Java 应用程序设计
面向对象编程进阶

KevinW@OUC
wangxiaodong@ouc.edu.cn

中国海洋大学

September 21, 2017

参考书目

1. 张利国、刘伟 [编著], Java SE 应用程序设计, 北京理工大学出版社, 2007.10.

大纲

接下来…

包

继承

访问控制

方法重写

关键字 super

关键字 this

多态性

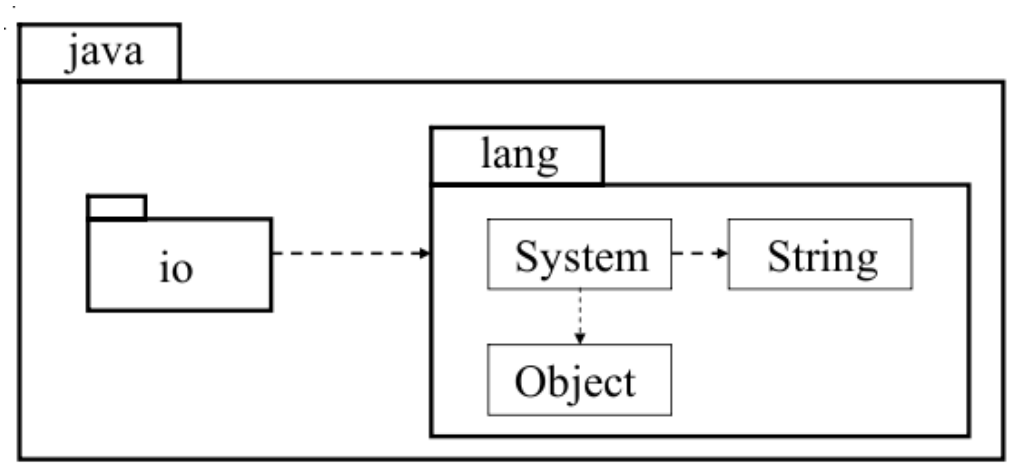
方法重载

关键字 static

关键字 final

包

为便于管理大型软件系统中数目众多的类，解决类的命名冲突问题以及进行访问控制，Java 引入包（package）机制，即将若干功能相关的类逻辑上分组打包到一起，提供类的多重类命名空间。



JDK API 中的常用包

包名	功能说明	包的含义
java.lang	Java 语言程序设计的基础类	language 的简写
java.awt	创建图形用户界面和绘制图形图像的相关类	抽象窗口工具集
java.util	集合、日期、国际化、各种实用工具	utility 的简写
java.io	可提供数据输入/输出相关功能的类	input/output 的简写
java.net	Java 网络编程的相关功能类	网络
java.sql	提供数据库操作的相关功能类	结构化查询语言的简写

包的创建

package 语句作为 Java 源文件的第一条语句，指明该文件中定义的类所在的包（若缺省该语句，则指定为无名包）。语法格式如下：

```
1 package pkg1[.pkg2[.pkg3...]];
```

CODE ▸ 创建包

```
1 package p1;
2 public class Test{
3   public void m1(){
4     System.out.println("In␣class␣Test,
5 ␣␣␣␣method␣m1␣is␣running!");
6   }
7 }
```

package 语句对所在源文件中定义的所有类型（包括接口、枚举、注解）均起作用。

包的创建

Java 编译器把包对应于文件系统的目录管理，package 语句中，用“.”来指明包（目录）的层次。如果在程序 Test.java 中已定义了包 p1，编译时采用如下方式：

```
1   > javac Test.java
```

则编译器会在当前目录下生成 Test.class 文件。
若在命令行下使用如下命令：

```
1   > java -d /home/xiaodong/work01 Test.java
```

“-d /home/xiaodong/work01”是传给 Java 编译器的参数，用于指定此次编译生成的.class 文件保存到该指定路径下，并且如果源文件中有 package 语句，则编译时会自动在目标路径下创建与包同名的目录 p1，再将生成的 Test.class 文件保存到该目录下。

导入包中的类

为使用定义在不同包中的 Java 类，需用 import 语句来引入所需要的类。语法格式：

```
1 import pkg1[.pkg2...].(classname|*);
```

CODE ◗ 导入和使用有名包中的类

```
1 import p1.Test; //or import p1.*;
2 public class TestPackage{
3   public static void main(String args[]){
4     Test t = new Test();
5     t.m1();
6   }
7 }
```

Java 包特性

一个类如果未声明为 public 的，则只能在其所在包中被使用，其他包中的类即使在源文件中使用 import 语句也无法引入它。可以不在源文件开头使用 import 语句导入要使用的有名包中的类，而是在程序代码中每次用到该类时都给出其完整的包层次，例如：

```
1  public class TestPackage{
2    public static void main(String args[]){
3      p1.Test t = new p1.Test();
4      t.m1();
5    }
6  }
```

练习

练习使用 Java 包。

接下来…

- 包
- 继承
- 访问控制
- 方法重写
- 关键字 super
- 关键字 this
- 多态性
- 方法重载
- 关键字 static
- 关键字 final

什么是继承

继承（Inheritance）是面向对象编程的核心机制之一，其本质是在已有类型基础之上进行扩充或改造，得到新的数据类型，以满足新的需要。

根据需要定义 Java 类描述“人”和“学生”信息：

CODE ▸ Class Person

```
1  public class Person {
2    public String name;
3    public int age;
4    public Date birthDate;
5    public String getInfo() {...}
6  }
```

什么是继承

CODE ◗ Class Student

```
1  public class Student {
2    public String name;
3    public int age;
4    public Date birthDate;
5    public String school;
6    public String getInfo() {...}
7  }
```

通过继承，简化 Student 类的定义：

CODE ◗ Class Student extends Person

```
1  public class Student extends Person {
2    public String school;
3  }
```

继承

Java 类声明语法格式：

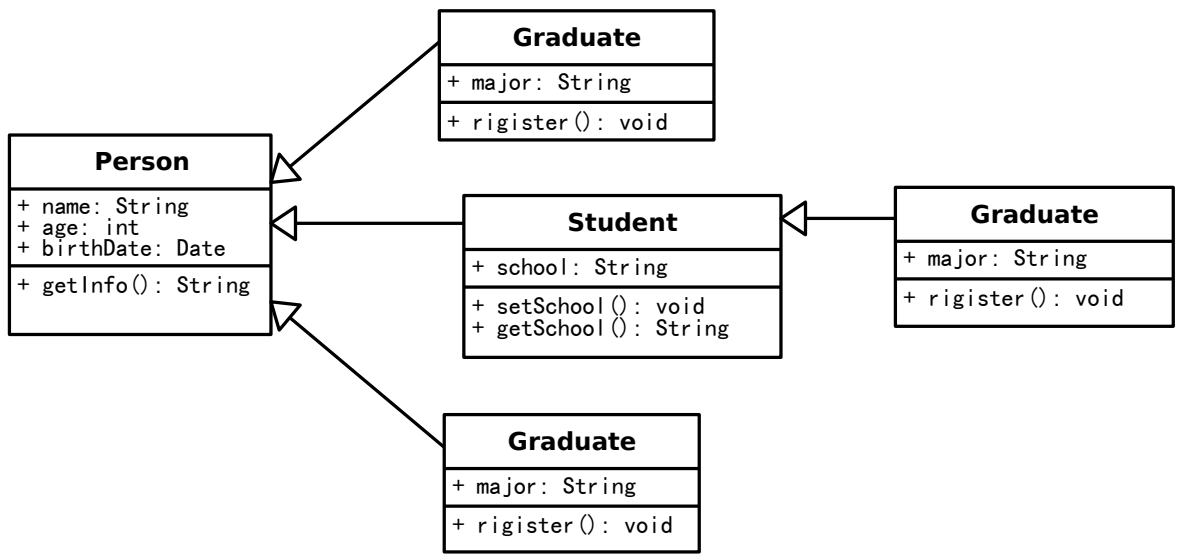
```
1   [< 修饰符 >] class < 类名 > [extends < 父类名 >] {
2     [< 属性声明 >]
3     [< 构造方法声明 >]
4     [< 方法声明 >]
5   }
```

Object 类是所有 Java 类的最高层父类, 如果在类的声明中未使用 extends 关键字指明其父类，则默认父类为 Object 类。

继承

Java 只支持单继承，不允许多重继承。

- 一个子类只能有一个父类；
- 一个父类可以派生出多个子类。



练习

1. 按照上述的类结构及继承关系, 定义实现 Java 类 Person、Student 和 Graduate，所有的类、属性及方法均定义为 public 的。
2. 编写 Java 程序，创建 Graduate 类型对象并测试访问其所有继承来的方法和属性。
3. 将父类 Person、Student 中的部分属性/方法改为 private 的，然后重复 2 中的测试。

类之间的关系

依赖关系 一个类的方法中使用到另一个类的对象（uses-a）[1]。

聚合关系 一个类的对象包含（通过属性引用）了另一个类的对象（has-a）[2]。

泛化关系 一般化关系（is-a），表示类之间的继承关系、类和接口之间的实现关系以及接口之间的继承关系。

[1] 车能够装载货物，车的装载功能（load() 方法）对货物（goods）有依赖。
[2] 车有发动机、车轮等，Car 对象是由 Engine 等对象构成的。

接下来…

- 包
- 继承
- 访问控制
- 方法重写
- 关键字 super
- 关键字 this
- 多态性
- 方法重载
- 关键字 static
- 关键字 final

访问控制

访问控制是指对 Java 类或类中成员的操作进行限制，即规定其在多大的范围内可以被直接访问。

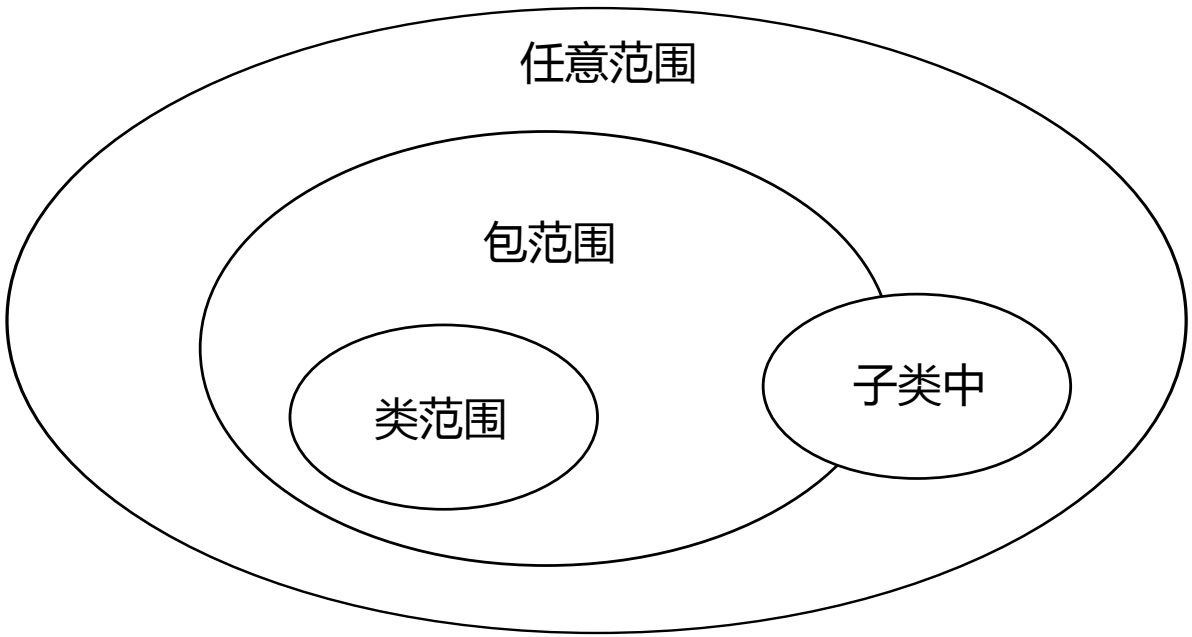
❖ 类的访问控制

在声明 Java 类时可以在 class 关键字前使用 public 来修饰，也可以不使用该修饰符。public 的类可在任意场合被引入和使用，而非 public 的类只能在其所在包中被使用。

❖ 类中成员的访问控制

修饰符/作用范围	同一个类	同一个包	子类	任何地方
public	yes	yes	yes	yes
protected	yes	yes	yes	no
无修饰符	yes	yes	no	no
private	yes	no	no	no

访问控制



访问控制注意的一些问题

- 一般不提倡将属性声明为 public 的，而构造方法和需要外界直接调用的普通方法则适合声明为 public 的。
- 在位于不同的包内，必须是子类的对象才可以直接访问其父类的 protected 成员，而父类自身的对象反而不能访问其所在类中声明的 protected 成员。
- 所谓“访问控制”只是控制对 Java 类或类中成员的直接访问，而间接访问是不做控制的，也不该进行控制。

访问控制 protected

CODE ◗ A.java

```
1  package p1;
2  public class A {
3    public int m = 5;
4    protected int n = 6;
5  }
```

CODE ◗ B.java

```
1  package p2;
2  import p1.A;
3  public class B extends A{
4    public void mb() {
5      m = m + 1;
6      n = n * 2;
7    }
8    public static void main(String[] args) {
9      B b = new B();
10     b.m = 7; // 合法
11     b.n = 8; // 合法
12     A a = new A();
13     a.m = 9 // 合法
14     a.n = 10 // 非法
15   }
16 }
```

接下来…

- 包
- 继承
- 访问控制
- 方法重写
- 关键字 super
- 关键字 this
- 多态性
- 方法重载
- 关键字 static
- 关键字 final

方法重写

❖ 什么是方法重写

在子类中可以根据需要对从父类中继承来的方法进行重新定义，此称方法重写（Override）或覆盖。

❖ 语法规则

- 重写方法必须和被重写方法具有相同的方法名称、参数列表和返回值类型；
- 重写方法不能使用比被重写方法更严格的访问权限；
- 重写方法不允许声明抛出比被重写方法范围更大的异常类型。

方法重写示例

CODE ◗ 方法重写示例 A

```
1  public class Person {
2    String name;
3    int age;
4    public String getInfo() {
5      return "Name:"+ name + "\t" +"age:"+ age;
6    }
7  }
```

```
1  public class Student extends Person {
2    private String school;
3    public void setSchool(String scholl) {
4      this.school = school;
5    }
6    public String getSchool(){
7      return school;
8    }
9    public String getInfo() {
10     return "Name:"+ name + "\tAge:"+ age + "\tSchool:" + school;
11   }
12 }
```

方法重写示例

CODE ◗ 方法重写示例 B

```
1 public class Parent {
2   public void method1() {...}
3 }
```

```
1 public class Child extends Parent {
2   private void method1() {} //非法，权限更严格
3 }
```

```
1 public class UseBoth {
2   public void doOtherThing() {
3     Parent p1 = new Parent();
4     Child p2 = new Child();
5     p1.method1();
6     p2.method1();
7   }
8 }
```

同名属性

CODE ◗ 同名属性

```
1  public class Person {
2    int age = 5;
3    public int getAge() {
4      return age;
5    }
6    public int getInfo() {
7      return age;
8    }
9  }
```

```
1  public class Student extends Person {
2    int age = 6;
3    public int getAge() {
4      return age;
5    }
6  }
```

同名属性

```
1  public class Test {
2    public static void main(String args[]) {
3      Person p = new Person();
4      System.out.println(p.getAge());
5      Student s = new Student();
6      System.out.println(s.age);
7      System.out.println(s.getAge());
8      System.out.println(s.getInfo());
9    }
10 }
```

输出结果：

output

```
5
6
6
5
```

同名属性

❖ 对上述 Student 对象同名属性的几点说明

1. 以“对象名.属性名”方式直接访问时，使用的是子类中添加的属性 age；
2. 调用子类添加或者重写的方法时，方法中使用的是子类定义的属性 age；
3. 调用父类中定义的方法时，方法中使用的是父类中的属性 age。

可以理解为“层次优先”[3]；不提倡使用同名属性。

[3]在哪个层次中的代码，就优先使用该层次类中定义的属性。

接下来…

- 包
- 继承
- 访问控制
- 方法重写
- 关键字 super
- 关键字 this
- 多态性
- 方法重载
- 关键字 static
- 关键字 final

关键字 super

在存在命名冲突（子类中存在方法重写或添加同名属性）的情况下，子类中的代码将自动使用子类中的同名属性或重写后的方法。当然也可以在子类中使用关键字 super 引用父类中的成分：

访问父类中定义的属性

super.< 属性名 >

调用父类中定义的成员方法

super.< 方法名 >(< 实参列表 >)

子类构造方法中调用父类的构造方法

super(< 实参列表 >)

super 的追溯不仅限于直接父类，先从直接父类开始查找，如果找不到则逐层上溯，一旦在某个层次父类中找到匹配成员即停止追溯并使用该成员。

super 用法示例

CODE ▸ super A

```
1   class Animal {
2     protected int i = 1; //用于测试同名属性，无现实含义
3   }

5   class Person extends Animal {
6     protected int i = 2; //用于测试同名属性，无现实含义
7     private String name = "Tom";
8     private int age = 9;
9     public String getInfo() {
10      return "Name:" + name + "\tAge:" + age;
11    }
12    public void testI() {
13      System.out.println(super.i);
14      System.out.println(i);
15    }
16  }
```

super 用法示例

CODE ▸ super B

```
1   class Student extends Person {
2     private int i = 3;
3     private String school = "THU";
4     public String getInfo() {  //重写方法
5       return super.getInfo() + "\tSchool:" + school;
6     }
7     public void testI() {   //重写方法
8       System.out.println(super.i);
9       System.out.println(i);
10    }
11  }
12  public class Test {
13    public static void main(String args[]) {
14      Person p = new Person();
15      System.out.println(p.getInfo());
16      p.testI();
17      Student s = new Student();
18      System.out.println(s.getInfo());
19      s.testI();
20    }
21  }
```

super 用法示例

上述代码的输出结果：

output

```
Name:Tom Age:9
1
2
Name:Tom Age:9 School:THU
2
3
```

练习

1. 编写应用程序，实现 Java 方法重写并测试。
2. 编写应用程序，测试 super 关键字用法。

接下来…

- 包
- 继承
- 访问控制
- 方法重写
- 关键字 super
- 关键字 this
- 多态性
- 方法重载
- 关键字 static
- 关键字 final

关键字 this

在 Java 方法中，不但可以直接使用方法的局部变量，也可以使用调用该方法的对象。
为解决可能出现的命名冲突，Java 语言引入 this 关键字来标明方法的当前对象。分为两种情况：

- 在普通方法中，关键字 this 代表方法的调用者，即本次调用了该方法的对象；
- 在构造方法中，关键字 this 代表该方法本次运行所创建的那个新对象。

this 作为一个特殊的引用类型变量，可以通过“this. 成员”的方式访问其引用的当前对象的属性和方法。

关键字 this

CODE this 用法示例

```
1  public class MyDate {
2    private int day = 17;
3    private int month = 2;
4
5    public MyDate(int day, int month) {
6      this.day = day; // A
7      this.month = month;
8    }
9    ... // Some methods
10
11   public void setAll(int day, int month) {
12     this.setMonth(month); // B
13     this.setDay(day);
14   }
15 }
```

关键字 this

❖ 关于 this 的归纳说明

1. 在 Java 方法中直接给出变量名而不是“对象名.变量名”的方式访问一个变量，系统首先尝试作为局部变量来处理；如果方法中不存在该名字的局部变量，才会到方法当前对象的成员变量中查找。
2. 在 Java 方法中直接调用一个方法而不指定其调用者时，则默认调用者为当前对象 this。

接下来…

- 包
- 继承
- 访问控制
- 方法重写
- 关键字 super
- 关键字 this
- 多态性
- 方法重载
- 关键字 static
- 关键字 final

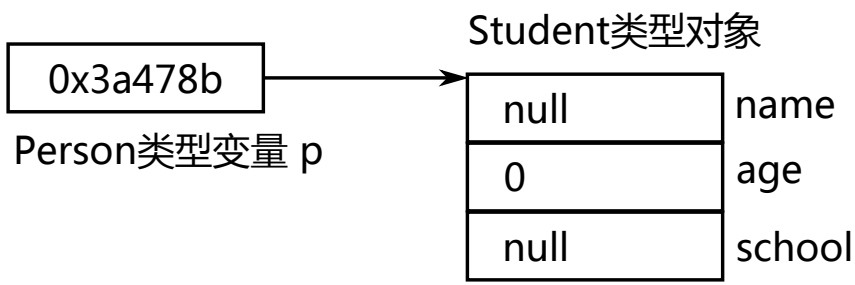
何为多态？

多态——在 Java 中，子类的对象可以替代父类的对象使用。

❖ Java 引用变量与所引用对象间的类型匹配关系

- 一个对象只能属于一种确定的数据类型，该类型自对象创建直至销毁不能改变。
- 一个引用类型变量可能引用（指向）多种不同类型的对象——既可以引用其声明类型的对象，也可以引用其声明类型的子类的对象。

```
1 Person p = new Student(); //Student 是 Person 的子类
```



多态用法示例

```
1   public class Person {...}
2
3   public class Student extends Person {
4     private String school;
5
6     public void setSchool(String school) {
7       this.school = school;
8     }
9     public String getSchool() {
10      return school;
11    }
12    public String getInfo() { //重写方法
13      return super.getInfo() + "\tSchool:␣" + school;
14    }
15  }
16
17  public class Test {
18    public static void main(String[] args) {
19      Person p = new Student();
20      System.out.println(p.getInfo());
21    }
22  }
```

多态性

引用类型数组元素相当于引用类型变量，多态性也同样适用：

```
1 Person[] p = new Person[3];
2 p[0] = new Student(); //假定 Student 类继承了 Person 类
3 p[1] = new Person();
4 p[2] = new Graduate(); //假定 Graduate 类继承了 Student 类
```

一个引用类型变量如果声明为父类的类型，但实际引用的是子类对象，那么该变量就不能再访问子类中添加的属性和方法。

```
1 Student m = new Student();
2 m.setSchool("pku"); //合法
3 Person e = new Student();
4 e.school = "pku"; //非法
5 e.setSchool("pku"); //非法
```

多态性的意义

```
1  public class Test {
2    public show(Person p) {
3      System.out.println(p.getInfo());
4    }

6    public static void main(String[] args) {
7      Test t = new Test();
8      Person p = new Person();
9      t.show(p);
10     Student s = new Student();
11     t.show(s);
12   }
13 }
```

show() 方法既可以处理 Person 类型的数据，又可以处理 Student 类型的数据，乃至未来定义的任何 Person 子类类型的数据，即不必为相关的每种类型单独声明一个处理方法，提高了代码的通用性。

虚方法调用

一个引用类型的变量如果声明为父类的类型，但实际引用的是子类对象，则该变量就不能再访问子类中添加的属性和方法。

但如果此时调用的是父类中声明过、且在子类中重写过的方法，情况如何？

虚方法调用示例

CODE ◗ Book.java

```
1  public class Book {
2    private String name;
3    private double price;
4
5    public void setName(String name) {
6    this.name = name;
7    }
8    public String getName() {
9      return name;
10   }
11   public void setPrice(double price) {
12     this.price = name;
13   }
14   public double getPrice() {
15     return price;
16   }
17   public void show() {
18     System.out.println("Book␣name:" + name + "\nprice" + price);
19   }
20 }
```

虚方法调用示例

CODE ▸ Novel.java

```
1  public class Novel extends Book {
2    private String author;
3
4    public void setAuthor(String author) {
5    this.author = author;
6    }
7    public String getAuthor() {
8      return author;
9    }
10   public void show() {
11     super.show();
12     System.out.println("Author:" + author);
13   }
14 }
```

虚方法调用示例

CODE ▸ Test.java

```
1  public void class Test {
2    public static void main(String[] args) {
3      Test t = new Test();
4      Book b = new Book();
5      b.setName("English␣Language");
6      b.setPrice(43);
7      t.process(b);
8      Novel v = new Novel();
9      v.setName("Great␣Expectations");
10     v.setPrice(35.48);
11     v.setAuthor("Charles␣Dickens");
12     t.process(v);
13   }
14   public void process(Book b) {
15     b.show();
16   }
17 }
```

虚方法调用示例

程序输出结果：

output

```
BookName:English Language
Price:43.0
BookName:Great Eespectations
Price:35.48
Author:Charles Dickens
```

从运行结果可以看出，系统根据运行时对象的真正类型来确定具体调用哪一个方法，这一机制被称为**虚方法调用**（Virtual Method Invocation）。

对象造型

有时我们可能需要恢复一个对象的本来面目，以发挥其全部潜力。引用类型数据值之间的强制类型转换称为造型（Casting），具体规则如下：

1. 从子类到父类的类型转换可以自动进行；

```
1 Person p = new Student();
```

2. 在多态的情况下，从父类到子类的类型转换必须通过造型（强制类型转换）实现；

```
1 Person p1 = new Student();
2 Student s1 = (Student)p1; //合法
3 Person p2 = new Person();
4 Student s2 = (Student)p2; //非法
```

3. 无继承关系的引用类型间的转换是非法的；

```
1 String s = "Hello World!";
2 Person p = (Person)s; //非法
```

对象造型示例

```
1  public class Test {
2    public void cast(Person p) {
3      System.out.println(p.getSchool()); // 非法
4      Student stu = (Student)p; // 非法
5      System.out.println(stu.getSchool());
6    }

8    public static void main(String[] args) {
9      Test t = new Test();
10     Student s = new Student();
11     t.cast(s);
12   }
13 }
```

instanceof 运算符

如果运算符 instanceof 左侧的变量当前时刻所引用的对象的真正类型是其右侧给出的类型、或者是其子类，则整个表达式的结果为 true。

```
1  class Person { --- }
2  class Student extends Person { --- }
3
4  public class Tool {
5    public void distribute(Person p) {
6      if (p instanceof Student) {
7        System.out.println("处理␣Student␣类型及其子类类型对象");
8      } else {
9        System.out.println("处理␣Person␣类型及其子类类型对象");
10     }
11   }
12 }
```

```
1  public class Test() {
2    public static void main(String[] args) {
3      Tool t = new Tool();
4      Student s = new Student();
5      t.distribute(t);
6    }
7  }
```

协变返回类型

协变返回类型——从 Java SE5.0 开始引入，允许重写方法时修改其返回值的类型，但必须是重写前方法返回值类型的**子类或实现类类型**[4]。

```
1  class A {
2    public Person getAssistor() {
3      Person p = new Person();
4      //---
5      return p;
6    }
7  }
8  class B extends A {
9    public Student getAssistor() { //重写方法时改变了返回值类型
10     Student s = new Student();
11     s.setName("Nancy");
12     s.setAge(18);
13     s.setSchool("THU");
14     return s;
15   }
16 }
```

[4]实现类后续课程讲述。

协变返回类型

CODE ◗ 如何不采用协变返回类型实现上述效果

```
1  class B extends A {
2    public Person getAssistor() {
3      Student s = new Student();
4      s.setName("Nancy");
5      s.setAge(18);
6      s.setSchool("THU");
7      return s; //合法，基于多态机制返回子类的实例
8    }
9  }
10 public class TestCovarianReturnType {
11   public static void main(String[] args) {
12     B b = new B();
13     Student stu = (Student)b.getAssistor();
14     System.out.println(stu.getSchool());
15   }
16 }
```

当重写方法中实际返回的是原返回类型的子类类型数据时，使用协变返回模式不需要再进行强制类型转换就可以访问其子类中添加的成员。

接下来…

- 包
- 继承
- 访问控制
- 方法重写
- 关键字 super
- 关键字 this
- 多态性
- 方法重载
- 关键字 static
- 关键字 final

什么是方法重载

在一个类中存在多个同名方法的情况称为**方法重载**（Overload），**重载方法参数列表必须不同**。

```
1  class Tool {
2    public void display(int i) {
3      System.out.println("输出整数:" + i);
4    }
5    public void display(double d) {
6      System.out.println("输出符点数:" + d);
7    }
8    public void display(String s) {
9      System.out.println("输出文本:" + s);
10   }
11 }
12 public class TestOverLoad {
13   public static void main(String args[]){
14     Tool t = new Tool();
15     t.display(3);
16     t.display(3.14);
17     t.display("Hello,␣你好!");
18   }
19 }
```

构造方法重载

与普通方法的重载类似，构造方法也可以重载。

```
1  public class Person {
2    private String name;
3    private int age;
4    public Person() {}
5    public Person(String name) {
6      this.name = name;
7    }
8    public Person(int age) {
9      this.age = age;
10   }
11   public Person(String name, int age) {
12     this.name = name;
13     this.age = age;
14   }
15   //其他成分
16 }
```

使用 this 调用重载构造方法

可以在构造方法的第一行使用关键字 this 调用其它（重载的）构造方法。

```
1  public class Person{
2    private String name;
3    private int age;
4    public Person(String name,int age) {
5      this.name = name;
6      this.age = age;
7    }
8    public Person(String name) {
9      this(name,18);
10   }
11   public Person(int age) {
12     this("Anonymous",age);
13   }
14   //---
15 }
```

注意： *关键字 this 的此种用法只能用在构造方法中，且 this() 语句如果出现必须位于方法体中代码的第一行。*

深究对象构造/初始化

在 Java 类的构造方法中一定直接或间接地调用了其父类的构造方法（Object 类除外）。

1. 在子类的构造方法中可使用 super 语句调用父类的构造方法，其格式为 super(< 实参列表 >)。
2. 如果子类的构造方法中既没有显式地调用父类构造方法，也没有使用 this 关键字调用同一个类的其他重载构造方法，则系统会默认调用父类无参数的构造方法，其格式为 super()。
3. 如果子类构造方法中既未显式调用父类构造方法，而父类中又没有无参的构造方法，则编译出错。

super 调用构造方法

CODE ▸ Person.java

```
1  public class Person {
2    private String name;
3    private int age;
4    public Person(String name, int age) {
5      this.name = name;
6      this.age = age;
7    }
8    public Person(String name) {
9      this(name,18);
10   }
11   public Person(int age) {
12     this("Anonymous",age);
13   }
14   public void showInfo() {
15     System.out.println("Name:" + name + "\tage:" + age);
16   }
17 }
```

super 调用构造方法

CODE ◗ Student.java

```
1  public class Student extends Person {
2    private String school;
3    public Student(String name, int age, String school) {
4      super(name, age); // 显式调用父类有参构造方法
5      this.school = school;
6    }
7    public Student(String name, String school) {
8      super(name); // 显式调用父类有参构造方法
9      this.school = school;
10   }
11   public Student(String name, int age) {
12     this(name, age, null);
13   }
14   public Student(String school) { //编译出错
15     //super(); // 隐式调用父类有参构造方法，则自动调用父类无参构造方法
16     this.school = school;
17   }
18 }
```

对象构造/初始化细节

第一阶段 为新建对象的实例变量分配存储空间并进行默认初始化。

第二阶段 按下述步骤继续初始化实例变量：

1. 绑定构造方法参数；
2. 如有 this() 调用，则调用相应的重载构造方法然后跳转到步骤 5；
3. 显式或隐式追溯调用父类的构造方法（Object 类除外）；
4. 进行实例变量的显式初始化操作；
5. 执行当前构造方法的方法体中其余的语句。

练习

编写应用程序，测试练习如下内容：

1. Java 对象造型；
2. 协变返回类型；
3. 方法重载；
4. this 调用重载构造方法；
5. super 调用父类构造方法；

接下来…

- 包
- 继承
- 访问控制
- 方法重写
- 关键字 super
- 关键字 this
- 多态性
- 方法重载
- 关键字 static
- 关键字 final

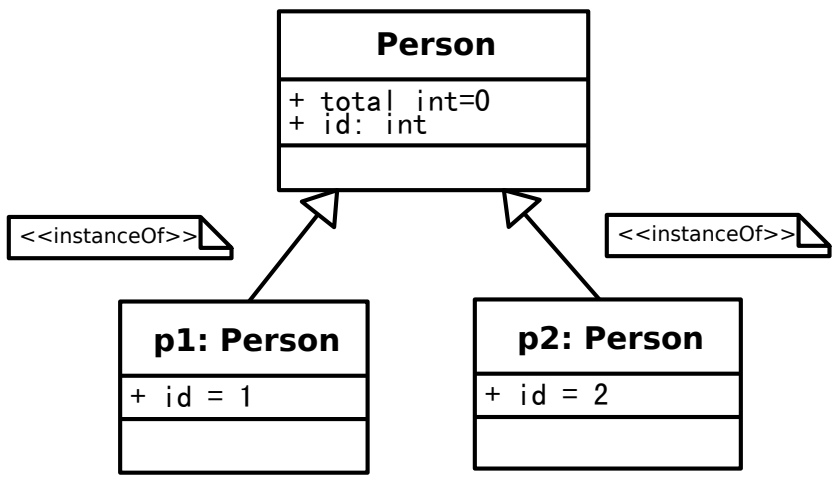
关键字 static

- 在 Java 类中声明属性、方法和内部类时，可使用关键字 static 作为修饰符。
- static 标记的属性或方法由整个类（所有实例）共享，如访问控制权限允许，可不必创建该类对象而直接用类名加“.”调用。
- static 成员也称“类成员”或“静态成员”，如“类属性”、“类变量”、“类方法”、“静态方法”等。

static 属性

static 属性由其所在类（包括该类所有的实例）共享，而非 static 属性则必须依赖具体/特定的对象（实例）而存在。

```
1  public class Person {
2    private int id;
3    public static int total = 0;
4    public Person() {
5      total++;
6      id = total;
7    }
8  }
```



static 属性

```
1  public class Person {
2    private int id;
3    public static int total = 0;
4    public Person() {
5      total++;
6      id = total;
7    }
8  }
```

```
1  public class Test {
2    public static void main(String args[]) {
3      Person.total = 100;
4      System.out.println(Person.total);
5      Person p1 = new Person();
6      Person p2 = new Person();
7      System.out.println(Person.total);
8    }
9  }
```

输出结果？

static 属性

```
1  public class Person {
2    private int id;
3    public static int total = 0;
4    public Person() {
5      total++;
6      id = total;
7    }
8  }
```

```
1  public class Test {
2    public static void main(String args[]) {
3      Person.total = 100;
4      System.out.println(Person.total);
5      Person p1 = new Person();
6      Person p2 = new Person();
7      System.out.println(Person.total);
8    }
9  }
```

输出结果？100 102

static 方法

```
1  public class Person {
2    private int id;
3    private static int total = 0;
4    public Person() {
5      total++;
6      id = total;
7    }
8    public int getId(){
9      return id;
10   }
11   public static int getTotalPerson() {
12     return total;
13   }
14 }
```

```
1  public class Test {
2    public static void main(String[] args) {
3      System.out.println(Person.getTotalPerson());
4      Person p1 = new Person();
5      System.out.println(p1.getTotalPerson());
6      System.out.println(Person.getTotalPerson());
7    }
8  }
```

static 方法

CODE ◗ static 方法对非 static 成员的访问

```
1  public class Test {
2    private i = 5;
3    public void m1() {
4      i++;
5    }
6    public void m2() {
7      m1(); // 合法，等价于 this.m1()
8    }
9    public static int m3() {
10     i++; // 非法
11   }
12   public static void main(String[] args) {
13     m2(); // 非法
14     m3(); // 合法，等价于 Test.m3()
15   }
16 }
```

上述代码会报“无法从静态上下文中引用非静态变量 i/方法 m2()”的编译错误。如果确实要在 static 方法中调用其所在类的非 static 成员，如何做？

static 方法

CODE ◗ static 方法对非 static 成员的访问

```
1  public class Test {
2    private i = 5;
3    public void m1() {
4      i++;
5    }
6    public void m2() {
7      m1(); // 合法，等价于 this.m1()
8    }
9    public static int m3() {
10     i++; // 非法
11   }
12   public static void main(String[] args) {
13     m2(); // 非法
14     m3(); // 合法，等价于 Test.m3()
15   }
16 }
```

上述代码会报“无法从静态上下文中引用非静态变量 i/方法 m2()”的编译错误。如果确实要在 static 方法中调用其所在类的非 static 成员，如何做？

应首先创建一个该类的对象，通过该对象来访问其非 static 成员。

static 初始化块

在类的定义体中，方法的外部可包含 static 语句块，static 块仅在其所属的类被载入时执行一次，通常用于初始化化 static（类）属性。

CODE ▸ Person.java

```
1 public class Person {
2   public static int total;
3   static {
4     total = 100;
5     System.out.println("in␣static␣block!");
6   }
7 }
```

CODE ▸ Test.java

```
1 public class Test {
2   public static void main(String[] args) {
3     System.out.println("total␣=␣"+ Person.total);
4     System.out.println("total␣=␣"+ Person.total);
5   }
6 }
```

输出结果？

static 初始化块

在类的定义体中，方法的外部可包含 static 语句块，static 块仅在其所属的类被载入时执行一次，通常用于初始化化 static（类）属性。

CODE ▸ Person.java

```
1 public class Person {
2   public static int total;
3   static {
4     total = 100;
5     System.out.println("in␣static␣block!");
6   }
7 }
```

CODE ▸ Test.java

```
1 public class Test {
2   public static void main(String[] args) {
3     System.out.println("total␣=␣"+ Person.total);
4     System.out.println("total␣=␣"+ Person.total);
5   }
6 }
```

输出结果？ in static block! total = 100 total = 100

非 static 初始化块

非 static 的初始化块在创建对象时被自动调用。

```
1  class A {
2    private int i = 5;
3    {
4      System.out.println("创建新对象---");
5    }
6    public A() {}
7    public A(int a) {
8      System.out.println("开始执行构造方法体中语句");
9      i = a;
10     System.out.println("构造方法体中语句执行完毕");
11   }
12 }
13 public class Test {
14   public static void main(String[] args) {
15     new A();
16     new A(3);
17   }
18 }
```

output

```
创建新对象---
创建新对象---
开始执行构造方法体中语句
构造方法体中语句执行完毕
```

静态导入

静态导入用于在一个类中导入其他类或接口中的 static 成员，语法格式：

import static < 包路径 >.< 类名 >.*

或：

import static < 包路径 >.< 类名 >.< 静态成员名 >

CODE ◗ 应用示例

```
1 import static java.lang.Math.*;
2 public class Test {
3   public static void main(String[] args) {
4     double d = sin(PI * 0.45);
5     System.out.println(d);
6   }
7 }
```

Singleton 设计模式

所谓“模式”就是被验证为有效的常规问题的典型解决方案。

设计模式（Design Pattern）在面向对象分析设计和软件开发中占有重要地位，好的设计模式可以使我们更加方便的重用已有的成功设计和体系结构，进而极大的提高代码的重用性和可维护性。

Singleton 设计模式

```
1  public class Single {
2    private String name;
3    private static Single onlyone = new Single();
4    private Single() {}
5    public void setName(String name) {
6      this.name = name;
7    }
8    public String getName(){
9      return name;
10   }
11   public static Single getSingle() {
12     return onlyone;
13   }
14 }
```

```
1  public class TestSingle {
2    public static void m1() {
3      Single s2 = Single.getSingle();
4      System.out.println(s2.getName());
5    }
6    public static void main(String args[]) {
7      Single s1 = Single.getSingle();
8      s1.setName("Tom");
9      m1();
10   }
11 }
```

Singleton 设计模式

```
1  public class Single {
2    private String name;
3    private static Single onlyone = new Single();
4    private Single() {}
5    public void setName(String name) {
6      this.name = name;
7    }
8    public String getName(){
9      return name;
10   }
11   public static Single getSingle() {
12     return onlyone;
13   }
14 }
```

```
1  public class TestSingle {
2    public static void m1() {
3      Single s2 = Single.getSingle();
4      System.out.println(s2.getName());
5    }
6    public static void main(String args[]) {
7      Single s1 = Single.getSingle();
8      s1.setName("Tom");
9      m1();
10   }
11 }
```

程序输出结果：Tom

Singleton 代码的特点

1. 使用静态属性 onlyone 来引用一个“全局性”的 Single 实例；
2. 将构造方法设置为 private 的，这样在外界将不能再使用 new 关键字来创建该类的新实例；
3. 提供 public static 的方法 getSingle() 以使外界能够获取该类的实例，达到全局可见的效果。

❖ 这样定义 Single 类的结果

在任何使用到 Single 类的 Java 程序中（这里指的是一次运行中），将确保只有一个 Single 类的实例存在。Java 语法规则的这种运用方式被称为“Singleton 设计模式”，也称“单子模式”或“单态模式”。

接下来…

- 包
- 继承
- 访问控制
- 方法重写
- 关键字 super
- 关键字 this
- 多态性
- 方法重载
- 关键字 static
- 关键字 final

关键字 final

在声明 Java 类、变量和方法时可以使用关键字 final 来修饰，以使其具有“终态”的特性：

1. final 标记的类不能被继承；
2. final 标记的方法不能被子类重写；
3. final 标记的变量（成员变量或局部变量）即成为常量，只能赋值一次；
4. final 标记的成员变量必须在声明的同时或在每个构造方法中显式赋值，然后才能使用；
5. final 不允许用于修饰构造方法、抽象类以及抽象方法。

关键字 final 应用举例

```
1  public final class Test {
2    public static int totalNumber = 5;
3    public final int id;
4    public Test() {
5      id = ++totalNumber; // 赋值一次
6    }
7    public static void main(String[] args) {
8      Test t = new Test();
9      System.out.println(t.id);
10     final int i = 10;
11     final int j;
12     j = 20;
13     j = 30; //非法
14   }
15 }
```

练习

练习上述有关例程，体会并掌握 static/final 关键字的用法。

THE END

wxd2870@163.com