

讲义制作：海风
参考《Java疯狂讲义（第3版）》

第4章 面向对象编程 下

韩 慧

hanhuie@126.com

A solid green horizontal bar at the bottom of the slide.

本章导读

- 4.1 继承 Inheritance
- 4.2 抽象类和抽象方法 Abstract Class & Abstract Method
- 4.3 内部类 Inner Class
- 4.4 接口 Interface
- 4.5 Lambda表达式 λ Expression （选讲）
- 4.6 接口与抽象类 Interface & Abstract Class
- 4.7 大话泛型 Generic Programming
- 4.8 枚举类 Enumeration Class
- 4.9 作业及延伸

本章导读

✓ 4.1 继承 Inheritance

□ 4.2 抽象类和抽象方法 Abstract Class & Abstract Method

□ 4.3 内部类 Inner Class

□ 4.4 接口 Interface

□ 4.5 Lambda表达式 λ Expression

□ 4.6 接口与抽象类 Interface & Abstract Class

□ 4.7 大话泛型 Generic Programming

□ 4.8 枚举类 Enumeration Class

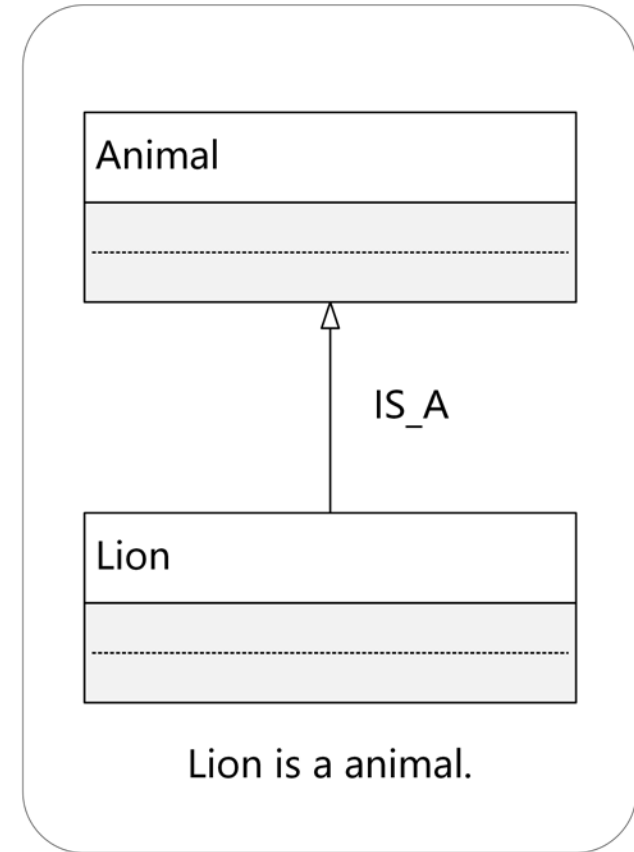
□ 4.9 作业及延伸

4.1 继承

- 4.1.1 继承的基本概念
- 4.1.2 成员变量的隐藏
- 4.1.3 父类方法的重写
- 4.1.4 关键字super
- 4.1.5 再话类的初始化顺序
- 4.1.6 关键字final
- 4.1.7 用继承的眼光看Object类
- 4.1.8 多态

4.1.1 继承的基本概念_Inheritance

- 继承是对现实生活中的“分类”概念的一种模拟。
- 狮子拥有动物的一切基本特性，但同时又拥有自己独特的特性，这就是“继承”关系的重要特性：通常简称为“IS_A”关系。



4.1.1 继承的基本概念

- 类之间的继承关系是一种由已经存在的类创建新类的机制，可以有效地实现代码的复用。
- 可以先定义一个共有属性的一般类（称之为父类parent class、超类super class），在此基础上定义新的类（子类child class）。
- 子类自动拥有父类父类声明为public和protected的成员（子类不继承父类的构造方法），也可以定义自己独特的属性和方法。
- 父类可以是不可变类（即final类）以外的任意类，既可以是Java类库中的类，也可以是自定义的类。
- Java不支持多重继承，即子类只能有一个父类。

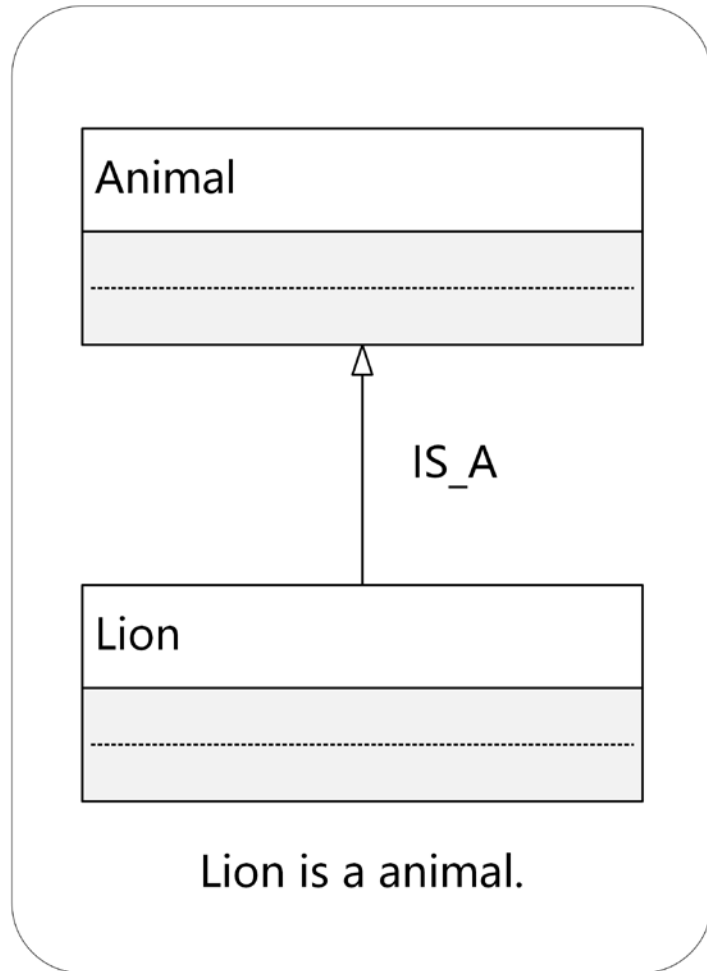
4.1.1 继承的基本概念

- 在类的声明中，使用关键字`extends`来声明一个类是另一个类的子类，语法格式如下：

```
class 子类名 extends 父类名 {  
    子类的类体  
}
```

- 从子类的角度来看，子类扩展（`extends`）了父类；但从父类的角度来看，父类派生（`derive`）出了子类。也就是说，扩展和派生所描述的是一个动作，只是观察角度不同。

4.1.1 继承的基本概念



```
class Animal {  
    ...  
}  
class Lion extends Animal {  
    ...  
}
```


4.1.1 继承的基本概念

如果一个Java类时并未显式指定这个类的直接父类，即没有使用关键字extends，则这个类被系统默认为是java.lang.Object类的子类。

因此，**java.lang.Object类是所有类的顶层基类**，要么是其直接父类，要么是其间接父类。



Object

- Object()
- registerNatives() : void
- getClass() : Class<?>
- hashCode() : int
- equals(Object) : boolean
- clone() : Object
- toString() : String
- notify() : void
- notifyAll() : void
- wait(long) : void
- wait(long, int) : void
- wait() : void
- finalize() : void

4.1.2 成员变量的隐藏

- 子类中定义的新的变量如果和继承过来的变量（public 和 protected）名称相同，则会隐藏继承过来的成员变量

```
class X {  
    double y=11.456789;  
    public void f() {  
        y=y+1;  
        System.out.printf("y是double型的变量， y=%f\n",y);  
    }  
}  
  
class Y extends X {  
    int y=0;  
    public void g() {  
        y=y+100;  
        System.out.println("y是int型的变量， y="+y);  
    }  
}  
  
public class PropertyHide {  
    public static void main(String args[]) {  
        Y b = new Y();  
        b.y = 200;  
        b.g();  
        b.f();  
    }  
}
```

一般不推荐在定义子类时，隐藏父类的变量

PropertyHide

4.1.3 父类方法的重写_子类与父类方法间的关系

- 由于Java并未对子类方法的命名做过多的限制，因此，子类与父类各自定义的方法之间，可以出现以下3中情况：
 - 扩充（extends）：子类定义的方法父类没有同名
 - **覆盖/重写（override）：子类父类定义了完全一样的方法**
 - 重载（overload）：子类有父类的同名方法，但两者的参数类型或参数数目不一样

4.1.3 父类方法的重写

- 方法的重写也是多态的一种表现。
- 如果父类中的方法被子类重写，则子类的对象调用这个方法时，实际运行的是子类中改写之后的方法，父类中的相应方法被隐藏了起来。
- 方法的重写要遵循“**两同两小一大**”规则，“两同”即方法名相同；“两小”指子类方法返回值类型应比父类方法返回值更小或相等，子类方法声明抛出的异常类应比父类方法声明抛出的异常类更小或相等；“一大”指的是子类方法的访问权限应比父类方法的访问权限更大或相等。

4.1.3 父类方法的重写

- 右图所示的程序，将鸟类进行抽象，定义了一个Bird类，并派生了一个子类“鸵鸟类”。该场景下，子类需要重写父类的方法，毕竟鸵鸟不会飞。

```
class Bird {  
    // Bird类的fly()方法  
    public void fly() {  
        System.out.println("我在天空里自由自在地飞翔...");  
    }  
}  
  
class Ostrich extends Bird {  
    // 重写Bird类的fly()方法  
    public void fly() {  
        System.out.println("我只能在地上奔跑...");  
    }  
}  
  
public class OverrideTest {  
    public static void main(String[] args){  
        // 创建Ostrich对象  
        Ostrich os = new Ostrich();  
        // 执行Ostrich对象的fly()方法，将输出"我只能在地上奔跑..."  
        os.fly();  
    }  
}
```

OverrideTest

4.1.3 父类方法的重写

- 子类中的重写方法与父类的被重写方法要么都是类方法，要么都是实例方法，不能一个是类方法，一个是实例方法。如下的代码会引发编译错误：

```
class BaseClass {  
    public static void test() {...}  
}  
class SubClass extends BaseClass {  
    public void test() {...}  
}
```

4.1.3 父类方法的重写

- 子类中的重写方法与父类的被重写方法要么都是类方法，要么都是实例方法，不能一个是类方法，一个是实例方法。如下的代码会引发编译错误：

```
class BaseClass {  
    public static void test() {...}  
}  
class SubClass extends BaseClass {  
    public void test() {...}  
}
```

- 如果父类方法具有private访问权限，则该方法对其子类是隐藏的，因此其子类无法访问该对象，也就是无法重写该方法。如果子类中定义了一个与父类private方法具有相同的方法名、相同的形参列表、相同返回值类型的方法，依然不是方法重写，只是在子类中重新定义了一个新方法。因此，如下代码并不会报编译错误。

```
class BaseClass {  
    private void test() {...}  
}  
class SubClass extends BaseClass {  
    public static void test() {...}  
}
```

4.1.4 关键字super

- 关键字**super**代表当前对象的父类
- super有两种用法
 - 子类使用super调用父类的构造方法
 - 子类使用super操作父类中被子类隐藏的成员变量和方法

4.1.4 关键字super

子类使用super调用父类的构造方法

- 子类不继承父类的构造方法
- 子类中有些初始化工作需要调用父类的构造方法帮助实现，需要用super关键字
- 子类构造方法中调用父类构造方法的语法为

super(参数列表)

- 子类构造方法中使用super调用父类构造方法时，必须把调用语句放在最开始

4.1.4 关键字super

子类使用super调用父类的构造方法代码示例

```
class Base {  
    public double size;  
    public String name;  
    public Base(double size , String name) {  
        this.size = size;  
        this.name = name;  
    }  
}  
  
class Sub extends Base {  
    public String color;  
    public Sub(double size , String name , String color) {  
        // 通过super调用来调用父类构造器的初始化过程  
        super(size , name);  
        this.color = color;  
    }  
}  
  
public class SuperConstructor {  
    public static void main(String[] args) {  
        Sub s = new Sub(5.6 , "测试对象" , "红色");  
        // 输出Sub对象的三个实例变量  
        System.out.println(s.size + "-" + s.name  
            + "-" + s.color);  
    }  
}
```

●思考：这种语法形式联想到了什么？

SuperConstructor

4.1.4 关键字super

子类使用super调用父类的构造方法代码示例

```
class Base {
    public double size;
    public String name;
    public Base(double size , String name) {
        this.size = size;
        this.name = name;
    }
}

class Sub extends Base {
    public String color;
    public Sub(double size , String name , String color) {
        // 通过super调用来调用父类构造器的初始化过程
        super(size , name);
        this.color = color;
    }
}

public class SuperConstructor {
    public static void main(String[] args) {
        Sub s = new Sub(5.6 , "测试对象" , "红色");
        // 输出Sub对象的三个实例变量
        System.out.println(s.size + "-" + s.name
            + "-" + s.color);
    }
}
```

- 思考：这种语法形式联想到了什么？
- super调用与this调用非常相像，区别在于super调用的是其父类的构造器，而this调用的是同一个类中重载的构造器。
- 思考：this和super关键字可以同时出现在一个构造器中吗？

4.1.4 关键字super

子类使用super操作被隐藏的成员变量和方法

- 语法格式：

super.变量名

super.方法名(参数列表)

- 如果子类里没有包含和父类同名的成员变量，那么在子类实例方法中访问该成员变量时，则无须显式使用super或父类名作为调用者。如果在某个方法中访问名为a的成员变量，但没有显式指定调用者，则系统查找a的顺序为：
 1. 查找该方法中是否有名为a的局部变量
 2. 查找当前类中是否包含名为a的成员变量
 3. 查找a的直接父类中是否包含名为a的成员变量，依次上溯a的所有父类，直到java.lang.Object类，如果最终不能找到名为a的成员变量，则系统出现编译错误。
- 如果被覆盖的是类变量，在子类的方法中则可以通过父类名作为调用者来访问被覆盖的类变量。

4.1.4 关键字super

子类使用super操作被隐藏的成员变量

```
class BaseClass {
    public int a = 5;
}
class SubClass extends BaseClass {
    public int a = 7;
    public void accessOwner(){
        System.out.println(a);
    }
    public void accessBase(){
        // 通过super来访问从父类继承得到的被隐藏的实例变量a
        System.out.println(super.a);
    }
}
public class SuperProperty {
    public static void main(String[] args){
        SubClass sc = new SubClass();
        sc.accessOwner(); // 输出7
        sc.accessBase(); // 输出5
    }
}
```

SuperProperty

4.1.4 关键字super

子类使用super操作被隐藏的方法

```
package chapter4.SuperMethod;

class Bird {
    // Bird类的fly()方法
    public void fly() {
        System.out.println("我在天空里自由自在地飞翔...");
    }
}

class Ostrich extends chapter4.SuperMethod.Bird {
    // 重写Bird类的fly()方法
    public void fly() {
        System.out.println("我只能在地上奔跑...");
    }
    public void callOverriddenMethod() {
        // 在子类方法中通过super来显式调用父类被覆盖的方法。
        super.fly();
    }
}

public class SuperMethod {
    public static void main(String[] args){
        // 创建Ostrich对象
        chapter4.SuperMethod.Ostrich os = new chapter4.SuperMethod.Ostrich();
        // 执行Ostrich对象的fly()方法，将输出"我只能在地上奔跑..."
        os.fly();
        os.callOverriddenMethod();
    }
}
```

SuperMethod

4.1.5 再话类的初始化顺序

- 程序员可定义的类的初始化操作，主要包含在初始化块、构造器以及成员变量声明语句中。
- 根据上一章节所学，我们知道：
 - 类初始化块总是比普通初始化块先执行，初始化块在构造器之前执行。
 - 普通初始化块、声明实例变量指定的默认值都可认为是对对象的初始化代码，它们的执行顺序与源程序中的排列顺序相同。（类初始化块与类变量声明默认值同理）。
 - 一个类可以有多个初始化块，相同类型的初始化块之间有顺序：前面定义的初始化块先执行，后面定义的初始化块后执行。（本质为同类初始化块按源程序中定义顺序依次合并到构造器中）。

4.1.5 再话类的初始化顺序

引入继承后，类的初始化顺序

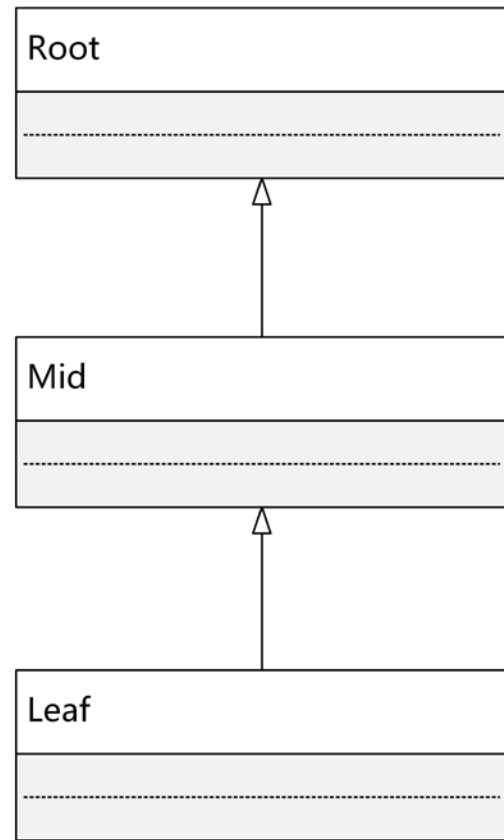
```
class Root {
    static{
        System.out.println("Root的静态初始化块");
    }
    {
        System.out.println("Root的普通初始化块");
    }
    public Root() {
        System.out.println("Root的无参数的构造器");
    }
}
```

```
class Leaf extends Mid {
    static{
        System.out.println("Leaf的静态初始化块");
    }
    {
        System.out.println("Leaf的普通初始化块");
    }
    public Leaf() {
        // 通过super调用父类中有一个字符串参数的构造器
        super("疯狂Java讲义");
        System.out.println("执行Leaf的构造器");
    }
}
```

```
class Mid extends Root {
    static{
        System.out.println("Mid的静态初始化块");
    }
    {
        System.out.println("Mid的普通初始化块");
    }
    public Mid() {
        System.out.println("Mid的无参数的构造器");
    }
    public Mid(String msg) {
        // 通过this调用同一类中重载的构造器
        this();
        System.out.println("Mid的带参数构造器，其参数值： "
            + msg);
    }
}

public class InitializationOrder{
    public static void main(String[] args)
    {
        new Leaf();
        new Leaf();
    }
}
```

InitializationOrder



继承结构

4.1.5 再话类的初始化顺序

引入继承后，类的初始化顺序

```
class Root {
    static{
        System.out.println("Root的静态初始化块");
    }
    {
        System.out.println("Root的普通初始化块");
    }
    public Root() {
        System.out.println("Root的无参数的构造器");
    }
}

class Leaf extends Mid {
    static{
        System.out.println("Leaf的静态初始化块");
    }
    {
        System.out.println("Leaf的普通初始化块");
    }
    public Leaf() {
        // 通过super调用父类中有一个字符串参数的构造器
        super("疯狂Java讲义");
        System.out.println("执行Leaf的构造器");
    }
}
```

```
class Mid extends Root {
    static{
        System.out.println("Mid的静态初始化块");
    }
    {
        System.out.println("Mid的普通初始化块");
    }
    public Mid() {
        System.out.println("Mid的无参数的构造器");
    }
    public Mid(String msg) {
        // 通过this调用同一类中重载的构造器
        this();
        System.out.println("Mid的带参数构造器，其参数值："
            + msg);
    }
}

public class InitializationOrder{
    public static void main(String[] args)
    {
        new Leaf();
        new Leaf();
    }
}
```

```
C:\Windows\system32\cmd.exe
E:\JAVA_TA>javac Test.java
E:\JAVA_TA>java Test
Root的静态初始化块
Mid的静态初始化块
Leaf的静态初始化块
Root的普通初始化块
Root的无参数的构造器
Mid的普通初始化块
Mid的无参数的构造器
Mid的带参数构造器，其参数值：疯狂Java讲义
Leaf的普通初始化块
执行Leaf的构造器
Root的普通初始化块
Root的无参数的构造器
Mid的普通初始化块
Mid的无参数的构造器
Mid的带参数构造器，其参数值：疯狂Java讲义
Leaf的普通初始化块
执行Leaf的构造器
E:\JAVA_TA>
```

4.1.6 关键字final_final类

- 回忆：使用final关键字修饰的成员变量，被视为常量，不允许更改。
- 以final关键字修饰的类，不能被继承，例如java.lang.Math类以及java.lang.String就是一个final类，他不可以有子类。

```
final class 类名 {  
    类体  
}
```

- 不可变的“类”有何用？
 - 可以方便和安全地用于多线程环境中，访问它们可以不用加锁，因而能提供较高的性能。

4.1.6 关键字final_final方法

- final 修饰的方法不可被重写。
- Java提供的Object类里就有一个final方法：getClass()，因为Java不希望任何子类重写这个方法，所以使用final把这个方法密封起来。但对于该类提供的toString()和equals()方法，都允许子类重写，因此没有用final修饰它们。
- final 修饰的方法仅仅是不能被重写，并不是不能被重载，因此左图所示程序会发生编译错误，而右图的程序完全没有问题。

```
class FinalMethodTest {  
    public final void test() {}  
}  
class Sub extends FinalMethodTest {  
    public void test() {}  
}
```

```
class FinalOverload {  
    public final void test() {}  
    public final void test(String str) {}  
}
```

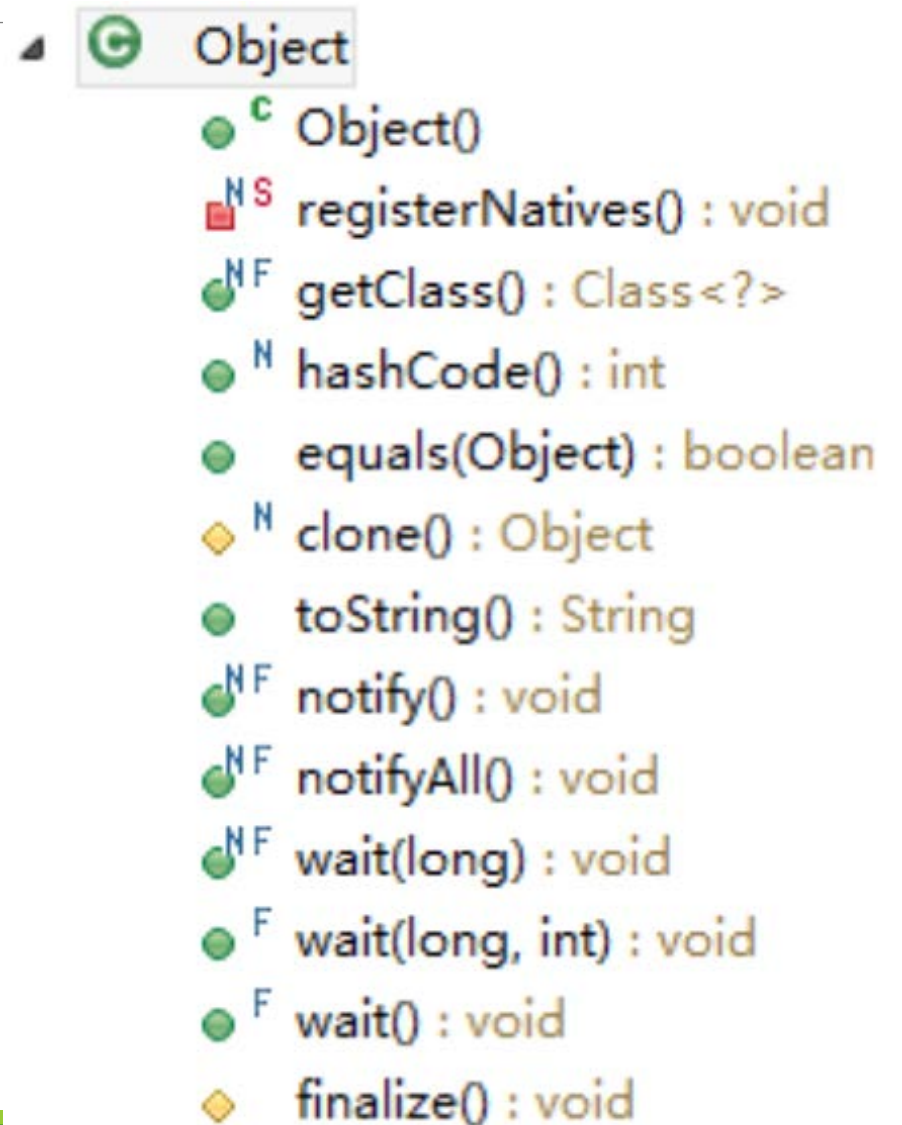
4.1.6 关键字final_思考

- 如下所示的代码是否会发生编译错误？请简要阐述。

```
public class PrivateFinalMethodTest
{
    private final void test(){}
}
class Sub extends PrivateFinalMethodTest
{
    public void test(){}
}
```

4.1.7 用继承的眼光看Object类

- 如果一个Java类时并未显式指定这个类的直接父类，即没有使用关键字 `extends`，则这个类被系统默认为是 `java.lang.Object` 类的子类。
- 因此， `java.lang.Object` 类是所有类的顶层基类，要么是其直接父类，要么是其间接父类。



4.1.7 用继承的眼光看Object类_思考

- 如右图所示的程序，定义一个类A，它没有任何成员，示例直接输出了这个类所创建的对象。

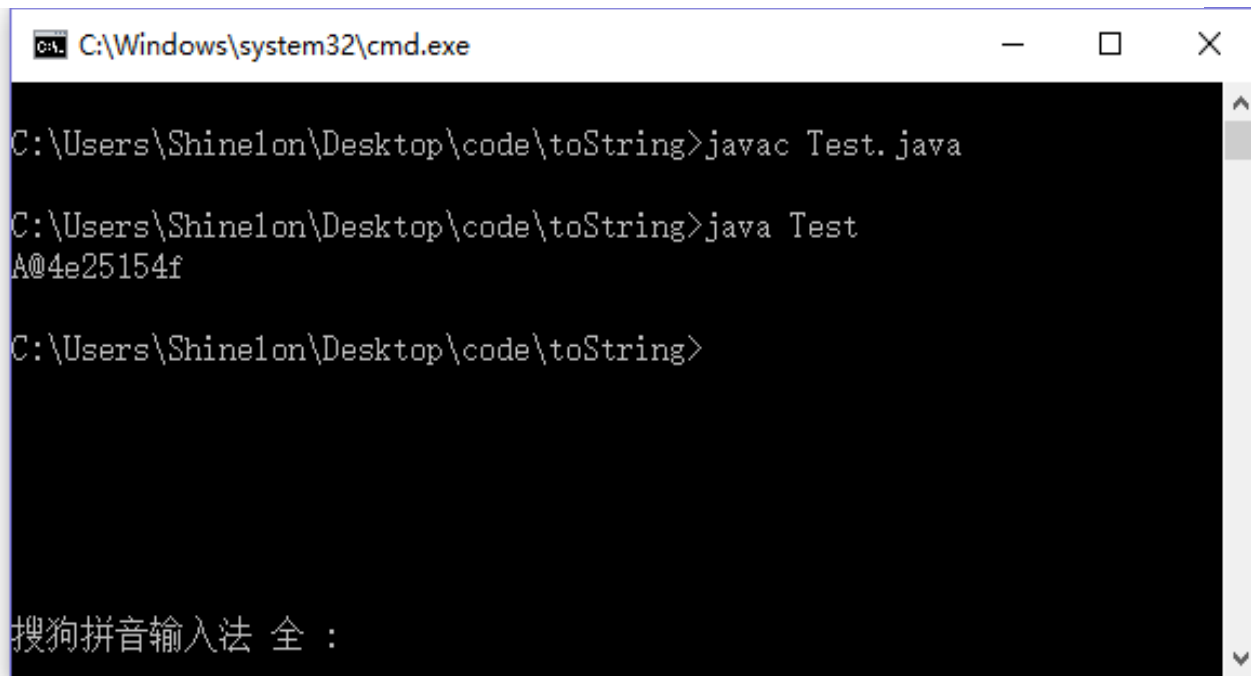
- 得到了一个神奇的运行结果：

A@4e25154f

- 请思考这句println语句，究竟调用了什么？

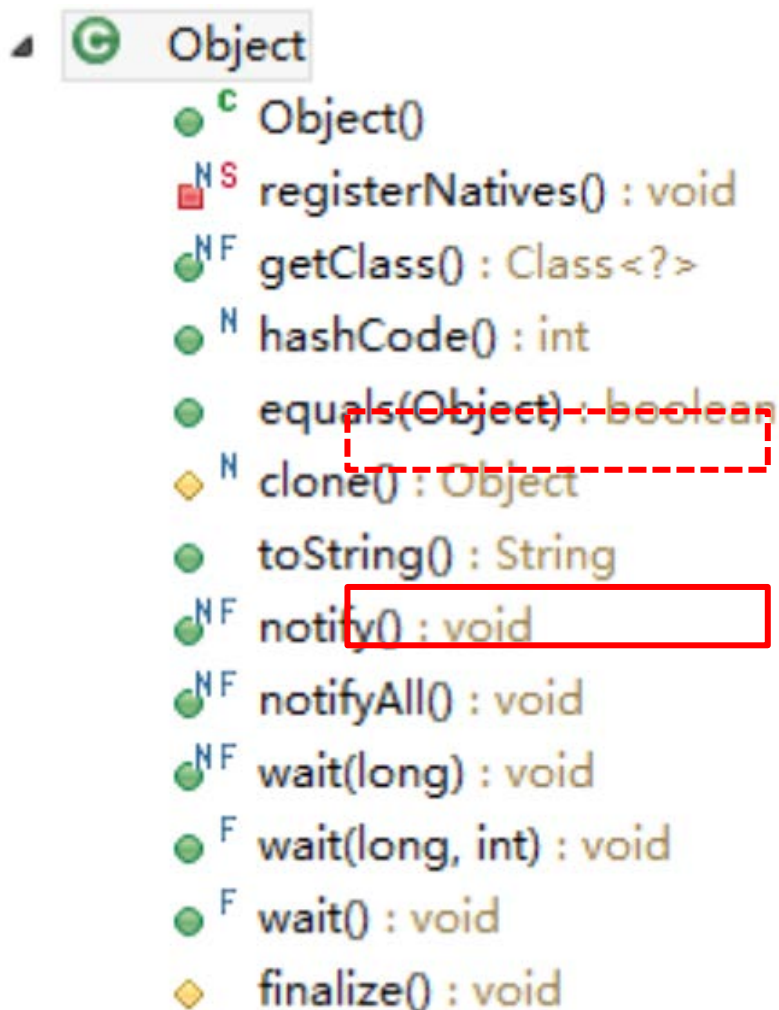
```
class A{  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        .....  
        System.out.println(new A());  
    }  
}
```



```
C:\Windows\system32\cmd.exe  
  
C:\Users\Shinelon\Desktop\code\toString>javac Test.java  
  
C:\Users\Shinelon\Desktop\code\toString>java Test  
A@4e25154f  
  
C:\Users\Shinelon\Desktop\code\toString>  
  
搜狗拼音输入法 全 :
```

4.1.7 用继承的眼光看Object类 源码面前，了无秘密



前面的示例中，main方法实际上执行的是

```
public void println(Object x)
```

这一方法内部调用了String类的valueOf方法。

valueOf方法内部又调用了Object.toString()方法。

```
public String toString() {  
    return getClass().getName() +  
        "@ " + Integer.toHexString(hashCode());  
}
```

hashCode()方法是本地方法，由JVM设计者实现：

```
public native int hashCode();
```

4.1.7 用继承的眼光看Object类_神奇的 “+” 号

- 如下图所示的程序，Fruit类覆盖了Object类的toString方法。

```
public class Fruit {  
    public String toString() {  
        return "Fruit toString." ;  
    }  
    public static void main(String[] args){  
        Fruit f = new Fruit();  
        System.out.println("f=" + f);  
    }  
}
```

- 为何一个字符串和一个对象“相加”，得到以下的结果？

```
C:\Users\Shinelon\Desktop\code\plus>javac Fruit.java  
C:\Users\Shinelon\Desktop\code\plus>java Fruit  
f=Fruit toString.
```


4.1.7 用继承的眼光看Object类_神奇的 “+” 号

- 如下图所示的程序，Fruit类覆盖了Object类的toString方法。

```
public class Fruit {  
    public String toString() {  
        return "Fruit toString." ;  
    }  
    public static void main(String[] args){  
        Fruit f = new Fruit();  
        System.out.println("f=" + f);  
    }  
}
```

结论：在“+”运算中,当任何一个对象与一个String对象连接时,会隐式地调用其toString()方法，默认情况下，此方法返回“类名 @ + hashCode”。为了返回有意义的信息，子类可以重写toString()方法。

- 为何一个字符串和一个对象“相加”，得到以下的结果？

```
C:\Users\Shinelon\Desktop\code\plus>javac Fruit.java  
  
C:\Users\Shinelon\Desktop\code\plus>java Fruit  
f=Fruit toString.
```

4.1.8 多态_Polymorphism

```
class Animal {  
    ...  
}  
class Lion extends Animal {  
    ...  
}
```

- 在面向对象的理论中，多态性的定义是：同一操作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。简单来说：**相同的一条语句，在不同的运行环境中可以产生不同的运行结果。**
- 多态的最本质特征就是父类（或接口）变量可以引用子类（或实现了接口的类）对象。换句话说：子类对象可以被当成基类对象使用！
- 典型代码如下：

```
Animal a = new Lion();
```
- 此时我们称对象a是Lion类的**上转型对象**，总是可以让更一般的对象容纳更具体化的对象。特别地，Java类库的最顶层基类是Object类。因此每个对象都可以赋值给Object变量。

4.1.8 多态

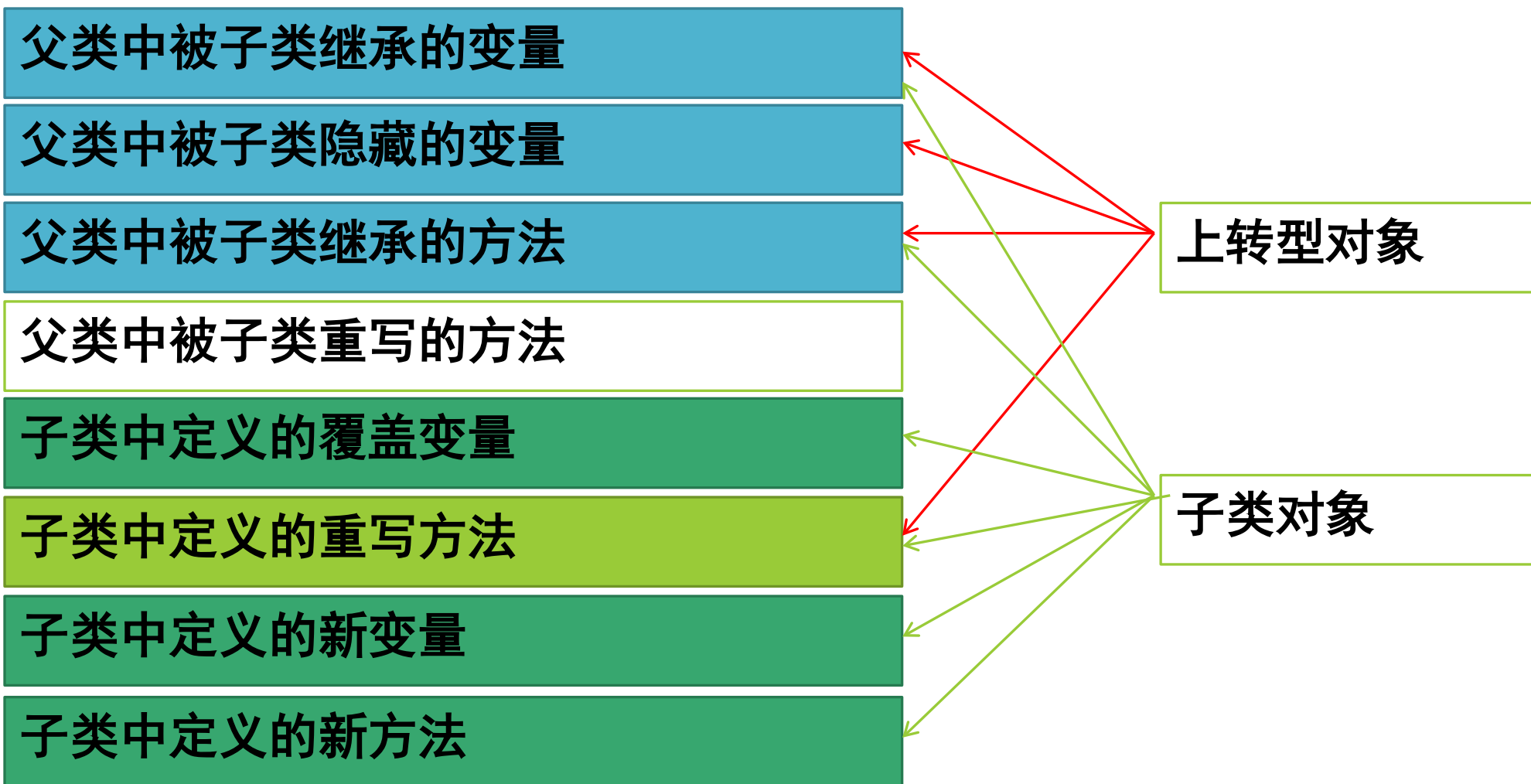
- 子类对象可以直接赋给基类变量。（上转型对象）
- 基类对象要赋给子类对象变量, 必须执行类型转换。
 - 语法格式:

子类对象变量=(子类名)基类对象名;
 - 不能乱转换。如果类型转换失败Java会抛出 `ClassCastException` 异常
- 怎样判断对象是否可以转换?
 - 可以使用 `instanceof` 运算符判断一个对象是否可以转换为指定的类型。

```
Object obj="Hello";
if(obj instanceof String)
    System.out.println("obj对象可以被转换为字符串");
```

4.1.8 多态

上转型对象的特点



本章导读

□4.1 继承 Inheritance

✓4.2 抽象类和抽象方法 Abstract Class & Abstract Method

□4.3 内部类 Inner Class

□4.4 接口 Interface

□4.5 Lambda表达式 λ Expression

□4.6 接口与抽象类 Interface & Abstract Class

□4.7 大话泛型 Generic Programming

□4.8 枚举类 Enumeration Class

□4.9 作业及延伸

4.2 抽象类与抽象方法

Abstract Class & Abstract Method

- 有abstract修饰的类称为“抽象类”，它只定义了什么方法应该存在，**不能创建对象**，必须派生出一个子类，并在子类中实现对其未实现方法之后，才能使用new关键字创建对象。
- 在方法前加上abstract就形成抽象方法，只有方法声明，没有实现代码。

```
abstract class Person {  
    public abstract String getDescription();  
}
```

- 一个抽象类中可以包含非抽象方法和成员变量。**包含抽象方法的类一定是抽象类，但抽象类中的方法不一定是抽象方法。**
- **从抽象类继承的子类必须实现父类的所有抽象方法**，否则，它仍然是抽象类。

4.2 抽象类与抽象方法_abstract关键字

- 思考？
- `abstract`和`final`不能同时修饰一个类。
- `abstract`不能修饰构造器，抽象类定义的构造器必须是普通构造器。
- `static`和`abstract`不能同时修饰某一方法，但它们可以同时修饰内部类。
- `abstract`不能和`private`同时修饰某一方法。

4.2 抽象类与抽象方法_抽象类的使用

- 抽象类不能创建对象，一般用来引用子类对象。

- 示例：

```
Person p;  
p=new Employee();
```

- 即如下模式：

抽象类 抽象类变量 = new 派生自抽象类的具体子类();

- 抽象类的作用：

1. 抽象类代表了一种未完成的类设计，体现的是一种模板，避免子类设计的随意性。
2. 抽象类和模板模式：父类提供多个子类的通用方法，并把一个或多个方法留给子类实现。

4.2 抽象类与抽象方法_代码示例

```
public abstract class Shape {
{
    System.out.println("执行Shape的初始化块...");
}
private String color;
// 定义一个计算周长的抽象方法
public abstract double calPerimeter();
// 定义一个返回形状的抽象方法
public abstract String getType();
// 定义Shape的构造器, 该构造器并不是用于创建Shape对象,
// 而是用于被子类调用
public Shape(){}
public Shape(String color) {
    System.out.println("执行Shape的构造器...");
    this.color = color;
}
public void setColor(String color) {
    this.color = color;
}
public String getColor() {
    return this.color;
}
}
```

```
public class Triangle extends Shape {
    // 定义三角形的三边
    private double a;
    private double b;
    private double c;
    public Triangle(String color, double a
        , double b, double c) {
        super(color);
        this.setSides(a, b, c);
    }
    public void setSides(double a, double b, double c) {
        if (a >= b + c || b >= a + c || c >= a + b) {
            System.out.println("三角形两边之和必须大于第三边");
            return;
        }
        this.a = a;
        this.b = b;
        this.c = c;
    }
    // 重写Shape类的计算周长的抽象方法
    public double calPerimeter() {
        return a + b + c;
    }
    // 重写Shape类的返回形状的抽象方法
    public String getType() {
        return "三角形";
    }
}
```

```
public class Circle extends Shape {
    private double radius;
    public Circle(String color, double radius) {
        super(color);
        this.radius = radius;
    }
    public void setRadius(double radius) {
        this.radius = radius;
    }
    // 重写Shape类的计算周长的抽象方法
    public double calPerimeter() {
        return 2 * Math.PI * radius;
    }
    // 重写Shape类的返回形状的抽象方法
    public String getType() {
        return getColor() + "圆形";
    }
    public static void main(String[] args) {
        Shape s1 = new Triangle("黑色", 3, 4, 5);
        Shape s2 = new Circle("黄色", 3);
        System.out.println(s1.getType());
        System.out.println(s1.calPerimeter());
        System.out.println(s2.getType());
        System.out.println(s2.calPerimeter());
    }
}
```

AbstractClass

本章导读

- 4.1 继承 Inheritance
- 4.2 抽象类和抽象方法 Abstract Class & Abstract Method
- ✓4.3 内部类 Inner Class
- 4.4 接口 Interface
- 4.5 Lambda表达式 λ Expression
- 4.6 接口与抽象类 Interface & Abstract Class
- 4.7 大话泛型 Generic Programming
- 4.8 枚举类 Enumeration Class
- 4.9 作业及延伸

4.3 内部类

- 4.3.1 非静态内部类
- 4.3.2 静态内部类
- 4.3.3 内部类的使用
- 4.3.4 局部内部类
- 4.3.5 Java8改进的匿名内部类

4.3 内部类_Inner Class

- 回忆：一个类中的成员类型有哪五种？（成员变量、方法、构造器、初始化块、？）
- 把一个类放在另一个类的内部定义，这个定义在其他类内部的类就被称为内部类，有的也叫嵌套类，包含内部类的类也被称为外部类有的也叫宿主类。
- 内部类提供了更好的封装，内部类成员可以直接访问外部类的私有数据，因为内部类被当成其他外部类成员。
- 匿名内部类适合用于创建那些仅需要一次使用的类。

4.3.1 非静态内部类

- 定义内部类非常简单，只要把一个类放在另一个类内部定义即可。从语法角度来看，内部类与外部类的区别主要有以下两点：
 1. 内部类比外部类可以多使用3个修饰符： `private` 、 `protected` 、 `static`。
 2. 非静态内部类不能拥有静态成员。
- 当在非静态内部类的方法内访问某个变量时，系统第一步先找局部变量，第二步，内部类的属性，第三步。外部类的属性。

4.3.1 非静态内部类

非静态内部类代码示例

- 右图所示的程序展示了一个典型的非静态内部类。
- 思考：Cow类对象与CowLeg对象的内存分布。

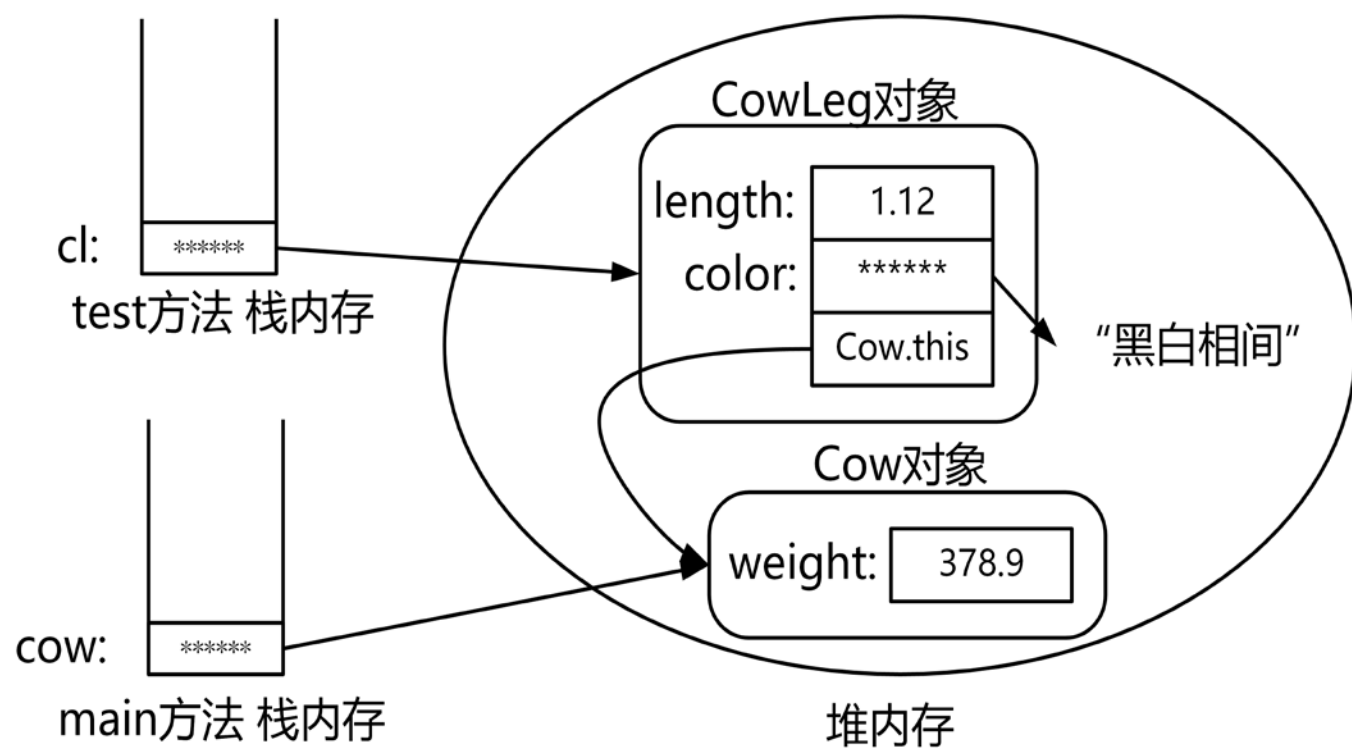
```
public class InnerClass {  
    public static void main(String[] args) {  
        Cow cow = new Cow(378.9);  
        cow.test();  
    }  
}
```

```
class Cow {  
    private double weight;  
    // 外部类的两个重载的构造器  
    public Cow(){}  
    public Cow(double weight) {  
        this.weight = weight;  
    }  
    private class CowLeg { // 定义一个非静态内部类  
        // 非静态内部类的两个实例变量  
        private double length;  
        private String color;  
        // 非静态内部类的两个重载的构造器  
        public CowLeg(){}  
        public CowLeg(double length, String color) {  
            this.length = length;  
            this.color = color;  
        }  
        // 下面省略length、color的setter和getter方法  
  
        public void info() { // 非静态内部类的实例方法  
            System.out.println("当前牛腿颜色是： "  
                + color + "，高： " + length);  
            // 直接访问外部类的private修饰的成员变量  
            System.out.println("本牛腿所在奶牛重： " + weight);  
        }  
    }  
}  
  
    public void test() {  
        CowLeg cl = new CowLeg(1.12, InnerClass  
            cl.info();  
    }  
}
```

4.3.1 非静态内部类

非静态内部类代码示例

- 思考：Cow类对象与CowLeg对象的内存分布。



```
public class InnerClass {  
    public static void main(String[] args) {  
        Cow cow = new Cow(378.9);  
        cow.test();  
    }  
}
```

```
//外部类的两个重载的构造器  
public Cow(){  
    public Cow(double weight) {  
        this.weight = weight;  
    }  
private class CowLeg { //定义一个非静态内部类  
    //非静态内部类的两个实例变量  
    private double length;  
    private String color;  
    //非静态内部类的两个重载的构造器  
    public CowLeg(){  
    public CowLeg(double length , String color) {  
        this.length = length;  
        this.color = color;  
    }  
    //下面省略length、color的setter和getter方法  
  
    public void info() { //非静态内部类的实例方法  
        System.out.println("当前牛腿颜色是: " + color + ", 高: " + length);  
        //直接访问外部类的private修饰的成员变量  
        System.out.println("本牛腿所在奶牛重: " + weight);  
    }  
}  
public void test() {  
    CowLeg cl = new CowLeg(1.12, "黑白相间");  
    cl.info();  
}
```

4.3.2 静态内部类

- 如果用static修饰一个内部类，称为类内部类，也称为静态内部类。
- 静态内部类可以包含静态成员，也可以包含非静态成员。静态内部类不能访问外部类的实例成员，只能访问外部类的类成员。
- 静态内部类的对象寄存在外部类里，非静态内部类的对象寄存在外部类实例里

4.3.3 内部类的使用

1. 在外部类内部使用内部类—不要在外部类的静态成员中使用非静态内部类，因为静态成员不能访问非静态成员。
2. 在外部类以外使用非静态内部类。
 - `private` 修饰的内部类只能在外部类内部使用。
 - 在外部类以外的地方使用内部类，内部类完整的类名应该为
 - 在外部类以外的地方使用非静态内部类创建对象的语法如下：

```
OuterClass.InnerClass
```

- 在外部类以外的地方使用静态内部类创建对象的语法如下：

```
OuterInstance.new InnerConstructor();
```

```
new OuterClass.InnerConstructor();
```

4.3.4 局部内部类

- 如果把一个内部类放在方法里定义，这就是局部内部类，仅仅在这个方法里有效。
- 局部内部类不能在外部类的方法以外的地方使用，那么局部内部类也不能使用访问控制符和static修饰。
- 如果需要用局部内部类定义变量、创建实例或派生子类，那么都只能在局部内部类所在的方法进行。

4.3.4 局部内部类

代码示例

```
public class LocalInnerClass {  
    public static void main(String[] args) {  
        // 定义局部内部类  
        class InnerBase {  
            int a;  
        }  
        // 定义局部内部类的子类  
        class InnerSub extends InnerBase {  
            int b;  
        }  
        // 创建局部内部类的对象  
        InnerSub is = new InnerSub();  
        is.a = 5;  
        is.b = 8;  
        System.out.println("InnerSub对象的a和b实例变量是： "  
            + is.a + ", " + is.b);  
    }  
}
```

LocalInnerClass

4.3.5 Java8改进匿名内部类

- 匿名内部类适合创建那种只需要一次使用的类，定义匿名内部类的语法格式如下：

```
new 实现接口 ( ) | 父类构造器 ( 实例列表) {  
    //匿名内部类的 类体部分  
}
```

- 匿名内部类不能是抽象类，匿名内部类不能定义构造器。
- 在Java8以前，Java要求局部内部类、匿名内部类访问的局部变量必须使用final修饰，从Java8开始这个限制被取消了，Java8更加智能：**如果局部变量被匿名内部类访问，那么该局部变量相当于自动使用了final修饰。**

4.3.5 Java8改进匿名内部类

代码示例

```
abstract class A {  
    abstract void test();  
}  
  
public class AnonymousInnerClass {  
    public static void main(String[] args) {  
        int age = 8; // ①  
        // 下面代码将会导致编译错误  
        // 由于age局部变量被匿名内部类访问了，因此age相当于被final修饰了  
        // age = 2;  
        A a = new A() {  
            public void test() {  
                // 在Java 8以前下面语句将提示错误：age必须使用final修饰  
                // 从Java 8开始，匿名内部类、局部内部类允许访问非final的局部变量  
                System.out.println(age);  
            }  
        };  
        a.test();  
    }  
}
```

AnonymousInnerClass

打响重构第一弹！_又见Student类

- 各位同学是否还记得上节课要求实现的Student类？
- 在学习完本节课的内容之后，我们将利用所学对Student类进行重构！使之更加完善！
- 主要的重构围绕内部类与抽象类展开！！！！
- 在重构之前！我们先加入几条新的需求！

本章导读

- 4.1 继承 Inheritance
- 4.2 抽象类和抽象方法 Abstract Class & Abstract Method
- 4.3 内部类 Inner Class
- ✓ 4.4 接口 Interface
- 4.5 Lambda表达式 λ Expression
- 4.6 接口与抽象类 Interface & Abstract Class
- 4.7 大话泛型 Generic Programming
- 4.8 枚举类 Enumeration Class
- 4.9 作业及延伸

4.4 接口

- 4.4.1 接口定义
- 4.4.2 接口的继承
- 4.4.3 使用接口
- 4.4.4 面向接口编程

4.4 接口_Interface

- 接口定义的是多个类共同的行为规范，这些行为是与外部交流的通道，这就意味着接口里通常是定义一组公用的方法。
- 程序员可以用接口进行程序的框架设计，而不必关心实现的细节，排除细节对框架设计的干扰。
- 通过接口，可以实现Java语言本身不具备的类的多继承机制，一个类可以实现多个接口。
- 接口体现了规范与实现分离的设计。

4.4.1 接口定义

- 和类定义不同，定义接口不再用class关键字，而是使用interface关键字。

语法如下：

```
[修饰符] interface 接口名 extends 父接口1,父接口2 ...  
{  
    零个到多个常量定义...  
    零个到多个抽象方法定义...  
    零个到多个内部类、接口、枚举定义...  
    零个到多个默认方法或类方法定义...  
}
```

只有在Java 8 以上的版本中才允许在接口中定义默认方法、类方法。

- 修饰符可以是public或者省略。

4.4.1 接口定义

- 接口定义的是多个类共同的公共行为规范，因此**接口中所有的成员都是public访问权限**，程序员在定义接口成员是**可以省略public访问控制符**。
- 在定义接口时，接口里可以包含成员变量（**只能是静态常量**），方法（只能是抽象实例方法、类方法或默认方法），内部类（包括内部接口、枚举类），也就是说
 - 接口中定义的变量，默认使用 `public static final` 修饰，定义时可省略，必须在定义时指定默认值。
 - 抽象实例方法默认使用 `public abstract` 修饰，定义时可省略。
 - 类方法使用 `static` 修饰，不可省略，可以通过接口名直接调用。
 - **默认方法使用 `default` 修饰，不可省略，需要通过实现接口的子类对象来调用。**

4.4.1 接口定义

代码示例

```
package yeeku;
public class OutputFieldTest
{
    public static void main(String[] args)
    {
        // 访问另一个包中的Output接口的MAX_CACHE_LINE
        System.out.println(lee.Output.MAX_CACHE_LINE);
        // 下面语句将引起"为final变量赋值"的编译异常
        // lee.Output.MAX_CACHE_LINE = 20;
        // 使用接口来调用类方法
        System.out.println(lee.Output.staticTest());
    }
}
```

```
package lee;
public interface Output {
    // 接口里定义的成员变量只能是常量
    int MAX_CACHE_LINE = 50;
    // 接口里定义的普通方法只能是public的抽象方法
    void out();
    void getData(String msg);
    // 在接口中定义默认方法，需要使用default修饰
    default void print(String... msgs) {
        for (String msg : msgs)
        {
            System.out.println(msg);
        }
    }
    // 在接口中定义默认方法，需要使用default修饰
    default void test() {
        System.out.println("默认的test()方法");
    }
    // 在接口中定义类方法，需要使用static修饰
    static String staticTest() {
        return "接口里的类方法";
    }
}
```

InterfaceFieldTest

4.4.2 接口的继承

- 接口的继承和类继承不一样，接口完全支持多继承，子接口扩展某个父接口将会获得父接口的所有抽象方法，常量属性，内部类和枚举类定义。

```
interface interfaceA {  
    int PROP_A = 5;  
    void testA();  
}  
interface interfaceB {  
    int PROP_B = 6;  
    void testB();  
}  
interface interfaceC extends interfaceA, interfaceB {  
    int PROP_C = 7;  
    void testC();  
}  
public class InterfaceExtendsTest {  
    public static void main(String[] args) {  
        System.out.println(interfaceC.PROP_A);  
        System.out.println(interfaceC.PROP_B);  
        System.out.println(interfaceC.PROP_C);  
    }  
}
```

InterfaceExtendsTest

4.4.3 使用接口

- 接口可以用于声明引用类型的变量，但接口**不能用于创建实例**。
- 当使用接口来声明引用类型的变量时，这个引用类型的变量必须**引用到其实现类的对象**。
- 一个类可以实现一个或多个接口，继承使用**extends**关键字，实现接口则使用**implements**关键字。

```
[修饰符] class 类名 extends 父类 implements 接口1 , 接口2 ... {  
    // 类体部分  
}
```

- 一个类实现了一个或多个接口之后，这个类必须完全实现这些接口里所定义的全部抽象方法（也就是重写这些抽象方法）；否则，该类将保留从父接口那里继承到的抽象方法，该类也必须定义成抽象类。

4.4.3 使用接口

代码示例

- 前文中，我们定义了一个输出接口Output，现在我们定义一个打印机类Printer来实现该接口。
- 为了演示一个类可以实现多个接口，我们新增加一个产品接口Product，包含一个抽象方法，返回产品的生产时间。

```
public interface Output {  
    // 接口里定义的成员变量只能是常量  
    int MAX_CACHE_LINE = 50;  
    // 接口里定义的普通方法只能是public的抽象方法  
    void out();  
    void getData(String msg);  
    // 在接口中定义默认方法，需要使用default修饰  
    default void print(String... msgs) {  
        for (String msg : msgs)  
        {  
            System.out.println(msg);  
        }  
    }  
    // 在接口中定义默认方法，需要使用default修饰  
    default void test() {  
        System.out.println("默认的test()方法");  
    }  
    // 在接口中定义类方法，需要使用static修饰  
    static String staticTest() {  
        return "接口里的类方法";  
    }  
}
```

```
// 定义一个Product接口  
public interface Product {  
    int getProduceTime();  
}
```

```
// 让Printer类实现Output和Product接口  
public class Printer implements Output, Product {  
    private String[] printData = new String[MAX_CACHE_LINE];  
    // 用以记录当前需打印的作业数  
    private int dataNum = 0;  
    public void out() {  
        // 只要还有作业，继续打印  
        while(dataNum > 0) {  
            System.out.println("打印机打印：" + printData[0]);  
            // 把作业队列整体前移一位，并将剩下的作业数减1  
            System.arraycopy(printData, 1, printData, 0, --dataNum);  
        }  
    }  
    public void getData(String msg) {  
        if (dataNum >= MAX_CACHE_LINE) {  
            System.out.println("输出队列已满，添加失败");  
        } else {  
            // 把打印数据添加到队列里，已保存数据的数量加1。  
            printData[dataNum++] = msg;  
        }  
    }  
    public int getProduceTime() {  
        return 45;  
    }  
}
```

4.4.3 使用接口

代码示例

- **接口回调**是指：可以把某一接口实现类创建的对象引用赋给该接口声明的接口变量中，那么该接口变量就可以调用被类实现的接口中的方法。
- 接口回调是多态的一种体现，不同的类在实现同一接口时，可能具有不同的功能体现，因此接口回调可能产生不同的行为。

```
public class Test {  
    public static void main(String[] args) {  
        // 创建一个Printer对象，当成Output使用  
        Output o = new Printer();  
        o.getData("轻量级Java EE企业应用实战");  
        o.getData("疯狂Java讲义");  
        o.out();  
        o.getData("疯狂Android讲义");  
        o.getData("疯狂Ajax讲义");  
        o.out();  
        // 调用Output接口中定义的默认方法  
        o.print("孙悟空", "猪八戒", "白骨精");  
        o.test();  
        // 创建一个Printer对象，当成Product使用  
        Product p = new Printer();  
        System.out.println(p.getProduceTime());  
        // 所有接口类型的引用变量都可直接赋给Object类型的变量  
        Object obj = p;  
    }  
}
```

InterfaceTest

4.4.4 面向接口编程

- 接口体现了规范与实现分离的原则。充分利用接口可以很好地提高系统的可扩展性和可维护性，降低程序间的耦合。
- 接下来我们将简单了解两种“面向接口编程”的设计模式：
 - 简单工厂模式
 - 命令模式

4.4.4 面向接口编程

简单工厂模式

- 让我们回顾一下上节定义的Output接口，及其实现类Printer，现在有一个场景：假设程序中有个Computer类需要组合一个输出设备，有两种方法：直接让Computer类组合一个Printer类，或者让Computer类组合一个Output。

```
public interface Output {  
    // 接口里定义的成员变量只能是常量  
    int MAX_CACHE_LINE = 50;  
    // 接口里定义的普通方法只能是public的抽象方法  
    void out();  
    void getData(String msg);  
    // 在接口中定义默认方法，需要使用default修饰  
    default void print(String... msgs) {  
        for (String msg : msgs)  
        {  
            System.out.println(msg);  
        }  
    }  
    // 在接口中定义默认方法，需要使用default修饰  
    default void test() {  
        System.out.println("默认的test()方法");  
    }  
    // 在接口中定义类方法，需要使用static修饰  
    static String staticTest() {  
        return "接口里的类方法";  
    }  
}
```

```
public class Printer implements Output {  
    private String[] printData = new String[MAX_CACHE_LINE];  
    // 用以记录当前需打印的作业数  
    private int dataNum = 0;  
    public void out() {  
        // 只要还有作业，继续打印  
        while(dataNum > 0) {  
            System.out.println("打印机打印: " + printData[0]);  
            // 把作业队列整体前移一位，并将剩下的作业数减1  
            System.arraycopy(printData, 1, printData, 0, -dataNum);  
        }  
    }  
    public void getData(String msg) {  
        if (dataNum >= MAX_CACHE_LINE) {  
            System.out.println("输出队列已满，添加失败");  
        } else {  
            // 把打印数据添加到队列里，已保存数据的数量加1。  
            printData[dataNum++] = msg;  
        }  
    }  
}
```

4.4.4 面向对象编程

简单工厂模式

- 对于上述场景两种选择都无可厚非，但是有一天系统需要重构，要求使用更好的打印机BetterPrinter来代替Printer类，这就需要打开Computer类的源码进行修改。如果系统中只有一个Computer类还好，但如果系统中有100个类组合了Printer类呢？
- 我们来看工厂模式是如何处理这个问题的：
 - 首先定义一个Computer类组合Output类型的变量。

```
public class Computer
{
    private Output out;
    public Computer(Output out)
    {
        this.out = out;
    }
    // 定义一个模拟获取字符串输入的方法
    public void keyIn(String msg)
    {
        out.getData(msg);
    }
    // 定义一个模拟打印的方法
    public void print()
    {
        out.out();
    }
}
```

4.4.4 面向接口编程

简单工厂模式

- 此时，Computer类已经完全与Printer类分离，只是与Output接口耦合。Computer类不再负责创建Output对象，系统提供了一个Output工厂来负责生成Output对象。

```
public class OutputFactory {  
    public Output getOutput() {  
        return new Printer();  
    }  
    public static void main(String[] args) {  
        OutputFactory of = new OutputFactory();  
        Computer c = new Computer(of.getOutput());  
        c.keyIn("轻量级Java EE企业应用实战");  
        c.keyIn("疯狂Java讲义");  
        c.print();  
    }  
}
```

4.4.4 面向接口编程

简单工厂模式

- 现在，让我们定义一个更为高级的打印机类，同样实现Output接口。

```
public class BetterPrinter implements Output {  
    private String[] printData = new String[MAX_CACHE_LINE * 2];  
    // 用以记录当前需打印的作业数  
    private int dataNum = 0;  
    public void out() {  
        // 只要还有作业，继续打印  
        while(dataNum > 0) {  
            System.out.println("高速打印机正在打印: " + printData[0]);  
            // 把作业队列整体前移一位，并将剩下的作业数减1  
            System.arraycopy(printData, 1, printData, 0, --dataNum);  
        }  
    }  
    public void getData(String msg) {  
        if (dataNum >= MAX_CACHE_LINE * 2) {  
            System.out.println("输出队列已满，添加失败");  
        } else {  
            // 把打印数据添加到队列里，已保存数据的数量加1。  
            printData[dataNum++] = msg;  
        }  
    }  
}
```

4.4.4 面向接口编程

简单工厂模式

- 接下来，我们需要做的仅仅是改动一行代码，就可以让所有组合了Output接口变量的类更换新型打印机！

```
public class OutputFactory {  
    public Output getOutput() {  
        //return new Printer();  
        return new BetterPrinter();  
    }  
    public static void main(String[] args) {  
        OutputFactory of = new OutputFactory();  
        Computer c = new Computer(of.getOutput());  
        c.keyIn("轻量级Java EE企业应用实战");  
        c.keyIn("疯狂Java讲义");  
        c.print();  
    }  
}
```

4.4.4 面向接口编程

命令模式

- 考虑这样一种场景：某个方法需要完成某一个行为，但这个行为的具体实现无法确定，必须等到执行该方法时才可以确定。具体举例：假设有个方法需要遍历某个数组元素，但无法确定在遍历数组元素时如何处理这些元素，需要在调用该方法时指定具体的处理行为。
- 这个需求看起来有点奇怪：它不是要求把数据作为参数传递给方法，而是需要把“处理行为”作为一个参数传给方法。部分编程语言（Ruby）支持这一特性，允许将一个代码块作为参数，而Java在jdk1.8版本新增Lambda表达式前并不支持。
- 让我们来看命令模式是如何不借助Lambda表达式做到这一点的。

4.4.4 面向接口编程

命令模式

- 首先定义一个命令接口，包含一个抽象方法来封装“处理行为”。

```
public interface Command {  
    // 接口里定义的process()方法用于封装“处理行为”  
    void process(int[] target);  
}
```

- 接下来定义需要处理数组的处理类，该类中的process方法并不能确定具体的处理行为，因此定义该方法时传入一个Command参数。

```
public class ProcessArray {  
    public void process(int[] target , Command cmd) {  
        .....  
        cmd.process(target);  
    }  
}
```


4.4.4 面向接口编程

命令模式

- 通过Command接口，我们实现了ProcessArray类与“处理行为”的分离，我们只需要传入Command接口不同的实现类对象，就能达到目标传入“处理行为”的目的。

```
public class AddCommand implements Command {  
    public void process(int[] target) {  
        int sum = 0;  
        for (int tmp : target )  
            sum += tmp;  
        System.out.println("数组元素的总和是:" + sum);  
    }  
}
```

```
public class PrintCommand implements Command {  
    public void process(int[] target) {  
        for (int tmp : target )  
            System.out.println("迭代输出目标数组的元素:" + tmp);  
    }  
}
```

```
public class CommandTest {  
    public static void main(String[] args) {  
        ProcessArray pa = new ProcessArray();  
        int[] target = {3, -4, 6, 4};  
        // 第一次处理数组，具体处理行为取决于PrintCommand  
        pa.process(target, new PrintCommand());  
        System.out.println("-----");  
        // 第二次处理数组，具体处理行为取决于AddCommand  
        pa.process(target, new AddCommand());  
    }  
}
```

4.4.4 面向对象编程

思考

- 如下的代码，利用我们学过的哪部分知识，可以将代码进一步精简？

```
public class AddCommand implements Command {  
    public void process(int[] target) {  
        int sum = 0;  
        for (int tmp : target )  
            sum += tmp;  
        System.out.println("数组元素的总和是:" + sum);  
    }  
}
```

```
public class PrintCommand implements Command {  
    public void process(int[] target) {  
        for (int tmp : target )  
            System.out.println("迭代输出目标数组的元素:" + tmp);  
    }  
}
```

```
public class CommandTest {  
    public static void main(String[] args) {  
        ProcessArray pa = new ProcessArray();  
        int[] target = {3, -4, 6, 4};  
        // 第一次处理数组，具体处理行为取决于PrintCommand  
        pa.process(target, new PrintCommand());  
        System.out.println("-----");  
        // 第二次处理数组，具体处理行为取决于AddCommand  
        pa.process(target, new AddCommand());  
    }  
}
```

本章导读

- 4.1 继承 Inheritance
- 4.2 抽象类和抽象方法 Abstract Class & Abstract Method
- 4.3 内部类 Inner Class
- 4.4 接口 Interface
- ✓ 4.5 **Lambda表达式 λ Expression**
- 4.6 接口与抽象类 Interface & Abstract Class
- 4.7 大话泛型 Generic Programming
- 4.8 枚举类 Enumeration Class
- 4.9 作业及延伸

4.5 Lambda表达式（选讲）

- 4.5.1 Lambda表达式形式
- 4.5.2 Lambda表达式与函数式接口
- 4.5.3 方法引用与构造器引用
- 4.5.4 Lambda表达式与匿名内部类

4.5 Lambda表达式

- Lambda表达式是Java8的主要更新。
- Lambda表达式将代码块作为方法的参数。
- Lambda表达式允许使用更加简洁的代码创建**只有一个抽象方法的接口（函数式接口）**
- 正式学习Lambda表达式前，先感受一下Lambda表达式在命令模式中的“威力”。

4.5 Lambda表达式

代码示例

- 上节的思考，答案当然是**匿名内部类**！我们无需再单独定一个AddCommand类。

```
public class CommandTest {  
    public static void main(String[] args) {  
        ProcessArray pa = new ProcessArray();  
        int[] array = {3, -4, 6, 4};  
        // 处理数组，具体处理行为取决于匿名内部类  
        pa.process(array, new Command() {  
            public void process(int[] target) {  
                int sum = 0;  
                for (int tmp : target) {  
                    sum += tmp;  
                }  
                System.out.println("数组元素的总和是:" + sum);  
            }  
        });  
    }  
}
```

而Lambda表达式还能再次基础上进一步简化！

4.5 Lambda表达式 代码示例

- 我们可以看到使用Lambda表达式，相比匿名内部类，省去方法的声明部分，直接传入一个代码块！

```
public class CommandTest2 {  
    public static void main(String[] args) {  
        ProcessArray pa = new ProcessArray();  
        int[] array = {3, -4, 6, 4};  
        // 处理数组，具体处理行为取决于匿名内部类  
        pa.process(array, (int[] target) -> {  
            int sum = 0;  
            for (int tmp : target )  
                sum += tmp;  
            System.out.println("数组元素的总和是:" + sum);  
        });  
    }  
}
```

```
public class CommandTest {  
    public static void main(String[] args) {  
        ProcessArray pa = new ProcessArray();  
        int[] array = {3, -4, 6, 4};  
        // 处理数组，具体处理行为取决于匿名内部类  
        pa.process(array, new Command() {  
            public void process(int[] target) {  
                int sum = 0;  
                for (int tmp : target )  
                    sum += tmp;  
                System.out.println("数组元素的总和是:" + sum);  
            }  
        });  
    }  
}
```

4.5.1 Lambda表达式形式

- Lambda表达式主要作用就是代替匿名内部类的繁琐语法。它由三部分组成：
 - 形参列表。形参列表允许省略形参类型。如果形参列表中只有一个参数，甚至连形参列表的圆括号也可以省略。
 - 箭头（->），必须通过英文等号和大于符号组成。
 - 代码块。
- 如果代码块只有包含一条语句，Lambda表达式允许省略代码块的花括号，这条语句不要用花括号表示语句结束。
- Lambda代码块只有一条return语句，甚至可以省略return关键字。而它的代码块中仅有一条省略了return的语句，Lambda表达式会自动返回这条语句的值。

4.5.1 Lamda表达式形式 代码示例

- 右图所示的程序展示了几种Lambda表达式的简化写法。

```
interface Eatable {  
    void taste();  
}  
interface Flyable {  
    void fly(String weather);  
}  
interface Addable {  
    int add(int a , int b);  
}
```

```
public class LambdaQs {  
    public void eat(Eatable e) { // 调用该方法需要Eatable对象  
        System.out.println(e);  
        e.taste();  
    }  
    public void drive(Flyable f) { // 调用该方法需要Flyable对象  
        System.out.println("我正在驾驶: " + f);  
        f.fly("【碧空如洗的晴日】");  
    }  
    public void test(Addable add) { // 调用该方法需要Addable对象  
        System.out.println("5与3的和为: " + add.add(5, 3));  
    }  
    public static void main(String[] args)  
    {  
        LambdaQs lq = new LambdaQs();  
        // Lambda表达式的代码块只有一条语句，可以省略花括号。  
        lq.eat(() -> System.out.println("苹果的味道不错！"));  
        // Lambda表达式的形参列表只有一个形参，省略圆括号  
        lq.drive(weather ->  
        {  
            System.out.println("今天天气是: " + weather);  
            System.out.println("直升机飞行平稳");  
        });  
        // Lambda表达式的代码块只有一条语句，省略花括号  
        // 代码块中只有一条语句，即使该表达式需要返回值，也可以省略return关键字。  
        lq.test((a , b) -> a + b);  
    }  
}
```

4.5.2 Lambda表达式与函数式接口

Functional Interface

- 函数式接口代表**只包含一个抽象方法的接口**，函数式接口可以包含多个默认方法、类方法，但只能声明一个抽象方法。Java 8 专门为函数式接口提供了@FunctionalInterface注解，该注解通常放在接口定义前面，对程序功能没有任何作用，用于告诉编译器执行更为严格的检查——检查该接口是否只包含一个抽象方法。
- 查阅Java 8 API可以发现大量函数式接口，例如：Runnable、ActionListener等。

4.5.2 Lambda表达式与函数式接口

- 如果采用匿名内部类语法来创建函数式接口的实例，只要实现一个抽象方法即可，在这种情况下即可采用Lambda表达式来创建对象，该**表达式创建出来的对象的目标类型就是这个函数式接口**。
- 由于Lambda表达式的结果是一个对象，因此完全可以用Lambda表达式赋值。

```
// Runnable接口中只包含一个无参数的方法
// Lambda表达式代表的匿名方法实现了Runnable接口中唯一的、无参数的方法
// 因此下面的Lambda表达式创建了一个Runnable对象
Runnable r = () -> {
    for(int i = 0 ; i < 100 ; i++)
    {
        System.out.println();
    }
};
```

4.5.2 Lambda表达式与函数式接口

- Lambda表达式有如下两个限制：
 - Lambda表达式的目标类型必须是明确的函数式接口。
 - Lambda表达式只能为函数式接口创建对象。Lambda表达式只能实现一个方法，因此它只能为只有一个抽象方法的接口（函数式接口）创建对象。

```
// 下面代码报错: 不兼容的类型: Object不是函数接口
Object obj = () -> {
    for(int i = 0 ; i < 100 ; i++)
    {
        System.out.println();
    }
};
```

4.5.2 Lambda表达式与函数式接口

- 为了保证Lambda表达式的目标类型是一个明确的函数式接口，可以有如下三种常见方式：
 - 将Lambda表达式赋值给函数式接口类型的变量。
 - 将Lambda表达式作为函数式接口类型的参数传给某个方法。
 - 使用函数式接口对Lambda表达式进行强制类型转换。

```
Object obj1 = (Runnable)() -> {  
    for(int i = 0 ; i < 100 ; i ++)  
    {  
        System.out.println();  
    }  
};
```

4.5.3 方法引用和构造器的引用

- 如果Lambda表达式的代码块只有一条语句，可以在代码块中使用方法的引用和构造器的引用。

种类	示例	说明	对应的Lambda表达式
引用类方法	类名::类方法	函数式接口中被实现方法的全部参数传给该类方法作为参数。	(a,b,...) -> 类名.类方法(a,b, ...)
引用特定对象的实例方法	特定对象::实例方法	函数式接口中被实现方法的全部参数传给该方法作为参数。	(a,b, ...) -> 特定对象.实例方法(a,b, ...)
引用某类对象的实例方法	类名::实例方法	函数式接口中被实现方法的第一个参数作为调用者，后面的参数全部传给该方法作为参数。	(a,b, ...) -> a.实例方法(b, ...)
引用构造器	类名::new	函数式接口中被实现方法的全部参数传给该构造器作为参数。	(a,b, ...) -> new 类名(a,b, ...)

4.5.3 方法引用和构造器引用

引用类方法

引用类方法	类名::类方法	函数式接口中被实现方法的全部参数传给该类方法作为参数。	(a,b,...) -> 类名.类方法(a,b, ...)
-------	---------	-----------------------------	-------------------------------

```
@FunctionalInterface
interface Converter {
    Integer convert(String from);
}

public class StaticMethodRefer {
    public static void main(String[] args) {
        // 下面代码使用Lambda表达式创建Converter对象
        Converter converter1 = from -> Integer.valueOf(from);
        // 方法引用代替Lambda表达式：引用类方法。
        // 函数式接口中被实现方法的全部参数传给该类方法作为参数。
        // Converter converter1 = Integer::valueOf;
        Integer val = converter1.convert("99");
        System.out.println(val); // 输出整数99
    }
}
```

StaticMethodRefer

4.5.3 方法引用和构造器引用

引用特定对象的实例方法

引用特定对象的
实例方法

特定对象::实例方法

函数式接口中被实现方法的全
部参数传给该方法作为参数。

(a,b, ...) -> 特定对象.实例方法(a,b, ...)

```
public class ObjectMethodRefer {  
  
    @FunctionalInterface  
    interface Converter {  
        Integer convert(String from);  
    }  
  
    public static void main(String[] args) {  
        // 下面代码使用Lambda表达式创建Converter对象  
        Converter converter2 = from -> "fkit.org".indexOf(from);  
        // 方法引用代替Lambda表达式：引用特定对象的实例方法。  
        // 函数式接口中被实现方法的全部参数传给该方法作为参数。  
        // Converter converter2 = "fkit.org"::indexOf;  
        Integer value = converter2.convert("it");  
        System.out.println(value); // 输出2  
    }  
}
```

ObjectMethodRefer

4.5.3 方法引用和构造器引用

引用某类对象的实例方法

引用某类对象的实例方法	类名::实例方法	函数式接口中被实现方法的第一个参数作为调用者，后面的参数全部传给该方法作为参数。	(a,b, ...) ->a.实例方法(b, ...)
-------------	----------	--	-----------------------------

```
@FunctionalInterface
interface MyTest
{
    String test(String a , int b , int c);
}

public class ClassMethodRefer {
    public static void main(String[] args) {
        // 下面代码使用Lambda表达式创建MyTest对象
        MyTest mt = (a , b , c) -> a.substring(b , c);
        // 方法引用代替Lambda表达式：引用某类对象的实例方法。
        // 函数式接口中被实现方法的第一个参数作为调用者，
        // 后面的参数全部传给该方法作为参数。
        // MyTest mt = String::substring;
        String str = mt.test("Java I Love you" , 2 , 9);
        System.out.println(str); // 输出:va I Lo
    }
}
```

ClassMethodRefer

4.5.3 方法引用和构造器引用

引用构造器

引用构造器	类名::new	函数式接口中被实现方法的全部参数传给该构造器作为参数。	(a,b, ...) ->new 类名(a,b, ...)
-------	---------	-----------------------------	-------------------------------

```
@FunctionalInterface
interface YourTest
{
    JFrame win(String title);
}

public class ConstructorRefer {
    public static void main(String[] args) {
        // 下面代码使用Lambda表达式创建YourTest对象
        YourTest yt = (String a) -> new JFrame(a);
        // 构造器引用代替Lambda表达式。
        // 函数式接口中被实现方法的全部参数传给该构造器作为参数。
        YourTest yt = JFrame::new;
        JFrame jf = yt.win("我的窗口");
        System.out.println(jf);
    }
}
```

ConstructorRefer

4.5.4 Lambda表达式与匿名内部类

- Lambda表达式与匿名内部类存在如下相同点：
 - Lambda表达式与匿名内部类一样，都可以直接访问“**effectively final**”的局部变量，以及外部类的成员变量（包括实例变量和类变量）。
 - Lambda表达式创建的对象与匿名内部类生成的对象一样，都可以直接调用从接口继承得到的默认方法。

4.5.4 Lambda表达式与匿名内部类

- Lambda表达式与匿名内部类主要存在如下区别：
 - 匿名内部类可以为任意接口创建实例——不管接口包含多少个抽象方法，只要匿名内部类实现所有的抽象方法即可。但Lambda表达式只能为函数式接口创建实例。
 - 匿名内部类可以为抽象类、甚至普通类创建实例，但Lambda表达式只能为函数式接口创建实例。
 - 匿名内部类实现的抽象方法的方法体允许调用接口中定义的默认方法；但Lambda表达式的代码块不允许调用接口中定义的默认方法。

本章导读

- 4.1 继承 Inheritance
- 4.2 抽象类和抽象方法 Abstract Class & Abstract Method
- 4.3 内部类 Inner Class
- 4.4 接口 Interface
- 4.5 Lambda表达式 λ Expression
- ✓ 4.6 接口与抽象类 Interface & Abstract Class
- 4.7 大话泛型 Generic Programming
- 4.8 枚举类 Enumeration Class
- 4.9 作业及延伸

4.6 接口与抽象类

- 相似性：

- 接口和抽象类都不能被实例化，它们都位于继承树的顶端，用于被其他类实现和继承。
- 接口和抽象类都可以包含抽象方法，实现接口或继承抽象类的普通子类都必须实现这些抽象方法。

4.6 接口与抽象类

● 差异性：

- 接口里只能包含抽象方法，不能包含已经提供实现的方法；抽象类则完全可以包含普通方法。
- 接口里只能定义静态常量属性，不能定义普通属性；抽象类里则既可以定义普通属性，也可以定义静态常量属性。
- 接口不包含构造器；抽象类里可以包含构造器，抽象类里的构造器并不是用于创建对象，而让其子类调用这些构造器来完成属于抽象类的初始化操作。
- 接口里不能包含初始化块，但抽象类则完全可以包含初始化块。
- 一个类最多只能有一个直接父类，包括抽象类；但一个类可以直接实现多个接口，通过实现多个接口可以弥补Java单继承的不足。

本章导读

- 4.1 继承 Inheritance
- 4.2 抽象类和抽象方法 Abstract Class & Abstract Method
- 4.3 内部类 Inner Class
- 4.4 接口 Interface
- 4.5 Lambda表达式 λ Expression
- 4.6 接口与抽象类 Interface & Abstract Class
- ✓ 4.7 大话泛型 Generic Programming
- 4.8 枚举类 Enumeration Class
- 4.9 作业及延伸

4.7 大话泛型

- 4.7.1 泛型概述
- 4.7.2 深入泛型
- 4.7.3 类型通配符
- 4.7.4 泛型方法（选讲）
- 4.7.5 擦除与转换（选讲）

4.7 大话泛型

- 回忆：右图所示的CPP的程序，同学们是否还记的是C++的什么语法特性？

```
//TemplateDemo.h
template<class T> class A{
public:
    T g(T a,T b);
    A();
};
```

```
//TemplateDemo.cpp
#include<iostream.h>
#include "TemplateDemo.h"

template<class T> A<T>::A(){}

template<class T> T A<T>::g(T a,T b){
    return a+b;
}

void main(){
    A<int> a;
    cout<<a.g(2,3.2)<<endl;
}
```

4.7.1 泛型概述_Generic

- Java5以后，引入了参数化类型的概念，允许程序在创建集合的时候指定集合元素的类型。这种参数化类型被称为泛型。
- 所谓泛型：就是允许在定义类、接口指定类型形参，这个类型形参在将在声明变量、创建对象时确定（即传入实际的类型参数，也可称为类型实参）。
- JDK1.5改写了集合框架中的全部接口和类，为这些接口、类增加了泛型支持，从而可以在声明集合变量、创建集合对象时传入类型实参，例如：`List<String>`和`ArrayList<String>`两种类型。

4.7.1 泛型概述_泛型类、接口的定义

- 泛型类：

- 泛型类的定义语法：

- 例： `class 名称 <泛型列表>` `class Cooperation <X,Y>`

- Cooperation是泛型类的名称，X，Y是泛型

- X，Y可以是任何类或接口，不能是基本类型数据

- 泛型接口：

- 泛型接口的定义语法与泛型类相似：

- `interface 名称 <泛型列表>`

4.7.1 泛型概述

泛型类的使用

- 使用泛型类声明对象时，必须要指定泛型的实际类型。

```
Cooperation <People,Dog> work = new Cooperation<People,Dog>();  
ArrayList<String> list = new ArrayList<String>();
```

- 在java7之前，使用泛型必须在调用构造器的时候后面也带有泛型，这样的写法比较复杂。在java7中引入了“**菱形**” / “**钻石**” 语法使得带有泛型的类的创建过程更加简单，Java可以自动推断尖括号里应该是什么泛型信息。

```
import java.util.*;  
public class DiamondTest {  
    public static void main(String[] args) {  
        // Java自动推断出ArrayList的<>里应该是String  
        List<String> books = new ArrayList<>();  
        books.add("疯狂Java讲义");  
        books.add("疯狂Android讲义");  
        // 遍历books集合，集合元素就是String类型  
        books.forEach(ele -> System.out.println(ele.length()));  
        // Java自动推断出HashMap的<>里应该是String, List<String>  
        Map<String, List<String>> schoolsInfo = new HashMap<>();  
        // Java自动推断出ArrayList的<>里应该是String  
        List<String> schools = new ArrayList<>();  
        schools.add("斜月三星洞");  
        schools.add("西天取经路");  
        schoolsInfo.put("孙悟空", schools);  
        // 遍历Map时，Map的key是String类型，value是List<String>类型  
        schoolsInfo.forEach((key, value) -> System.out.println(key + "->" + value));  
    }  
}
```

DiamondTest

4.7.1 泛型概述

代码示例

// 定义Book类时使用了泛型声明

```
class Book<T> {  
    // 使用T类型形参定义实例变量  
    private T info;
```

```
    public Book() {  
    }
```

// 下面方法中使用T类型形参来定义构造器

```
    public Book(T info) {  
        this.info = info;  
    }
```

// omit the getter & setter

```
}
```

```
public class GenericTest {
```

```
    public static void main(String[] args) {
```

// 由于传给T形参的是String，所以构造器参数只能是String

```
        Book<String> a1 = new Book<>("好好说话");
```

```
        System.out.println(a1.getInfo());
```

// 由于传给T形参的是Double，所以构造器参数只能是Double或double

```
        Book<Double> a2 = new Book<>(5.67);
```

```
        System.out.println(a2.getInfo());
```

```
    }
```

```
}
```

4.7.1 泛型概述

从泛型类派生子类

- 可以为创建泛型的接口和类创建子类，但是子类不能包含类型参数，必须是确定的参数类型。因此，以下代码是错误的：

```
public class A extends Apple<T> {}
```

- 如果想要从Apple类派生子类，可以采取下面两种写法：

```
public class A extends Apple<String> {}  
public class A extends Apple {}
```

- 第1种写法，所有Apple中的T则直接转换为String类型，子类A继承Apple的方法或参数中出现T的地方也是String类型。

4.7.2 深入泛型

- 在Java5引入“泛型”以前，Java提供的ArrayList类（顺序表的数据结构），是去类型化的，所有添加到ArrayList的对象都被作为Object对象，在使用时再强制转化为原本的类型。如下：

```
public class ListTest
{
    public static void main(String[] args)
    {
        // 创建一个只想保存字符串的List集合
        List strList = new ArrayList();
        strList.add("疯狂Java讲义");
        strList.add("疯狂Android讲义");
        strList.forEach(str -> System.out.println(((String)str)));
    }
}
```

GenericList

4.7.2 深入泛型

- 在Java5引入“泛型”以前，Java提供的ArrayList类（顺序表的数据结构），是去类型化的，所有添加到ArrayList的对象都被作为Object对象，在使用时再强制转化为原本的类型。如下：

```
public class ListTest
{
    public static void main(String[] args)
    {
        // 创建一个只想保存字符串的List集合
        List strList = new ArrayList();
        strList.add("疯狂Java讲义");
        strList.add("疯狂Android讲义");
        strList.forEach(str -> System.out.println(((String)str).length()));
    }
}
```

4.7.2 深入泛型

- 在Java5引入“泛型”以前，Java提供的ArrayList类（顺序表的数据结构），是去类型化的，所有添加到ArrayList的对象都被作为Object对象，在使用时再强制转化为原本的类型。如下：

```
public class ListTest
{
    public static void main(String[] args)
    {
        // 创建一个只想保存字符串的List集合
        List strList = new ArrayList();
        strList.add("疯狂Java讲义");
        strList.add("疯狂Android讲义");
        strList.forEach(str -> System.out.println(((String)str).length()));
    }
}
```

假设程序员“不小心”
把一个Integer对象丢
进了集合会发生什么？

4.7.2 深入泛型

- 假设程序员“不小心”把一个Integer对象添加到了ArrayList中。

```
import java.util.*;
public class ListErr {
    public static void main(String[] args) {
        // 创建一个只想保存字符串的List集合
        List strList = new ArrayList();
        strList.add("疯狂Java讲义");
        strList.add("疯狂Android讲义");
        // "不小心"把一个Integer对象"丢进"了集合
        strList.add(5);
        strList.forEach(str -> System.out.println(((String)str).length()));
    }
}
```

程序能够通过编译，但是会发生运行时错误，程序会在将Integer对象强制转换为String类型时，抛出ClassCastException异常。

4.7.2 深入泛型

- 利用泛型机制，我们可以将这个错误在编译阶段就暴露出来，不用等到系统上线运行才发现异常。

```
import java.util.*;
public class GenericList {
    public static void main(String[] args) {
        // 创建一个只想保存字符串的List集合
        List<String> strList = new ArrayList<String>();
        strList.add("疯狂Java讲义");
        strList.add("疯狂Android讲义");
        // 下面代码将引起编译错误
        strList.add(5); // ①
        strList.forEach(str -> System.out.println(str.length())); // ②
    }
}
```

右图所示的程序使用了泛型类 ArrayList，传入了类型参数 String，程序会在①处发生编译错误。

同时与不使用泛型的程序相比，程序在②处不必进行强制类型转换。

4.7.2 深入泛型 本质！并不存在泛型类

- 虽然可以把 `ArrayList<String>` 类当成 `ArrayList` 的子类，事实上 `ArrayList<String>` 类也确实是一种特殊的 `ArrayList` 类，这个 `ArrayList<String>` 对象只能添加 `String` 对象作为集合元素。但实际上，系统并没有为 `ArrayList<String>` 生成新的 `class` 文件，而且也不会把 `ArrayList<String>` 当成新类来处理。
- 不管为泛型的类型形参传入哪一种类型参数，对 `Java` 而言，他们依然被当成同一个类处理，在内存也只占用一块内存，因此在类方法、类初始块或者类变量的声明和初始化中不允许使用类型形参。

4.7.2 深入泛型

思考

- 除了观察ArrayList<String>类编译后的class文件外，还有什么方法可以证明本质上，泛型类并不存在？
- 提示：Object类

4.7.2 深入泛型

并不存在泛型类

- 由于系统中并不会真正生成泛型类，所以 instanceof 运算符后不能使用泛型类：

```
java.util.Collection<String> cs = new java.util.ArrayList<>();  
// 下面的代码编译时引起错误， instanceof 运算后不能使用泛型类  
if ( cs instanceof java.util.ArrayList<String>) {  
    ...  
}
```

4.7.3 类型通配符

- `List<String>`对象不能被当成`List<Object>`对象使用，也就是说：`List<String>`类并不是`List<Object>`类的子类。
- 为了表示各种泛型List的父类，我们需要使用**类型通配符**，类型通配符是一个问号（**?**），将一个问号作为类型实参传给List集合，写作：`List<?>`（意思是未知类型元素的List）。这个问号（**?**）被称为通配符，它的元素类型可以匹配任何类型。
- 使用**`List<?>`**这种形式是，即表明这个List集合可以是任何泛型List的父类，主要使用在方法声明中的参数类型。

4.7.3 类型通配符

- 使用List<?>这种形式是，即表明这个List集合可以是任何泛型List的父类，但是并不能把元素加入到其中，如下的代码将引起编译错误：

```
List<?> c = new ArrayList<String>();  
//下面的程序引起编译错误  
c.add(new Object());
```

- 但还有一种特殊的情形，我们不想这个List<?>是任何泛型List的父类，只想表示它是某一类泛型List的父类，此时我们需要设定类型通配符的上限！

4.7.3 类型通配符

设定类型通配符的上限

- 系统中存在一个抽象类Shape，现在我们希望传给List泛型类的类型参数必须是Shape类的子类，为了满足这种需求，Java泛型提供了被限制的泛型通配符。被限制的泛型通配符的如下表示：

List<? extends Shape>

- List<? extends Shape>，? 代表一个未知类型，但此处限定未知类型为Shape的子类，也可以是Shape本身，因此，称Shape为通配符? 的上限。
- Java泛型不仅允许在使用通配符形参时设定类型上限，也可以在定义类型形参时设定上限，在多个上限的情况下，用&分隔。

4.7.3 类型通配符

设定类型通配符的上限

- 形状抽象类Shape及其子类圆类Circle、长方形类Rectangle

```
// 定义一个抽象类Shape
public abstract class Shape
{
    public abstract void draw(Canvas c);
}
```

```
// 定义Shape的子类Circle
public class Circle extends Shape {
    // 实现画图方法，以打印字符串来模拟画图方法实现
    public void draw(Canvas c) {
        System.out.println("在画布" + c + "上画一个圆");
    }
}
```

UpperBound

```
// 定义Shape的子类Rectangle
public class Rectangle extends Shape {
    // 实现画图方法，以打印字符串来模拟画图方法实现
    public void draw(Canvas c) {
        System.out.println("把一个矩形画在画布" + c + "上");
    }
}
```

4.7.3 类型通配符

设定类型通配符的上限

- 有了形状抽象类Shape及其子类圆类Circle、长方形类Rectangle，现在我们要定义一个画布类用以一次性在画布上绘制多个图形。

```
public class Canvas {  
    // 同时在画布上绘制多个形状，使用被限制的泛型通配符  
    public void drawAll(List<? extends Shape> shapes) {  
        for (Shape s : shapes) {  
            s.draw(this);  
        }  
    }  
    public static void main(String[] args) {  
        List<Circle> circleList = new ArrayList<Circle>();  
        List<Rectangle> rectList = new ArrayList<>();  
        Canvas c = new Canvas();  
        c.drawAll(circleList);  
        c.drawAll(rectList);  
    }  
}
```

4.7.3 类型通配符

思考：设定类型通配符的上限

- 如果不设定类型通配符的上限，以下的两种实现分别有什么问题？

```
public void drawAll(List<Shape> shapes) {  
    for (Shape s : shapes) {  
        s.draw(this);  
    }  
}
```

```
public void drawAll(List<?> shapes) {  
    for (Object obj : shapes) {  
        Shape s = (Shape)obj;  
        s.draw(this);  
    }  
}
```

4.7.4 泛型方法

- 如果定义类、接口是没有使用类型形参，但定义方法时想自己定义类型形参，这也是可以的，JDK1.5还提供了泛型方法的支持。

- 泛型方法的语法格式为：

```
修饰符 <T, S> 返回值类型 方法名(形参列表) {  
    //方法体...  
}
```

- 泛型方法的方法签名比普通方法的方法签名多了类型形参声明，类型形参声明以尖括号括起来，多个类型形参之间以逗号（,）隔开，所有类型形参声明放在方法修饰符和方法返回值类型之间。
- 与类、接口中使用泛型参数不同的是，方法中的泛型参数无需显式传入实际类型参数，因为编译器根据实参推断类型形参的值。它通常推断出最直接的类型参数。

4.7.4 泛型方法 代码示例

```
public class GenericMethodTest {
    // 声明一个泛型方法，该泛型方法中带一个T类型形参，
    static <T> void fromArrayToCollection(T[] a, Collection<T> c) {
        for (T o : a)
            c.add(o);
    }
    public static void main(String[] args) {
        Object[] oa = new Object[100];
        Collection<Object> co = new ArrayList<>();
        // 下面代码中T代表Object类型
        fromArrayToCollection(oa, co);
        String[] sa = new String[100];
        Collection<String> cs = new ArrayList<>();
        // 下面代码中T代表String类型
        fromArrayToCollection(sa, cs);
        // 下面代码中T代表Object类型
        fromArrayToCollection(sa, co);
        Integer[] ia = new Integer[100];
        Float[] fa = new Float[100];
        Number[] na = new Number[100];
        Collection<Number> cn = new ArrayList<>();
        // 下面代码中T代表Number类型
        fromArrayToCollection(ia, cn);
        // 下面代码中T代表Number类型
        fromArrayToCollection(fa, cn);
        // 下面代码中T代表Number类型
        fromArrayToCollection(na, cn);
        // 下面代码中T代表Object类型
        fromArrayToCollection(na, co);
        // 下面代码中T代表String类型，但na是一个Number数组，
        // 因为Number既不是String类型，
        // 也不是它的子类，所以出现编译错误
        fromArrayToCollection(na, cs);
    }
}
```

GenericMethodTest

与类、接口中使用泛型参数不同的是，方法中的泛型参数无需显式传入实际类型参数，因为编译器根据实参推断类型形参的值。它通常推断出最直接的类型参数。

4.7.4 泛型方法

泛型方法和类型通配符

- 大多数时候都可以使用泛型方法来代替类型通配符。

```
public static <T> void copy(List<T> dest, List<? Extends T> src)  
public static <T,S extends T> void copy(List<T> dest, List<S> src)
```

- 泛型方法允许类型形参被用来表示方法的一个或多个参数之间的类型依赖关系，或者方法返回值与参数之间的类型依赖关系。如果没有这样的类型依赖关系，不应该使用泛型方法。

4.7.4 泛型方法

设定通配符的下限

```
class MyUtils {  
    // 下面dest集合元素类型必须与src集合元素类型相同，或是其父类  
    public static <T> T copy(Collection<? super T> dest, Collection<T> src) {  
        T last = null;  
        for (T ele : src) {  
            last = ele;  
            dest.add(ele);  
        }  
        return last;  
    }  
}
```

```
public class LowerBound {  
    public static void main(String[] args) {  
        List<Number> ln = new ArrayList<>();  
        List<Integer> li = new ArrayList<>();  
        li.add(5);  
        // 此处可准确的知道最后一个被复制的元素是Integer类型  
        // 与src集合元素的类型相同  
        Integer last = MyUtils.copy(ln, li); // ①  
        System.out.println(ln);  
    }  
}
```

LowerBound

4.7.5 擦除和转换

- 在严格的泛型代码里，带泛型声明的类总应该带着类型参数。但为了与老的Java代码保持一致，也允许在使用带泛型声明的类时不指定类型参数。如果没有为这个泛型类指定类型参数，则该类型参数被称作一个raw type（原始类型），默认是该声明该参数时指定的第一个上限类型。
- 当把一个具有泛型信息的对象赋给另一个没有泛型信息的变量时，则所有在尖括号之间的类型信息都被扔掉了。比如说一个List<String>类型被转换为List，则该List对集合元素的类型检查变成了成类型变量的上限（即Object），这种情况被为擦除。

4.7.5 擦除和转换

代码示例

- 右图所示的程序，在②处将a赋给一个不带泛型信息的变量b时，编译器就会丢失a对象的泛型信息，由于Apple类的类型参数的上限是Number类，所以编译器依然知道b的getSize()方法返回Number类型，但具体是哪个子类就不得而知了，因此③处的代码会引起编译错误。

```
class Apple<T extends Number> {
    T size;
    public Apple() {}
    public Apple(T size) {
        this.size = size;
    }
    public void setSize(T size) {
        this.size = size;
    }
    public T getSize() {
        return this.size;
    }
}

public class ErasureTest {
    public static void main(String[] args) {
        Apple<Integer> a = new Apple<>(6); // ①
        // a的getSize方法返回Integer对象
        Integer as = a.getSize();
        // 把a对象赋给Apple变量，丢失尖括号里的类型信息
        Apple b = a; // ②
        // b只知道size的类型是Number
        Number size1 = b.getSize();
        // 下面代码引起编译错误
        Integer size2 = b.getSize(); // ③
    }
}
```

ErasureTest

本章导读

- 4.1 继承 Inheritance
- 4.2 抽象类和抽象方法 Abstract Class & Abstract Method
- 4.3 内部类 Inner Class
- 4.4 接口 Interface
- 4.5 Lambda表达式 λ Expression
- 4.6 接口与抽象类 Interface & Abstract Class
- 4.7 大话泛型 Generic Programming
- ✓ 4.8 枚举类 Enumeration Class
- 4.9 作业及延伸

4.8 枚举类

- 4.8.1 枚举类入门
- 4.8.2 枚举类主要方法
- 4.8.3 枚举类的成员变量，方法和构造器

4.8 枚举类

Enumeration Class

- **Java5**新增了一个**enum**关键字，用以定义枚举类，与**class**和**interface**（接口，我们将在下一章进行学习）关键字的地位相同。
- 枚举类是一种特殊的类，它一样可以有自己的方法和属性，可以实现一个或者多个接口，也可以定义自己的构造器。一个**Java**源文件中最多只能定义一个**public**访问权限的枚举类，且该**Java**源文件也必须和该枚举类的类名相同。

4.8 枚举类入门

- 枚举类可以实现一个或多个接口，使用enum定义的枚举类默认继承了java.lang.Enum类，而不是继承Object类。
- 枚举类的所有实例必须在枚举类必须在第一行显式列出，否则这个枚举类将永远都不能产生实例。列出这些实例时系统会自动添加public static final修饰，无需程序员显式添加。
- 所有枚举类都提供了一个values方法，该方法可以很方便地遍历所有的枚举值。

4.8.1 枚举类入门

代码示例

```
public enum SeasonEnum
{
    // 在第一行列出4个枚举实例
    SPRING,SUMMER,FALL,WINTER;
}
```

```
public class EnumTest {
    public void judge(SeasonEnum s) {
        // switch语句里的表达式可以是枚举值
        switch (s) {
            case SPRING:
                System.out.println("春暖花开，正好踏青");
                break;
            case SUMMER:
                System.out.println("夏日炎炎，适合游泳");
                break;
            case FALL:
                System.out.println("秋高气爽，进补及时");
                break;
            case WINTER:
                System.out.println("冬日雪飘，围炉赏雪");
                break;
        }
    }

    public static void main(String[] args) {
        // 枚举类默认有一个values方法，返回该枚举类的所有实例
        for (SeasonEnum s : SeasonEnum.values()) {
            System.out.println(s);
        }
        // 使用枚举实例时，可通过EnumClass.variable形式来访问
        new EnumTest().judge(SeasonEnum.SPRING);
    }
}
```

EnumTest

4.8.2 枚举类的主要方法

- **int compareTo(E o)** : 用于指定枚举对象比较顺序，统一枚举实例只能与相同类型的枚举实例进行比较。
 - 如果该枚举对象位于指定枚举对象之后，则返回正整数；
 - 如果该枚举对象位于指定枚举对象之前，则返回负整数；
 - 否则返回零。
- **String toString()** : 返回枚举常量的名称。
- **Public static <T extends Enum<T>>T valueOf(Class<T> enumType, String name)**

用户返回指定name所对应的枚举值。（有关<T>泛型的内容，我们将在后续章节学习）

4.8.3 枚举类的成员变量，方法和构造器

- 枚举类也是一种类，只是它是一种比较特殊的类，因此它一样可以使用成员变量和方法。
- 枚举类的实例不能使用`new`创建。
- 枚举类通常应该设计成不可变类，也就说它的属性值不应该允许改变，这样会更安全，而且代码更加简洁。为此，我们应该将枚举类的属性都使用`private final`修饰。

4.8.3 枚举类的成员变量、方法和构造器

最佳实践代码示例

```
public enum Gender {  
    // 此处的枚举值必须调用对应构造器来创建  
    MALE("男"), FEMALE("女");  
    private final String name;  
    // 枚举类的构造器只能使用private修饰  
    private Gender(String name) {  
        this.name = name;  
    }  
    public String getName() {  
        return this.name;  
    }  
}
```

```
public class GenderTest {  
    public static void main(String[] args) {  
        // 通过Enum的valueOf()方法来获取指定枚举类的枚举值  
        Gender g = Enum.valueOf(Gender.class, "FEMALE");  
        // 直接访问枚举值的name实例变量  
        System.out.println(g + "代表:" + g.name);  
    }  
}
```

EnumClass

本章导读

- 4.1 继承 Inheritance
- 4.2 抽象类和抽象方法 Abstract Class & Abstract Method
- 4.3 内部类 Inner Class
- 4.4 接口 Interface
- 4.5 Lambda表达式 λ Expression
- 4.6 接口与抽象类 Interface & Abstract Class
- 4.7 大话泛型 Generic Programming
- 4.8 枚举类 Enumeration Class
- ✓ 4.9 作业及延伸

作业 1

- 请根据第3章和第4章所学内容，对涉及到的关键字及其用法进行总结：
- 要求使用自己组织语言，不能照抄书本以及课件，如果需要可以配上代码说明。
- 请至少包含如下关键字：
 - `final static abstract`（重点要求，需要总结可使用的类成员类型及其效果）
 - `private protected public default`
 - `class enum extends interface`

作业 2

- 思考

- 为什么外部类的访问权限 只能是public或者default；而内部类的访问权限，可以使用所有的访问修饰符？

作业 3

- 到目前为止，我们已经学习了Java语言对于面向对象所支持绝大部分特性，已经具备足够的能力去阅读JDK中的部分源代码！
- 代码阅读任务——BigInteger类
- 要求：
 1. 通过阅读源码，理解对于“大整数”的存储原理
 2. 理解加减乘除的基本操作以及“BigInteger”的各个构造器
 3. 梳理源码中各种学习过的知识点，并列举出源码中不理解点

作业 4

- 使用 `javac` 命令编译课件中内部类以及枚举类的示例代码（若未提供源码，请自行编写），观察编译后生成的 `class` 文件，并自行总结编译得到 `class` 文件的命名规则。



延伸

native关键字

- 我们在Object类的hashCode方法声明中，第一次遇到了native关键字，native方法称为本地方法。在java源程序中以关键字native声明，不提供函数体。其实现使用C/C++语言在另外的文件中编写，编写的规则遵循Java本地接口的规范Java native Interface (简称JNI)。
`public native int hashCode();`
- 除abstract以外，native可以与所有其它的java关键字连用。
- JNI编程入门：
<http://blog.csdn.net/huachao1001/article/details/53906237>



延伸 native关键字

- JNI实现native方法的步骤如下：
- 1. 编写带有native声明的方法的Java类，编译生成class文件
- 2. 使用javah编译上一步得到的class文件，生成后缀名为.h的文件
- 3. 使用C/C++实现native方法（需要包含上一步生成头文件以及JDK自带的jni.h文件）
- 4. 将实现native方法的cpp文件编译生成动态链接库
- 5. 在Java中用System类的loadLibrary.. ()方法或者Runtime类的loadLibrary()方法加载动态链接文件

延伸 类加载机制

- 当调用 java 命令运行某个 Java 程序时，该命令将会启动一个 **JVM 进程**，不管 Java 程序有多么复杂、该程序启动了多少个线程，它们都处于该 JVM 进程里。
- 当系统出现以下几种情况时，JVM 进程将被终止：
 - 程序运行到最后正常结束。
 - 程序运行到使用 `System.exit()` 或 `Runtime.getRuntime().exit()` 代码结束程序。
 - 程序执行过程中遇到未捕获的异常或错误而结束。
 - 程序所在平台强制结束了 JVM 进程。

延伸——类加载机制

类的加载

- 当程序主动使用某个类时，如果该类还未被加载到内存中，系统会通过加载、连接、初始化三个步骤来对该类进行初始化。
- 如果没有意外，JVM将会连续完成这三个步骤，所以有时也把这三个步骤统称为类加载或类初始化。
- 类加载指的是将类的class文件读入内存，并为之创建一个java.lang.Class对象，也就是说当程序使用任何类时，系统都会为之建立一个java.lang.Class对象。

延伸——类加载机制 类的连接

- 当类被加载之后，系统为之生成一个对应的Class对象，接着进入连接阶段，该阶段将会负责把类的二进制数据合并到JRE中。类连接又可分为验证、准备、解析三个阶段。
- native延伸部分，JNI最后编译得到的动态链接库，也是在连接阶段链接到JRE中

延伸——类加载机制 类的初始化

- 在类的初始化阶段，虚拟机负责对类进行初始化，主要就是对静态属性进行初始化。在Java类中对静态属性指定初始值有两种方式：
 - 声明静态属性时指定初始值；
 - 使用静态初始化块为静态属性指定初始值。

延伸 注解 Annotation

- 我们在之前的学习中，遇到了@override以及@FunctionalInterface两个奇怪的语句，这是Java 5新增的一项特性，称之为注解。
- 注解相当于一种标记，加了注解就等于给代码打上了某个标记。Javac编译器、IntelliJ IDEA之类的IDE以及其他一些注解处理工具（Annotation Processing Tool, APT）就可以利用它来完成各式各样的工作。

延伸 注解 Annotation

- 注解可以附加在包、类、字段、方法、方法参数以及局部变量上。
- 在各种软件开发工具和框架中，注解被广泛使用，比如JUnit工具会自动运行标记有@Test的单元测试方法，Spring MVC框架会将标有@Controller的类识别为控制器等等。
- 带有注解的代码被编译时，如果程序员明确指明需要的话，javac编译器可以将相关信息保存到class文件中，当JVM加载.class文件时，这些注解信息一并被装入内存。

延伸 注解 Annotation

- JDK中内置了一些直接可用的注解，主要集中于java.lang、java.lang.annotation和javax.annotation中。
 - **@Deprecated**：标记类的成员已经过时
 - **@SuppressWarnings**（“deprecation”）：忽略编译时的警告信息
 - **@Override**：要求子类必须覆盖基类的方法
 - **@Serializable**：指定某个类是可以序列化的
 - **@FunctionalInterface**：指定某个方法重写了基类的同名方法

延伸 注解 Annotation

- Java支持自定义注解，其声明关键字如下，实际就是一个使用@interface定义的接口，具体细节交给同学们课后自学。

```
public @interface MyTestAnnotation {  
    // code  
}
```