - 对于同一个包中的其他类来说，能够隐藏
  - 可非常方便地编写事件驱动程序
  - 声明方式
    - 命名的内部类：可在类的内部多次使用
    - 匿名内部类：可在new关键字后声明内部类，并立即创建一个对象
  - 假设外层类名为Myclass，则该类的内部类名为
    - Myclass$c1.class (c1为命名的内部类名)
    - Myclass$1.class (表示类中声明的第一个匿名内部类)

```
public class Parcel1 {
    class Contents { //内部类
        private int i = 11;
        public int value() { return i; }
    }
    class Destination  { //内部类
        private String label;
        Destination(String whereTo)  {   label = whereTo;    }
        String readLabel() { return label; }
    }
    public void ship(String dest) {
        Contents c = new Contents();
        Destination d = new Destination(dest);
        System.out.println(d.readLabel());
    }
    public static void main(String[] args)  {
        Parcel1 p = new Parcel1();
        p.ship("Tanzania");
    }
}
```

内部类——Parcel1.java

3.15 内部类

- 说明
  - 在Parcel1类中声明了两个内部类Contents、Destination
  - 在ship方法中生成两个内部类对象，并调用了内部类中声明的一个方法

3.15 内部类

- 外部类的方法可以返回内部类的引用变量

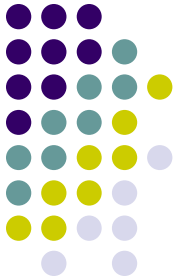
```
public class Parcel2 {
  class Contents  {
     private int i = 11;
     public int value() { return i; }
   }
   class Destination {
      private String label;
      Destination(String whereTo) {  label = whereTo;  }
      String readLabel() { return label; }
    }
   public Destination to(String s)
     { return new Destination(s);  }
   public Contents cont()
     { return new Contents();  }
```

内部类——Parcel2.java

```
public void ship(String dest) {
        Contents c = cont();
        Destination d = to(dest);
        System.out.println(d.readLabel());
    }
    public static void main(String[] args) {
         Parcel2  p = new Parcel2();
         p.ship("Tanzania");
         Parcel2 q = new Parcel2();
         Parcel2.Contents  c = q.cont();
         Parcel2.Destination  d =q.to("Borneo");
     }
}
```

- 说明
  - to()方法返回内部类Destination的引用
  - cont()方法返回内部类Contents的引用

3.15 内部类



- 内部类实现接口
  - 可以完全不被看到，而且不能被调用
  - 可以方便实现“隐藏实现细则”。你所能得到的仅仅是指向基类(base class)或者接口的一个引用
- 例子

```
abstract class Contents {
    abstract public int value();
}
interface Destination {
    String readLabel();
}
```

3.15 内部类

- 例子

```
public class Parcel3 {
    private class PContents extends Contents {
        private int i = 11;
        public int value() { return i; }
    }
    protected class PDestination implements Destination {
        private String label;
        private PDestination(String whereTo) { label = whereTo;}
        public String readLabel() { return label; }
     }
    public PDestination dest(String s) { return new PDestination(s); }
    public PContents cont() { return new PContents(); }
}
```

内部类——Parcel3.java

3.15 内部类

```
class Test {
    public static void main(String[] args) {
        Parcel3 p = new Parcel3();
        Contents c = p.cont();
        Destination d = p.dest("Tanzania");
    }
}
```

- 说明
  - 内部类PContents实现了抽象类Contents
  - 内部类PDenstination实现了接口Destination
  - 外部类Test中不能声明对private的内部类的引用

3.15 内部类

- 在方法内定义一个内部类
  - 为实现某个接口，产生并返回一个引用
  - 为解决一个复杂问题，需要建立一个类，而又不想它为外界所用

3.15 内部类

- 例子

```
public class Parcel4 {
    public Destination dest(String s) {
        class PDestination implements Destination {
            private String label;
            private PDestination(String whereTo) { label = whereTo; }
            public String readLabel() { return label; }
        }
        return new PDestination(s);
    }
    public static void main(String[] args) {
        Parcel4  p = new Parcel4();
        Destination d = p.dest("Tanzania");
    }
}
```

内部类——Parcel4.java

3.15 内部类

- 作用域中的内部类

```
public class Parcel5 {
    private void internalTracking(boolean b) {
        if(b) {
            class TrackingSlip {
                private String id;
                TrackingSlip(String s) { id = s;   }
                String getSlip() { return id; }
             }
            TrackingSlip ts = new TrackingSlip("slip");
            String s = ts.getSlip();
         }
    }
    public void track() { internalTracking(true); }
    public static void main(String[] args)  {
        Parcel5 p = new Parcel5();
        p.track();
    }
}
```

内部类——Parcel5.java

3.15 内部类

- 匿名的内部类

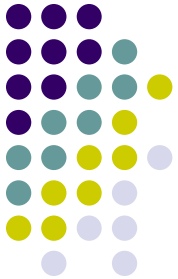
```
public class Parcel6 {
    public Contents cont() {
        return new Contents() {
            private int i = 11;
            public int value() { return i; }
        };
    }
    public static void main(String[] args) {
        Parcel6 p = new Parcel6();
        Contents c = p.cont();
    }
}
```

内部类——Parcel6.java

小结

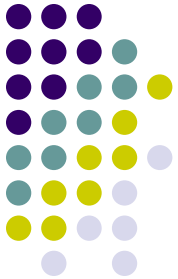
- 内容
  - 接口作用及语法
  - 塑型的概念及应用
  - 多态的概念及引用
  - 构造方法的调用顺序及其中的多态方法
  - 内部内的有关知识
- 要求
  - 理解接口、塑型、多态的概念并能熟练应用
  - 熟练掌握构造方法的调用顺序，理解编写时需要注意的问题
  - 掌握内部类的语法结构及其应用场合

第三章要点



- 第三章内容
  - 面向对象程序设计的基本概念和思想
  - Java语言类与对象的基本概念和语法，包括类的声明、类成员的访问，以及对象的构造、初始化和回收
  - 方法的重载
  - 介绍了Java语言类的重用机制，形式可以是组合或继承
  - Object类的主要方法
  - 终结类和终结方法的特点和语法
  - 抽象类和抽象方法的特点和语法
  - Java基础类库的一些重要的类
  - 接口作用及语法
  - 塑型的概念及应用
  - 多态的概念及引用

第三章要点



- 第三章要求
  - 理解类和对象的概念
  - 熟练使用类及其成员的访问控制方法
  - 熟练掌握各种构造方法
  - 了解java的垃圾回收机制
  - 掌握方法重载的含义，并熟练应用
  - 理解组合和继承的区别，能够知道何时使用那种方法
  - 了解终结类、终结方法、抽象类、抽象方法的概念
  - 熟练掌握本章提到的Java基础类库中的一些常见类
  - 理解接口、塑型、多态的概念并能熟练应用