



# JAVA程序设计

潘微科

感谢：教材《Java大学实用教程》的作者和其他老师提供PowerPoint讲义等资料！  
说明：本课程所使用的所有讲义，都是在以上资料上修改的。

# Outline

- 5.1 子类与父类
- 5.2 子类的继承性
- 5.3 子类对象的构造过程
- 5.4 成员变量隐藏和方法重写
- 5.5 关键字super
- 5.6 final类与final方法
- 5.7 对象的上转型对象
- 5.8 继承与多态
- 5.9 abstract类
- 5.10 面向抽象
- 5.11 接口
- 5.12 接口回调
- 5.13 面向接口
- 5.14 抽象类与接口的比较
- 5.15 内部类
- 5.16 匿名类
- 5.17 异常类
- 5.18 泛型类

## 5.1 子类与父类

- 第4章：类、对象
- 第5章：类的继承、与继承有关的多态性（polymorphism）、接口（interface）、泛型（generics）

## 5.1 子类与父类

- 继承（**inheritance**）是一种由已有的类创建新类的机制。
- 利用继承，我们可以先创建一个共有属性的一般类，根据该一般类再创建具有特殊属性的新类。
- 新类继承一般类的属性（状态）和功能（行为），并根据需要增加它自己的新的属性（状态）和功能（行为）。
- 由继承而得到的类称为子类（**subclass, child class, or extended class**），被继承的类称为父类（**superclass, parent class, or base class**）。

## 5.1 子类与父类

- 父类可以是自己编写的类也可以是Java类库中的类。
- 利用继承有利于实现代码的**重用**，子类只需要添加新的属性、功能。
- Java不支持多重继承，即**子类只能有一个父类**。
- 使用关键字**extends**来声明一个类是另外一个类的子类：

```
class 子类名 extends 父类名
{
    ...
}
```

## 5.1 子类与父类

- Every class in Java is descended from **java.lang.Object** class. If no inheritance is specified when a class is defined, its superclass is **Object**.

# Outline

- 5.1 子类与父类
- 5.2 子类的继承性
- 5.3 子类对象的构造过程
- 5.4 成员变量隐藏和方法重写
- 5.5 关键字super
- 5.6 final类与final方法
- 5.7 对象的上转型对象
- 5.8 继承与多态
- 5.9 abstract类
- 5.10 面向抽象
- 5.11 接口
- 5.12 接口回调
- 5.13 面向接口
- 5.14 抽象类与接口的比较
- 5.15 内部类
- 5.16 匿名类
- 5.17 异常类
- 5.18 泛型类

## 5.2 子类的继承性

- 1.继承的定义
- 所谓类继承就是子类继承父类的成员变量和方法作为自己的成员变量和方法，就好象它们是在子类中直接声明的一样。当然，子类能否继承父类的成员变量和方法还有一定的限制。
- 2.子类和父类在同一包中的继承性
- 如果子类和父类在同一包中，那么子类自然地继承了父类中不是private的成员变量（即：**friendly, protected, public**）作为自己的成员变量，并且也自然地继承了父类中不是private的方法（即：**friendly, protected, public**）作为自己的方法。继承的成员变量和方法的访问权限保持不变。



## 5.2 子类的继承性

- 【例子】

```
class Father
{
    int money=100;
    int add(int x, int y)
    {
        return x+y;
    }
}
```

```
class Son extends Father
{
    public void changeMoney(int x)
    {
        money=x;
    }
}
```

```
public class Example5_1
{
    public static void main(String args[])
    {
        Son son = new Son();
        son.changeMoney(5000);
        System.out.println("son.money: " + son.money);
        System.out.printf("son.add(a,b): %d\n", son.add(1,2));
    }
}
```

```
son.money: 5000
son.add(1,2): 3
```

## 5.2 子类的继承性

- 3.子类和父类不在同一包中的继承性
- 如果子类和父类不在同一个包中，那么子类**只能继承**父类的**protected, public成员变量和方法**，继承的成员变量和方法的**访问权限保持不变**。
- 如果子类和父类不在同一个包里，子类**不能继承**父类的**friendly成员变量和friendly方法**。

**friendly/默认**: 关键看是否在同一个包中

# Outline

- 5.1 子类与父类
- 5.2 子类的继承性
- 5.3 子类对象的构造过程
- 5.4 成员变量隐藏和方法重写
- 5.5 关键字super
- 5.6 final类与final方法
- 5.7 对象的上转型对象
- 5.8 继承与多态
- 5.9 abstract类
- 5.10 面向抽象
- 5.11 接口
- 5.12 接口回调
- 5.13 面向接口
- 5.14 抽象类与接口的比较
- 5.15 内部类
- 5.16 匿名类
- 5.17 异常类
- 5.18 泛型类

## 5.3 子类对象的构造过程

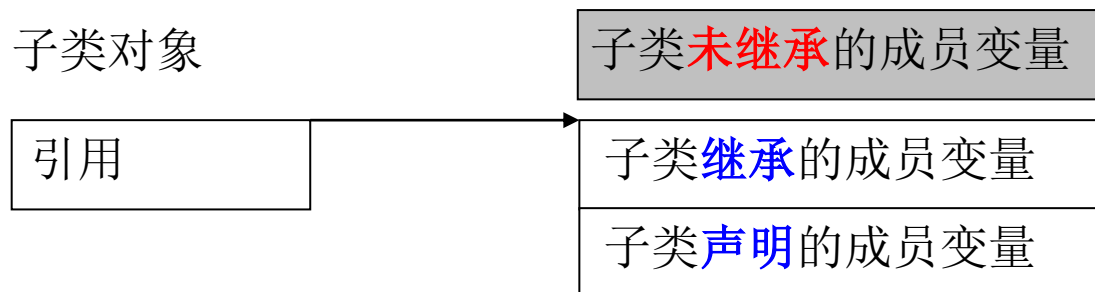
- 当用子类的构造方法创建一个子类的对象时，**子类的构造方法总是先调用父类的某个构造方法。**
- 如果子类的构造方法没有指明使用父类的哪个构造方法，子类就调用父类的不带参数的构造方法。
- 注：**父类的构造方法（constructor）不会被子类继承。**

## 5.3 子类对象的构造过程

- 子类如何创建对象？
  - 将子类中声明的成员变量作为子类对象的成员变量。
  - 父类的成员变量也都分配了内存空间，但**只将其中一部分（继承的那部分）作为子类对象的成员变量**。父类的`private`成员变量尽管分配了内存空间，但它不作为子类的成员变量，即父类的`private`成员变量不归子类管理。
  - 方法的继承性与成员变量的继承性**相同**。
  - 如果子类和父类**不在同一包中**，尽管父类的`friendly`成员变量分配了内存空间，也不作为子类的成员变量。

## 5.3 子类对象的构造过程

- 子类对象的内存示意图



子类中**声明定义**的方法**不**  
**可以操作**这些内存单元

子类中**声明定义**的方法  
**可以操作**这些内存单元

## 5.3 子类对象的构造过程

- 子类创建对象时**似乎浪费了一些内存**，因为当用子类创建对象时，父类的成员变量也都分配了内存空间，但只将其中一部分作为子类对象的成员变量。
- 实际情况并非如此，子类中**还有一部分方法是从父类继承的**，这部分方法却可以操作这部分没有继承的变量。

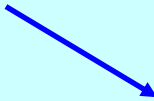
## 5.3 子类对象的构造过程

- 【例子】

```
class B extends A
{
    void g()
    {
        y=y+1;
        System.out.println(y);
    }
}
```

```
class A
{
    private int x=10;
    protected int y=20;
    void f()
    {
        y=y+x;
        System.out.println(y);
    }
}
```

```
class Example5_2
{
    public static void main(String args[])
    {
        B b=new B();
        b.g();
        b.f();
        b.g();
    }
}
```



21  
31  
32

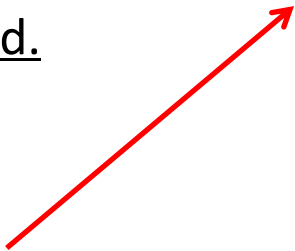


# Outline

- 5.1 子类与父类
- 5.2 子类的继承性
- 5.3 子类对象的构造过程
- 5.4 成员变量隐藏和方法重写
- 5.5 关键字super
- 5.6 final类与final方法
- 5.7 对象的上转型对象
- 5.8 继承与多态
- 5.9 abstract类
- 5.10 面向抽象
- 5.11 接口
- 5.12 接口回调
- 5.13 面向接口
- 5.14 抽象类与接口的比较
- 5.15 内部类
- 5.16 匿名类
- 5.17 异常类
- 5.18 泛型类

## 5.4 成员变量隐藏和方法重写

- 1.成员变量的隐藏
- 子类可以隐藏继承的成员变量，当在子类中定义和父类中**同名的成员变量时**，子类就隐藏了继承的成员变量。
- Within a class, a field that has **the same name** as a field in the superclass hides the superclass's field, **even if their types are different**. Within the subclass, the field in the superclass cannot be referenced by its simple name. Instead, the field must be accessed through **super**. Generally speaking, we **don't** recommend hiding fields as it makes code difficult to read.



## 5.4 成员变量隐藏和方法重写

- 【例子】

```
class B extends A
{
    → int y=0;
    public void g()
    {
        y=y+100;
        System.out.printf("y (int), y=%d\n",y);
    }
}
```

```
class A
{
    public double y=11.456789;
    public void f()
    {
        y=y+1;
        System.out.printf("y
(double), y=%f\n",y);
    }
}
```

```
class Example5_3
{
    public static void main(String args[])
    {
        B b=new B();
        b.y=200;
        b.g(); //调用子类新增的方法
        b.f(); //调用子类继承的方法
    }
}
```

```
y (int), y=300
y (double), y=12.456789
```

## 5.4 成员变量隐藏和方法重写

- 2.方法重写
- 子类也可以隐藏方法，**子类通过方法重写（overriding）来隐藏继承的方法。**
- 方法重写：子类中定义一个方法，并且这个方法的名字、返回类型、参数个数和类型与从父类继承的方法完全相同。
- 如果子类想使用被隐藏的方法，必须使用关键字super，我们将在后面讲述super的用法。

## 5.4 成员变量隐藏和方法重写

- 【例子】 子类隐藏了继承的成员变量y

```
class A
{
    protected double x=8.0,y=0.888888;
    public void speak()
    {
        System.out.println("I Love NBA");
    }
    public void cry()
    {
        y=x+y;
        System.out.printf("y=%f\n",y);
    }
}
```

```
class B extends A
{
    int y=100,z;
    public void speak()
    {
        z=2*y;
        System.out.println("I Love This Game");
        System.out.printf("y=%d,z=%d",y,z);
    }
}
```

方法重写

```
public class Example5_4
{
    public static void main(String args[])
    {
        B b=new B();
        b.cry();
        b.speak();
    }
}
```

```
y=8.888888
I love This Game
y=100,z=200
```

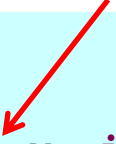
## 5.4 成员变量隐藏和方法重写

- 如果子类在准备隐藏继承的方法时，**参数个数或参数类型与父类的方法不尽相同**，那实际上也没有隐藏继承的方法，这时子类就出现两个方法具有相同的名字，即重载（**overloading**）。

## 5.4 成员变量隐藏和方法重写

- 【例子】

```
class B extends A
{
    public int f(byte x, int y)
    {
        return x*y;
    }
}
```



```
public class Example5_5
{
    public static void main(String args[])
    {
        B b=new B();
        System.out.println(b.f(10,10));
        System.out.println(b.f((byte)10,10));
    }
}
```

```
class A
{
    public int f(int x, int y)
    {
        return x+y;
    }
}
```

```
20
100
```

## 5.4 成员变量隐藏和方法重写

- **Overloading** (重载) means to define **multiple methods with the same name** but **different signatures**.
- **Overriding** (重写) means to **provide a new implementation** for a method in the **subclass**.

Overloading与overriding的区别



## 5.4 成员变量隐藏和方法重写

- 3.访问修饰符protected的进一步说明
- 一个类A中的protected成员变量和方法可以被它的直接子类 and 间接子类继承，比如B是A的子类，C是B的子类，D又是C的子类，那么B、C和D类都继承了A的protected成员变量和方法。
- 如果用D类在D本身中创建了一个对象，那么该对象总是可以通过“.”运算符访问继承的或自己定义的protected变量和protected方法的。
- 但是，如果在另外一个类中，比如E类，用D类创建了一个对象d，该对象通过“.”运算符访问protected成员变量和protected方法的权限如下：

## 5.4 成员变量隐藏和方法重写

- 子类D的protected成员变量和方法，如果不是从父类继承来的，对象d访问这些protected成员变量和方法时，只要E类和D类在同一个包中就可以了。
- 如果子类D的对象的protected成员变量或protected方法是“从父类继承的”，那么就要一直追溯到该protected成员变量或方法的“祖先”类，即A类，如果E类和A类在同一个包中，对象能访问继承的protected成员变量和protected方法。

关键看是否在同一个包中

# Outline

- 5.1 子类与父类
- 5.2 子类的继承性
- 5.3 子类对象的构造过程
- 5.4 成员变量隐藏和方法重写
- 5.5 关键字super
- 5.6 final类与final方法
- 5.7 对象的上转型对象
- 5.8 继承与多态
- 5.9 abstract类
- 5.10 面向抽象
- 5.11 接口
- 5.12 接口回调
- 5.13 面向接口
- 5.14 抽象类与接口的比较
- 5.15 内部类
- 5.16 匿名类
- 5.17 异常类
- 5.18 泛型类

## 5.5 关键字super

- super关键字有两种用法：
  - 在子类中使用super调用父类的构造方法
  - 在子类中使用super调用被子类隐藏的成员变量和方法
- 1.使用super调用父类的构造方法
- 子类不继承父类的构造方法，因此，子类如果想使用父类的构造方法，必须在子类的构造方法中使用关键字super来表示，而且super必须是子类构造方法中的第一条语句。

## 5.5 关键字super

```
class A
{
    int x;
    A()
    {
        x=100;
    }
    A(int x)
    {
        this.x=x;
    }
}
```

```
class B extends A
{
    int z;
    B(int x)
    {
        super(x);
        z=30;
    }
    B()
    {
        super(); //可以省略
        z=300;
    }
    public void f()
    {
        System.out.printf("x=%d,z=%d\n",x,z);
    }
}
```

```
public class Example5_6
{
    public static void main(String args[])
    {
        B b1=new B(10);
        b1.f();
        B b2=new B();
        b2.f();
    }
}
```

```
x=10,z=30
x=100,z=300
```

## 5.5 关键字super

- 2.使用super操作被隐藏的成员变量和方法
- 如果我们在子类中想使用被子类隐藏的**成员变量**或**方法**就可以使用关键字super 。

## 5.5 关键字super

- 【例子】

```
class B extends A
{
    int m=1;
    long f()
    {
        → super.m=10;
        → return super.f()+m;
    }
    long g()
    {
        → return super.f()/2;
    }
}
```

```
class A
{
    int m=0;
    long f()
    {
        return m;
    }
}
```

```
public class Example5_7
{
    public static void main(String args[])
    {
        B b=new B();
        b.m=3;
        System.out.println(b.g());
        System.out.println(b.f());
        System.out.println(b.g());
    }
}
```

```
0
13
5
```

# Outline

- 5.1 子类与父类
- 5.2 子类的继承性
- 5.3 子类对象的构造过程
- 5.4 成员变量隐藏和方法重写
- 5.5 关键字super
- 5.6 final类与final方法
- 5.7 对象的上转型对象
- 5.8 继承与多态
- 5.9 abstract类
- 5.10 面向抽象
- 5.11 接口
- 5.12 接口回调
- 5.13 面向接口
- 5.14 抽象类与接口的比较
- 5.15 内部类
- 5.16 匿名类
- 5.17 异常类
- 5.18 泛型类



## 5.6 final类与final方法

- final类不能被继承，即不能有子类

```
final class A
{
    ...
}
```

- 将一个类声明为final类一般是由于**安全性**考虑。因为一旦一个方法被修饰为**final方法**，则这个方法不能被重写（overriding），即不允许子类通过重写来隐藏继承的**final方法**。

# Outline

- 5.1 子类与父类
- 5.2 子类的继承性
- 5.3 子类对象的构造过程
- 5.4 成员变量隐藏和方法重写
- 5.5 关键字super
- 5.6 final类与final方法
- 5.7 对象的上转型对象
- 5.8 继承与多态
- 5.9 abstract类
- 5.10 面向抽象
- 5.11 接口
- 5.12 接口回调
- 5.13 面向接口
- 5.14 抽象类与接口的比较
- 5.15 内部类
- 5.16 匿名类
- 5.17 异常类
- 5.18 泛型类

## 5.7 对象的上转型对象

- 1.对象的上转型
- A subclass (子类) is a specialization of its superclass (父类); **every instance of a subclass is also an instance of its superclass**, but not vice versa.
- 假设B是A的子类或间接子类，我们用子类B创建一个对象，可以把这个对象的引用放到类A声明的对象中。

```
A a;  
B b;  
b = new B();  
a = b;
```

implicit casting, 又称upcasting.  
注: 与之对应的是explicit casting (又称downcasting) .

变量a的**declared type**是class A; 变量a的**actual type**是class B.

## 5.7 对象的上转型对象

- 对象a是对象b的上转型对象，对象的上转型对象的实体是子类负责创建的，但上转型对象会失去原对象的一些属性和功能。

- 子类新增的成员变量
  - 子类新增的方法
  - 被子类继承或隐藏的成员变量
  - 被子类继承或重写的方法
- 
- 对象a
- 对象b

- 上转型对象不能操作子类新增的成员变量和方法
- 上转型对象可以访问被子类继承或隐藏的成员变量（即父类中的变量），也可以调用被子类继承的方法（即父类中的方法）或重写的方法（即被子类重写的方法）

关键看与父类有没有关系（如：继承、隐藏、重写）

## 5.7 对象的上转型对象

- 可以将对象的上转型对象再强制转换到一个子类对象，这时，该子类对象又具备了子类所有的属性和功能。

## 5.7 对象的上转型对象

- 【例子】

```
class B extends A
```

```
{
```

```
    int n=12;
```

```
    void g() ←
```

```
    {
```

```
        System.out.printf("g(): n=%d,m=%d\n",n,m);
```

```
    }
```

```
    void h()
```

```
    {
```

```
        System.out.printf("h(): n=%d,m=%d\n",n,m);
```

```
    }
```

```
}
```

```
class A
```

```
{
```

```
    double n;
```

```
    int m;
```

```
    void f()
```

```
    {
```

```
        System.out.printf("f(): n=%f,m=%d\n",n,m);
```

```
    }
```

```
    void g()
```

```
    {
```

```
        System.out.printf("n=%f,m=%d\n",n,m);
```

```
    }
```

```
}
```

## 5.7 对象的上转型对象

- 【例子】

```
public class Example5_8
{
    public static void main(String args[])
    {
        A a;
        a=new B();
        a.n=0.618;
        a.m=200;
        a.f();
        a.g();
        B b=(B)a;
        b.n=555;
        b.h();
    }
}
```

upcasting

A a;

a=new B();

a.n=0.618;

a.m=200;

a.f();

a.g();

downcasting

B b=(B)a;

b.n=555;

b.h();

}

}

被子类隐藏的变量，访问父类中的变量

被子类继承的变量，访问父类中的变量

被子类继承的方法，调用父类中的方法

被子类重写的方法，调用子类重写的方法

```
f(): n=0.618000,m=200
g(): n=12,m=200
h(): n=555,m=200
```

## 5.7 对象的上转型对象

- 不要将父类创建的对象和子类对象的上转型对象混淆，对象的上转型对象的实体是由子类负责创建的，只不过失掉了一些属性和功能而已。
- 关于对象的上转型的好处我们在后面将对对比介绍。



# Outline

- 5.1 子类与父类
- 5.2 子类的继承性
- 5.3 子类对象的构造过程
- 5.4 成员变量隐藏和方法重写
- 5.5 关键字super
- 5.6 final类与final方法
- 5.7 对象的上转型对象
- 5.8 继承与多态
- 5.9 abstract类
- 5.10 面向抽象
- 5.11 接口
- 5.12 接口回调
- 5.13 面向接口
- 5.14 抽象类与接口的比较
- 5.15 内部类
- 5.16 匿名类
- 5.17 异常类
- 5.18 泛型类

## 5.8 继承与多态

- **和继承有关的多态**是指父类的某个方法被其子类重写（overriding）时，可以产生自己的功能行为。
- 例如，狗和猫都具有哺乳类的功能“叫声”；当狗操作“叫声”时产生的声音是“汪汪...；而猫操作“叫声”时产生的声音是“喵喵”；这就是“叫声”的多态。
- 当一个类有**多个子类**时，并且这些子类都**重写（overriding）**了父类中的某个方法，我们把子类创建的对象引用放到该父类的对象中时，就得到了该对象的一个**上转型对象**，那么**这个上转型对象在调用这个方法时就可能具有多种形态**（polymorphism, from a Greek word meaning "many forms"）。

## 5.8 继承与多态

- Polymorphism:
  - An object of a subclass can be used **wherever** its superclass is used.
  - A variable of a superclass (or supertype) can **refer to** a subclass (or subtype) object.
  - The **JVM** dynamically determines which of these methods to invoke **at runtime**, depending on the **actual object** that invokes the method.

## 5.8 继承与多态

- 【例子】

```
class Animal
{
    void cry(){}
}
```

```
class Dog extends Animal
{
    void cry()
    {
        System.out.println("Wang!...");
    }
}
```

```
class Cat extends Animal
{
    void cry()
    {
        System.out.println("miao~~");
    }
}
```

```
public class Example5_9
{
    public static void main(String args[])
    {
        Animal animal;
        animal=new Dog();
        animal.cry();
        animal=new Cat();
        animal.cry();
    }
}
```

```
Wang!...
miao~~~
```

# Outline

- 5.1 子类与父类
- 5.2 子类的继承性
- 5.3 子类对象的构造过程
- 5.4 成员变量隐藏和方法重写
- 5.5 关键字super
- 5.6 final类与final方法
- 5.7 对象的上转型对象
- 5.8 继承与多态
- 5.9 abstract类
- 5.10 面向抽象
- 5.11 接口
- 5.12 接口回调
- 5.13 面向接口
- 5.14 抽象类与接口的比较
- 5.15 内部类
- 5.16 匿名类
- 5.17 异常类
- 5.18 泛型类

## 5.9 abstract类

- 用关键字abstract修饰的类称为abstract类（抽象类）。

```
abstract class A
{
    ...
}
```

## 5.9 abstract类

- abstract类中如果自己提供constructor（构造方法），则用**protected**修饰为好，因为是给子类用的。
- **abstract类不能用new运算符创建对象**，必须产生其子类，由子类创建对象。
- 如果abstract类的类体中有**abstract方法**，只允许声明，而不允许实现；而该类的**非abstract子类**必须实现**abstract方法**，即重写（override）父类中的abstract方法。
- 一个abstract类只关心子类**是否具有某种功能**，不关心功能的具体实现。

## 5.9 abstract类

- An abstract method cannot be contained in a nonabstract class (i.e., concrete class). If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be defined as abstract. In other words, in a nonabstract subclass extended from an abstract class, all the abstract methods must be implemented. Also note that abstract methods are nonstatic.
- An abstract class cannot be instantiated using the new operator, but you can still define its constructors, which are invoked in the constructors of its subclasses.



## 5.9 abstract类

- A class that contains abstract methods must be abstract. However, it is possible to define an abstract class that doesn't contain any abstract methods. In this case, you cannot create instances of the class using the new operator. This class is used as a base class for defining subclasses.
- A subclass can override a method from its superclass to define it as abstract. This is very unusual, but it is useful when the implementation of the method in the superclass becomes invalid in the subclass. In this case, the subclass must be defined as abstract.

## 5.9 abstract类

- A subclass can be abstract even if its superclass is concrete. For example, the Object class is concrete, but its subclasses may be abstract.
- You cannot create an instance from an abstract class using the new operator, but an abstract class can be used as a data type.

# Outline

- 5.1 子类与父类
- 5.2 子类的继承性
- 5.3 子类对象的构造过程
- 5.4 成员变量隐藏和方法重写
- 5.5 关键字super
- 5.6 final类与final方法
- 5.7 对象的上转型对象
- 5.8 继承与多态
- 5.9 abstract类
- 5.10 面向抽象
- 5.11 接口
- 5.12 接口回调
- 5.13 面向接口
- 5.14 抽象类与接口的比较
- 5.15 内部类
- 5.16 匿名类
- 5.17 异常类
- 5.18 泛型类

## 5.10 面向抽象

- 面向抽象的核心思想
- 抽象细节
  - 面向抽象的第一步就是将**经常需要变化的细节**分割出来，将其作为**abstract类中的abstract方法**，不让设计者去关心实现的细节，避免所设计的类依赖这些细节。
- 面向抽象来设计类
  - 面向抽象编程的第二步就是**继承**抽象类，进而设计一个新类。

## 5.10 面向抽象

- 【例子1/3】

```
→ public abstract class Geometry
{
    → public abstract double getArea();
}
```

```
public class Circle extends Geometry
{
    double r;
    Circle(double r)
    {
        this.r = r;
    }
    public double getArea()
    {
        return(3.14*r*r);
    }
}
```

```
public class Lader extends Geometry
{
    double a,b,h;
    Lader(double a,double b,double h)
    {
        this.a = a;
        this.b = b;
        this.h = h;
    }
    public double getArea()
    {
        return((1/2.0)*(a+b)*h);
    }
}
```

## 5.10 面向抽象

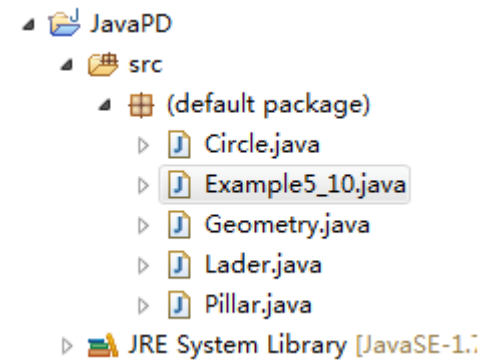
- 【例子2/3】

```
public class Pillar
{
    Geometry bottom;
    double height;
    Pillar (Geometry bottom, double height)
    {
        this.bottom = bottom;
        this.height = height;
    }
    void changeBottom(Geometry bottom)
    {
        this.bottom = bottom;
    }
    public double getVolume()
    {
        return bottom.getArea()*height;
    }
}
```

## 5.10 面向抽象

- 【例子3/3】

```
Lader area: 1.0  
Pillar volume: 1.0  
Circle area: 3.14  
Pillar volume: 3.14
```



```
public class Example5_10  
{  
    public static void main(String args[])  
    {  
        Pillar pillar;  
        Geometry geometry;  
        → geometry = new Lader(1,1,1);  
        System.out.println("Lader area: " + geometry.getArea());  
  
        pillar = new Pillar (geometry,1);  
        System.out.println("Pillar volume: " + pillar.getVolume());  
  
        → geometry = new Circle(1);  
        System.out.println("Circle area: " + geometry.getArea());  
  
        pillar.changeBottom(geometry);  
        System.out.println("Pillar volume: " + pillar.getVolume());  
    }  
}
```

# Outline

- 5.1 子类与父类
- 5.2 子类的继承性
- 5.3 子类对象的构造过程
- 5.4 成员变量隐藏和方法重写
- 5.5 关键字super
- 5.6 final类与final方法
- 5.7 对象的上转型对象
- 5.8 继承与多态
- 5.9 abstract类
- 5.10 面向抽象
- 5.11 接口
- 5.12 接口回调
- 5.13 面向接口
- 5.14 抽象类与接口的比较
- 5.15 内部类
- 5.16 匿名类
- 5.17 异常类
- 5.18 泛型类



## 5.11 接口

- Java不支持多重继承，即一个类只能继承一个父类。
- 单继承使得Java程序简单、易于管理。
- 为了克服单继承的缺点，Java使用了**接口**，一个类可以**实现**多个接口。
- 使用关键字**interface**来定义一个接口。接口的定义和类的定义很相似，分为接口的声明和接口体。
- 1.接口的声明与使用
- （1）接口声明：通过使用关键字**interface**来声明

**interface** 接口的名字

## 5.11 接口

- (2) 接口体
- 接口体中只能包含**常量定义**和**方法定义**两部分。接口体中只进行方法的声明，**不许提供方法的实现**，所以，方法的定义没有方法体，且**用分号“;”结尾**。
- (3) 接口的使用
- 一个类通过使用关键字**implements**声明实现一个或多个接口。如果实现多个接口，用逗号隔开接口名，如：

```
class A implements Printable, Addable
```

## 5.11 接口

- 如果一个类实现某个接口，那么这个类可以实现该接口中的所有（或部分）方法，如果只实现部分方法，则为抽象类
- 接口中的常量用**public static final**来修饰，但可以省略public static final
- 接口中的方法用**public abstract**来修饰，但可以省略public abstract
- 在实现接口中的方法时，一定要用**public**来修饰，**不可以省略**
- 如果父类实现了某个接口，则其子类也就自然实现了这个接口。
- **接口也可以被继承**，即可以通过关键字**extends**声明一个接口是另一个接口的子接口。

## 5.11 接口

- 【例子1/2】

```
class B implements Computable
{
    public int f(int x)
    {
        return x*x*x;
    }
    public int g(int x,int y)
    {
        return x*y;
    }
}
```

```
interface Computable
{
    final int MAX=100; // public static final
    int f(int x); // public abstract
    public abstract int g(int x,int y);
}
```

```
class A implements Computable
{
    public int f(int x)
    {
        return x*x;
    }
    public int g(int x,int y)
    {
        return x+y;
    }
}
```

## 5.11 接口

- 【例子2/2】

```
public class Example5_11
{
    public static void main(String args[])
    {
        A a=new A();
        B b=new B();
        System.out.println(a.MAX);
        System.out.println(a.f(5)+" "+a.g(1,2));
        System.out.println(b.MAX);
        System.out.println(b.f(5)+" "+b.g(1,2));
    }
}
```

```
100
25 3
100
125 2
```

## 5.11 接口

- 2.接口与多态
- 为什么要用接口？
  - 假如轿车、拖拉机、客车都是机动车的子类，其中，机动车是一个抽象类。
  - 如果机动车中有3个abstract方法：“刹车”、“收取费用”、“调节温度”，那么所有的子类都要实现这3个方法，产生各自的收费等行为。这显然不符合人们的思维方法，因为拖拉机可能不需要有“收取费用”或“调节温度”的功能，合理的处理就是去掉机动车的“收取费用”和“调节温度”这两个方法。

## 5.11 接口

- **如果允许多继承**，轿车类想具有“调节温度”的功能，轿车类可以是机动车的子类，同时也是另外一个具有“调节温度”功能的类的子类。多继承有可能增加子类的负担，因为轿车可能从它的多个父类继承了一些并不需要的功能。
- **Java不支持多继承**，即一个类只能有一个父类。**单继承使得程序更加容易维护和健壮**，多继承使得编程更加灵活，但却增加了子类的负担，使用不当会引起混乱。
- 为了使程序容易维护和健壮，**且不失灵活性**，Java使用了**接口**，一个类可以**实现**多个接口，接口可以增加很多类都需要实现的功能，不同的类可以实现相同的接口，同一个类也可以实现多个接口。
- 接口的思想在于它可以增加很多类都需要实现的功能。

# Outline

- 5.1 子类与父类
- 5.2 子类的继承性
- 5.3 子类对象的构造过程
- 5.4 成员变量隐藏和方法重写
- 5.5 关键字super
- 5.6 final类与final方法
- 5.7 对象的上转型对象
- 5.8 继承与多态
- 5.9 abstract类
- 5.10 面向抽象
- 5.11 接口
- 5.12 接口回调
- 5.13 面向接口
- 5.14 抽象类与接口的比较
- 5.15 内部类
- 5.16 匿名类
- 5.17 异常类
- 5.18 泛型类



## 5.12 接口回调

- 在讲述继承与多态时，我们通过子类对象的上转型体现了继承的多态性，即把子类创建的对象引用放到一个父类的对象中时，得到该对象的一个上转型对象，那么这个上转型对象在调用方法时就可能具有多种形态，不同对象的上转型对象调用同一方法可能产生不同的行为。
- 1.接口回调
- 接口回调是多态的另一种体现。
- 接口回调：把实现某一接口的类创建的对象引用赋给该接口声明的接口变量，那么该接口变量就可以调用被类实现的接口中的方法，当接口变量调用被类实现的接口中的方法时，就是通知相应的对象调用接口的方法，这一过程称作对象的接口回调。
- 不同的类在实现同一接口时，可能具有不同的功能体现，即接口的方法体不必相同，因此，接口回调可能产生不同的行为。

## 5.12 接口回调

- 【例子】

```
public class Example5_12
{
    public static void main(String args[])
    {
        ElectricalAppliances ea;
        → ea=new TV();
        ea.showTradeMark();
        → ea=new PC();
        ea.showTradeMark();
    }
}
```

```
interface ElectricalAppliances
{
    void showTradeMark();
}
```

```
class PC implements ElectricalAppliances
{
    public void showTradeMark()
    {
        System.out.println("PC");
    }
}
```

```
class TV implements ElectricalAppliances
{
    public void showTradeMark()
    {
        System.out.println("TV");
    }
}
```

TV  
PC

## 5.12 接口回调

- 2.接口做参数
- 当一个方法的参数是一个接口类型时（i.e., 接口作为一个data type），如果一个类实现了该接口，那么，就可以把该类的实例的引用传值给该参数，参数可以回调类实现的接口中的方法。

## 5.12 接口回调

- 【例子】

```
interface Show
{
    void show();
}
```

```
class A implements Show
{
    public void show()
    {
        System.out.println("I Love This Game");
    }
}
```

```
class B implements Show
{
    public void show()
    {
        System.out.println("I Love NBA");
    }
}
```

```
public class Example5_13
{
    public static void main(String args[])
    {
        C c = new C();
        c.f(new A());
        c.f(new B());
    }
}
```

```
class C
{
    public void f(Show s)
    {
        s.show();
    }
}
```

```
I love This Game
I love NBA
```

# Outline

- 5.1 子类与父类
- 5.2 子类的继承性
- 5.3 子类对象的构造过程
- 5.4 成员变量隐藏和方法重写
- 5.5 关键字super
- 5.6 final类与final方法
- 5.7 对象的上转型对象
- 5.8 继承与多态
- 5.9 abstract类
- 5.10 面向抽象
- 5.11 接口
- 5.12 接口回调
- **5.13 面向接口**
- 5.14 抽象类与接口的比较
- 5.15 内部类
- 5.16 匿名类
- 5.17 异常类
- 5.18 泛型类

## 5.12 面向回调

- 面向接口也可以体现程序设计的“开-闭”原理（Open-Closed Principle），即对扩展开放，对修改关闭。将经常需要变化的细节分割出来，作为接口中的abstract方法，然后面向接口来设计类。

## 5.12 面向接口

- 【例子1/3】

```
public interface Geometry
{
    public abstract double getArea();
}
```

```
public class Circle implements Geometry
{
    double r;
    Circle(double r)
    {
        this.r=r;
    }
    public double getArea()
    {
        return(3.14*r*r);
    }
}
```

```
public class Lader implements Geometry
{
    double a,b,h;
    Lader(double a,double b,double h)
    {
        this.a=a; this.b=b; this.h=h;
    }
    public double getArea()
    {
        return((1/2.0)*(a+b)*h);
    }
}
```

## 5.12 面向接口

- 【例子2/3】

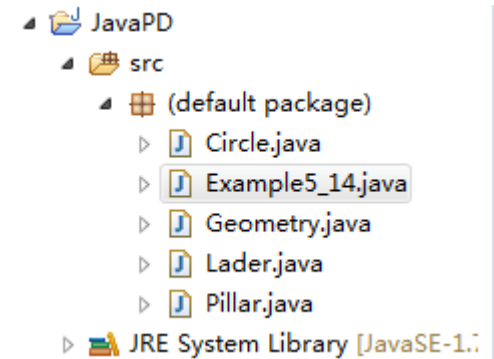
```
public class Pillar
{
    Geometry bottom;
    double height;
    Pillar (Geometry bottom, double height)
    {
        this.bottom=bottom;
        this.height=height;
    }

    void changeBottom(Geometry bottom)
    {
        this.bottom=bottom;
    }
    public double getVolume()
    {
        return bottom.getArea()*height;
    }
}
```



## 5.12 面向接口

```
Lader area: 1.0  
Pillar volume: 1.0  
Circle area: 3.14  
Pillar volume: 3.14
```



- 【例子3/3】

```
public class Example5_14  
{  
    public static void main(String args[])  
    {  
        Pillar pillar;  
        Geometry geometry;  
  
        geometry=new Lader(1,1,1);  
        System.out.println("Lader area: " + geometry.getArea());  
  
        pillar =new Pillar (geometry,1);  
        System.out.println("Pillar volume: " + pillar.getVolume());  
  
        geometry=new Circle(1);  
        System.out.println("Circle area: " + geometry.getArea());  
  
        pillar.changeBottom(geometry);  
        System.out.println("Pillar volume: " + pillar.getVolume());  
    }  
}
```

# Outline

- 5.1 子类与父类
- 5.2 子类的继承性
- 5.3 子类对象的构造过程
- 5.4 成员变量隐藏和方法重写
- 5.5 关键字super
- 5.6 final类与final方法
- 5.7 对象的上转型对象
- 5.8 继承与多态
- 5.9 abstract类
- 5.10 面向抽象
- 5.11 接口
- 5.12 接口回调
- 5.13 面向接口
- 5.14 抽象类与接口的比较
- 5.15 内部类
- 5.16 匿名类
- 5.17 异常类
- 5.18 泛型类

## 5.14 抽象类与接口的比较

- 抽象类中可以有abstract方法、非abstract方法；接口中只可以有abstract方法【注：Java 8之后可以有静态方法】
- 抽象类中可以有常量、变量；接口中只可以有常量

interface更纯粹

	<i>Variables</i>	<i>Constructors</i>	<i>Methods</i>
Abstract class	No restrictions.	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	No restrictions.
Interface	All variables must be <b>public static final</b> .	No constructors. An interface cannot be instantiated using the new operator.	All methods must be public abstract instance methods

Java 8之后可以有静态方法

## 5.14 抽象类与接口的比较

- 抽象类和接口：让设计忽略细节，将重心放在整个系统的设计上。
- 如果某个问题需要使用**继承**才能更好的解决，如子类除了需要实现父类的抽象方法，还需要从父类继承一些变量或继承一些重要的非抽象方法，可以考虑用**抽象类**。
- 如果某个问题**不需要继承**，只是需要给出某些重要的抽象方法的实现细节，就可以考虑使用**接口**。

关键看是否需要**继承**

## 5.14 抽象类与接口的比较

- A superclass defines **common behavior for related subclasses**.
- An interface can be used to define **common behavior for classes (including unrelated classes)**.

### Design Guide

- In general, **a strong is-a relationship** that clearly describes a parent-child relationship should be modeled using classes.
- **A weak is-a relationship**, also known as **an is-kind-of relationship**, indicates that an object possesses a certain property. A weak is-a relationship can be modeled using interfaces.

## 5.14 抽象类与接口的比较

- Interfaces are more **flexible** than abstract classes, because a subclass can extend only one superclass but can implement any number of interfaces.
- However, interfaces cannot contain concrete methods.
- The virtues of interfaces and abstract classes can be combined by creating **an interface with an abstract class that implements it**. Then you can use the interface or the abstract class, whichever is convenient.

# Outline

- 5.1 子类与父类
- 5.2 子类的继承性
- 5.3 子类对象的构造过程
- 5.4 成员变量隐藏和方法重写
- 5.5 关键字super
- 5.6 final类与final方法
- 5.7 对象的上转型对象
- 5.8 继承与多态
- 5.9 abstract类
- 5.10 面向抽象
- 5.11 接口
- 5.12 接口回调
- 5.13 面向接口
- 5.14 抽象类与接口的比较
- 5.15 内部类
- 5.16 匿名类
- 5.17 异常类
- 5.18 泛型类

## 5.15 内部类

- 类可以有两种重要成员：成员变量和方法
- 类还可以有一种成员：**内部类**
- Java支持在一个类中声明另一个类，这样的类称作**内部类**，而包含内部类的类称为内部类的**外嵌类**
- **外嵌类**把**内部类**看作是自己的成员
- **外嵌类**的成员变量在**内部类**中仍然有效，**内部类**中的方法可以调用**外嵌类**中的方法
- **内部类**的类体中不可以声明静态变量（类变量）和静态方法（类方法）
  - 如果这个**内部类**被声明为**static**，则可以有静态变量和静态方法
- **外嵌类**可以用**内部类**声明对象，作为外嵌类的成员




## 5.15 内部类

- 【例子1/2】

```
class A
{
    int x=10,y=10;
    A2 a2; // 外嵌类用内部类声明对象，作为外嵌类的成员
    A()
    {
        a2 = new A2();
    }
    void f()
    {
        System.out.println("A");
        a2.speak();
    }

    // inner class
    class A2
    {
        int z;
        void speak()
        {
            System.out.println("A2");
        }
        void g()
        {
            z=x+y; // 外嵌类的成员变量在内部类中仍然有效
            System.out.println(z);
            f(); // 内部类中的方法可以调用外嵌类中的方法
        }
    }
}
```



## 5.15 内部类

- 【例子2/2】

```
public class Example5_15
{
    public static void main(String args[])
    {
        A a = new A();
        a.f();
        a.a2.g();
    }
}
```

```
A
A2
20
A
A2
```

# Outline

- 5.1 子类与父类
- 5.2 子类的继承性
- 5.3 子类对象的构造过程
- 5.4 成员变量隐藏和方法重写
- 5.5 关键字super
- 5.6 final类与final方法
- 5.7 对象的上转型对象
- 5.8 继承与多态
- 5.9 abstract类
- 5.10 面向抽象
- 5.11 接口
- 5.12 接口回调
- 5.13 面向接口
- 5.14 抽象类与接口的比较
- 5.15 内部类
- 5.16 匿名类
- 5.17 异常类
- 5.18 泛型类

## 5.16 匿名类

- 1. 和类有关的匿名类
- 当使用类创建对象时，程序允许我们把类体与对象的创建组合在一起，此**类体**被认为是该类的一个**子类**去掉类声明后的类体，称作**匿名类**。
- **匿名类就是一个子类**，由于无名可用，所以不可能用匿名类声明对象，但却可以直接用匿名类创建一个对象。

## 5.16 匿名类

- 匿名类可以继承类的方法也可以重写类的方法。
- 我们使用匿名类时，必然是在某个类中直接用匿名类创建对象，**因此匿名类一定是内部类**
  - 匿名类可以访问外嵌类中的成员变量和方法
  - 匿名类不可以声明静态成员变量和静态方法
- 匿名类的**主要用途**就是**向方法的参数传值**。

## 5.16 匿名类

- 【例子】

```
abstract class Student
{
    abstract void speak();
}
class Teacher
{
    void look(Student stu)
    {
        stu.speak();
    }
}
public class Example5_16
{
    public static void main(String args[])
    {
        Teacher zhang = new Teacher();
        zhang.look(
            new Student()
            {
                void speak()
                {
                    System.out.println("这是匿名类中的方法");
                }
            }
        );
    }
}
```

匿名类的类体，  
即Student的子类的类体

## 5.16 匿名类

- 2. 和接口有关的匿名类
- 假设`Comparable`是一个接口，那么，Java允许直接用接口名和一个类体创建一个匿名对象，此类体被认为是实现了`Comparable`接口的类去掉类声明后的类体，称作匿名类。
- 如果某个方法的参数是接口类型，那么我们可以使用接口名和类体组合创建一个匿名对象传递给方法的参数，类体必须要实现接口中的全部方法。

## 5.16 匿名类

- 【例子】

```
interface Show
{
    public void show();
}
class A
{
    void f(Show s)
    {
        s.show();
    }
}

public class Example5_17
{
    public static void main(String args[])
    {
        A a=new A();
        a.f(
            new Show(){
                public void show()
                {
                    System.out.println("这是实现了接口的匿名类");
                }
            });
    }
}
```

匿名类的类体，  
即实现接口Show的类的类体



# Outline

- 5.1 子类与父类
- 5.2 子类的继承性
- 5.3 子类对象的构造过程
- 5.4 成员变量隐藏和方法重写
- 5.5 关键字super
- 5.6 final类与final方法
- 5.7 对象的上转型对象
- 5.8 继承与多态
- 5.9 abstract类
- 5.10 面向抽象
- 5.11 接口
- 5.12 接口回调
- 5.13 面向接口
- 5.14 抽象类与接口的比较
- 5.15 内部类
- 5.16 匿名类
- 5.17 异常类
- 5.18 泛型类

## 5.17 异常类

- 所谓异常就是程序运行时可能出现一些错误，比如试图打开一个根本不存在的文件等，异常处理将会改变程序的控制流程，**让程序有机会对错误作出处理**。
- 当程序运行出现异常时，Java运行环境就用异常类**Exception**的相应子类创建一个异常对象，并等待处理。
- Java使用try-catch语句来处理异常，将可能出现的异常操作放在try-catch语句的try部分，当try部分中的某个语句发生异常后，try部分将立刻结束执行，而转向执行相应的catch部分。

## 5.17 异常类

- 所以，程序可以将发生异常后的处理放在catch部分。

- 1. try-catch语句

```
try
{
    包含可能发生异常的语句
}
catch(ExceptionSubClass1 e)
{

}
catch(ExceptionSubClass2 e)
{

}
```

- 各个catch参数中的异常类都是Exception的某个子类，表明try部分可能发生的异常，**这些子类之间不能有父子关系**，否则保留一个含有父类参数的catch即可。

## 5.17 异常类

- 【例子】

```
public class Example5_18
{
    public static void main(String args[])
    {
        int n=0,m=0,t=0;
        try{
            t=3;
            m=Integer.parseInt("2");
            n=Integer.parseInt("1s"); // Exception
            System.out.println("我没有机会输出");
        }
        catch(Exception e)
        {
            System.out.println("Exception");
            n=1;
        }

        System.out.println("n=" + n + ",m=" + m + ",t=" + t);
    }
}
```

Exception  
n=1,m=2,t=3

## 5.17 异常类

- 2. 自定义异常类
- 我们也可以继承Exception类，定义自己的异常类，然后规定哪些方法产生这样的异常。
- 一个方法在声明时可以使用**throw**关键字声明抛出所要产生的若干个异常，并在该方法的方法体中具体给出产生异常的操作，即用**相应的异常类创建对象**，这将导致该方法结束执行并抛出所创建的异常对象。
- 程序必须在try-catch语句块中调用抛出异常的方法。

## 5.17 异常类

- 【例子】

```
class MyException extends Exception
{
    String message;
    MyException(int n)
    {
        message = n + ": not a positive number";
    }
    public String getMessage()
    {
        return message;
    }
}

class A
{
    public void f(int n) throws MyException
    {
        if(n<0)
        {
            MyException ex = new MyException(n);
            → throw(ex); // 抛出异常，结束方法f的执行
        }
        double number = Math.sqrt(n);
        System.out.println("square root of " + n + ": " + number);
    }
}
```

## 5.17 异常类

- 【例子】

```
public class Example5_19
{
    public static void main(String args[])
    {
        A a=new A();
        try{
            a.f(28);
            a.f(-8);
        }
        catch(MyException e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

```
square root of 28: 5.291502622129181
-8: not a positive number
```

# Outline

- 5.1 子类与父类
- 5.2 子类的继承性
- 5.3 子类对象的构造过程
- 5.4 成员变量隐藏和方法重写
- 5.5 关键字super
- 5.6 final类与final方法
- 5.7 对象的上转型对象
- 5.8 继承与多态
- 5.9 abstract类
- 5.10 面向抽象
- 5.11 接口
- 5.12 接口回调
- 5.13 面向接口
- 5.14 抽象类与接口的比较
- 5.15 内部类
- 5.16 匿名类
- 5.17 异常类
- 5.18 泛型类



## 5.18 泛型类

- 泛型（**generics**）是Sun公司在SDK1.5中推出的，其主要目的是可以建立具有**类型安全**的集合框架，如链表、散列映射等数据结构。
- 1. 泛型类声明
- 可以使用“**class 名称<泛型列表>**”声明一个类，为了和普通的类有所区别，这样声明的类称作**泛型类**，如：

```
class A<E>;
```

- 其中**A**是泛型类的名称，**E**是其中的泛型，也就是说我们并没有指定**E**是何种类型的数据，它**可以是任何对象或接口**，但**不能是基本类型数据**。

## 5.18 泛型类

- 泛型类的**类体**和普通类的类体完全类似，由成员变量和方法构成

```
class Chorus<E,F>
{
    void makeChorus(E person, F instrument)
    {
        person.toString();
        instrument.toString();
    }
}
```

## 5.18 泛型类

- 2. 使用泛型类声明对象
- 使用泛型类**声明变量、创建对象**时，必须要指定类中使用的泛型的**实际类型**。

```
Chorus<Student, Button> model;  
model = new Chorus<Student, Button> ();
```

## 5.18 泛型类

- 【例子1/2】

```
class Chorus<E,F>
{
    void makeChorus(E person, F instrument)
    {
        person.toString();
        instrument.toString();
    }
}

class Singer
{
    public String toString()
    {
        System.out.println("好一朵美丽的茉莉花");
        return "";
    }
}

class MusicalInstrument
{
    public String toString()
    {
        System.out.println("/3 35 6116/5 56 5-|");
        return "";
    }
}
```

## 5.18 泛型类

- 【例子2/2】

```
public class Example5_20
{
    public static void main(String args[])
    {
        Chorus<Singer, MusicalInstrument> model = new Chorus<Singer, MusicalInstrument>();
        Singer singer = new Singer();
        MusicalInstrument piano = new MusicalInstrument();
        model.makeChorus(singer, piano);
    }
}
```

好一朵美丽的茉莉花

| 3 35 6116 | 5 56 5- |

## 5.18 泛型类

- Java中的泛型类和**C++的类模板**有很大的不同，在上述例子中，泛型类中的泛型数据person和instrument**只能调用Object类中的方法**，因此Singer和MusicalInstrument两个类都重写了Object类的toString()方法。
- 下面我们再看一个例子，我们声明了一个**泛型类Cone**，一个Cone对象计算体积时，只关心它的底是否能计算面积，并不关心底的类型。

## 5.18 泛型类

- 【例子1/3】

```
class Cone<E>
{
    double height;
    E bottom;
    public Cone(E b)
    {
        bottom = b;
    }
    public void computeVolume()
    {
        String s = bottom.toString();
        double area = Double.parseDouble(s);
        System.out.println("Volume:" + 1.0/3.0*area*height);
    }
}
```

## 5.18 泛型类

- 【例子2/3】

```
class Rectangle
{
    double sideA,sideB,area;
    Rectangle(double a,double b)
    {
        sideA=a;
        sideB=b;
    }
    public String toString()
    {
        area = sideA*sideB;
        return ""+area;
    }
}
```

```
class Circle
{
    double area,radius;
    Circle(double r)
    {
        radius = r;
    }
    public String toString()
    {
        area = radius*radius*Math.PI;
        return "" + area;
    }
}
```



## 5.18 泛型类

- 【例子3/3】

```
public class Example5_21
{
    public static void main(String args[])
    {
        Circle circle = new Circle(1);
        Cone<Circle> coneCircle = new Cone<Circle>(circle);
        coneCircle.height=1;
        coneCircle.computeVolume();

        Rectangle rect = new Rectangle(1,1);
        Cone<Rectangle> coneRectangle = new Cone<Rectangle>(rect);
        coneRectangle.height = 1;
        coneRectangle.computeVolume();
    }
}
```

```
Volume:1.0471975511965976
Volume:0.3333333333333333
```

## 5.18 泛型类

- 3. 泛型接口
- 可以使用“interface 名称<泛型列表>”声明一个接口，这样声明的接口称作**泛型接口**

```
interface Computer<E>;
```

## 5.18 泛型类

泛型接口

- 【例子1/2】

```
interface Computer<E,F>
{
    void makeChorus(E x, F y);
}
```

```
class Chorus<E,F> implements Computer<E,F>
{
    public void makeChorus(E x, F y)
    {
        x.toString();
        y.toString();
    }
}
```

```
class MusicalInstrument
{
    public String toString()
    {
        System.out.println("/5 6 3-/5 17 56/");
        return "";
    }
}

class Singer
{
    public String toString()
    {
        System.out.println("美丽的草原,我可爱的家乡");
        return "";
    }
}
```

## 5.18 泛型类

- 【例子2/2】

```
public class Example5_22
{
    public static void main(String args[ ])
    {
        Chorus<Singer, MusicalInstrument> model = new Chorus<Singer, MusicalInstrument>();
        Singer singer = new Singer();
        MusicalInstrument piano = new MusicalInstrument();
        model.makeChorus(singer, piano);
    }
}
```

美丽的草原,我可爱的家乡

| 5 6 3 - | 5 17 56 |

## 5.18 泛型类

- Java泛型的主要目的是可以建立具有**类型安全**的数据结构，如链表（LinkedList）、散列映射（HashMap）等数据结构。
- SDK1.5是支持泛型的编译器，它将**运行时类型检查提前到编译时执行**，使代码更安全。

## 5.18 泛型类

- Generics is the capability **to parameterize types**. With it you can define a class or a method with **generic types** that can be replaced with **concrete types** by the compiler.
- The key benefit of generics is to **enable errors to be detected at compile time** rather than at runtime.
- A generic class or method **permits you to specify allowable types of objects** that the class or method can work with. If you attempt to use the class or method with an incompatible object, the compiler can detect the error.

# 小节

- 5.1 子类与父类
- 5.2 子类的继承性
- 5.3 子类对象的构造过程
- 5.4 成员变量隐藏和方法重写
- 5.5 关键字super
- 5.6 final类与final方法
- 5.7 对象的上转型对象
- 5.8 继承与多态
- 5.9 abstract类
- 5.10 面向抽象
- 5.11 接口
- 5.12 接口回调
- 5.13 面向接口
- 5.14 抽象类与接口的比较
- 5.15 内部类
- 5.16 匿名类
- 5.17 异常类
- 5.18 泛型类