Java 应用与开发
面向对象编程进阶 PART2

王晓东
wangxiaodong@ouc.edu.cn

中国海洋大学

September 30, 2018

学习目标

1. 理解多态和虚方法调用的概念，掌握其用法
2. 掌握方法重载的方法
3. 掌握 static 属性、方法和初始化块的用法
4. 了解设计模式，掌握单例设计模式
5. 掌握 final 关键字的概念和使用方法

大纲

接下来…

多态性

方法重载

关键字 static

关键字 final

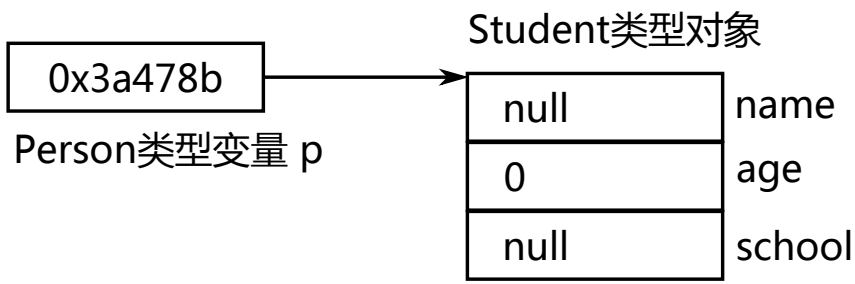
什么是多态？

在 Java 中，子类的对象可以替代父类的对象使用称为**多态**。

❖ 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
```



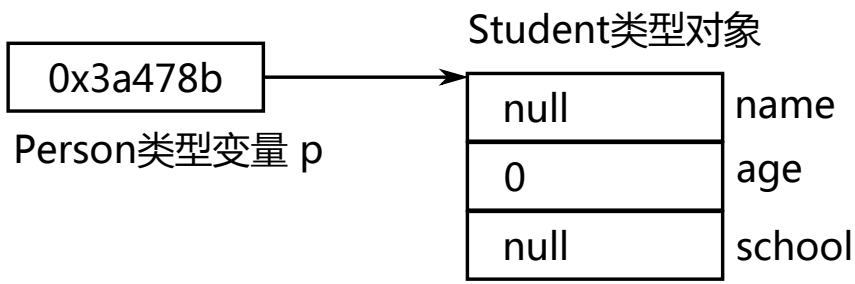
什么是多态？

在 Java 中，子类的对象可以替代父类的对象使用称为**多态**。

❖ 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
```



多态性

多态性同样适用与引用类型数组元素

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

☛ 父类引用对子类对象的能力屏蔽性

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，该变量则不能再访问子类中添加的属性和方法。

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

多态性

多态性同样适用与引用类型数组元素

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

☛ 父类引用对子类对象的能力屏蔽性

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，该变量则不能再访问子类中添加的属性和方法。

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

多态用法示例

CODE ◗ Person.java

```
1  public class Person {...}
```

CODE ◗ Student.java

```
1  public class Student extends Person {
2    private String school;
3
4    public void setSchool(String school) {
5      this.school = school;
6    }
7
8    public String getSchool() {
9      return school;
10   }
11
12   @Override
13   public String getInfo() {
14     return super.getInfo() + "\tSchool:␣" + school;
15   }
16 }
```

多态用法示例

CODE ▸ PolymorphismSample.java

```
2   public class PolymorphismSample {
3     public void show(Person p) {
4       System.out.println(p.getInfo());
5     }
7     public static void main(String[] args) {
8       PolymorphismSample ps = new PolymorphismSample();
9       Person p = new Person();
10      ps.show(p);
11      Student s = new Student();
12      ps.show(s);
13    }
14  }
```

☛ 多态提升方法通用性

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。

多态用法示例

CODE ▸ PolymorphismSample.java

```
2   public class PolymorphismSample {
3     public void show(Person p) {
4       System.out.println(p.getInfo());
5     }
7     public static void main(String[] args) {
8       PolymorphismSample ps = new PolymorphismSample();
9       Person p = new Person();
10      ps.show(p);
11      Student s = new Student();
12      ps.show(s);
13    }
14  }
```

☛ 多态提升方法通用性

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

☛ 思考

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

☛ 思考

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？

对象造型

引用类型数据值之间的强制类型转换称为**造型（Casting）**。

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实现。[1]

```
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
```

[1] 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。

对象造型

引用类型数据值之间的强制类型转换称为**造型（Casting）**。

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实现。[1]

```
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
```

[1] 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。

对象造型

引用类型数据值之间的强制类型转换称为**造型（Casting）**。

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实现。[1]

```
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
```

[1] 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。

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。

```
class Person { --- }
class Student extends Person { --- }

public class Tool {
  public void distribute(Person p) {
    if (p instanceof Student) {
      System.out.println("处理␣Student␣类型及其子类类型对象");
    } else {
      System.out.println("处理␣Person␣类型及其子类类型对象");
    }
  }
}
```

```
public class Test() {
  public static void main(String[] args) {
    Tool t = new Tool();
    Student s = new Student();
    t.distribute(t);
  }
}
```

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  }
```

虚方法调用和造型

课程配套代码 ◗ package sample.oop.poly

- VirtualMethodSample.java
- Person.java
- Student.java

❖ 虚方法调用和造型强化

- 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**。
- 造型是引用类型数据值之间的强制类型转换。
- instanceof 运算符判断的是当前所引用对象的真正类型是什么，而不是声明的引用类型。

虚方法调用和造型

课程配套代码 ◗ package sample.oop.poly

- VirtualMethodSample.java
- Person.java
- Student.java

❖ 虚方法调用和造型强化

- 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**。
- 造型是引用类型数据值之间的强制类型转换。
- instanceof 运算符判断的是当前所引用对象的真正类型是什么，而不是声明的引用类型。

虚方法调用和造型

课程配套代码 ◗ package sample.oop.poly

- VirtualMethodSample.java
- Person.java
- Student.java

❖ 虚方法调用和造型强化

- 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**。
- 造型是引用类型数据值之间的强制类型转换。
- instanceof 运算符判断的是当前所引用对象的真正类型是什么，而不是声明的引用类型。

接下来…

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。
- 重载既可以用于普通方法，也可以用于构造方法。

课程配套代码 ◗ sample.oop.MethodOverloadSample.java

如何调用重载的构造方法

❖ 使用 this 调用当前类中重载构造方法

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     ...
3     public Person(String name,int age) {
4       this.name = name;
5       this.age = age;
6     }
7     public Person(String name) {
8       this(name, 18);
9     }
10    ...
11  }
```

☛ 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。

如何调用重载的构造方法

❖ 使用 super 调用父类构造方法

CODE ◗ Person.java

```
1  public class Person {
2    ... (此处没有无参构造方法)
3    public Person(String name, int age) {
4      this.name = name;
5      this.age = age;
6    }
7    ...
8  }
```

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 school) { //编译出错
8      // 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
9      this.school = school;
10   }
11 }
```

上述代码为什么会编译出错

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。

课程配套代码 ➧ sample.oop.ConstructorOverloadSample.java

上述代码为什么会编译出错

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。

课程配套代码 ➧ sample.oop.ConstructorOverloadSample.java

上述代码为什么会编译出错

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。

课程配套代码 ➧ sample.oop.ConstructorOverloadSample.java

接下来…

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”和“静态方法”等。

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”和“静态方法”等。

static 属性和方法

❖ static 属性

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

❖ static 方法

要在 static 方法中调用其所在类的非 static 成员，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。

课程配套代码 ◗ sample.oop.StaticMemberAndMethodSample.java

初始化块

❖ 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。

❖ 非 static 初始化块

非 static 的初始化块在创建对象时被自动调用。

课程配套代码 ➧ sample.oop.StaticInitBlockSample.java

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[2]

❖ 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，极大的提高代码的重用性和可维护性。

[2]Singleton 设计模式也称“单子模式”或“单例模式”。

经典设计模式分类

创建型模式 涉及对象的实例化，特点是不让用户代码依赖于对象的创建或排列方式，避免用户直接使用 new 创建对象。

工厂方法模式、抽象工厂方法模式、生成器模式、原型模式和单例模式

行为型模式 涉及怎样合理的设计对象之间的交互通信，以及合理为对象分配职责，让设计富有弹性、易维护、易复用。

责任链模式、命令模式、解释器模式、迭代器模式、中介者模式、备忘录模式、观察者模式、状态模式、策略模式、模板方法模式和访问者模式

结构型模式 涉及如何组合类和对象以形成更大的结构，和类有关的结构型模式涉及如何合理使用继承机制，和对象有关的结构型模式涉及如何合理的使用对象组合机制。

适配器模式、组合模式、代理模式、享元模式、外观模式、桥接模式和装饰模式

经典设计模式分类

创建型模式 涉及对象的实例化，特点是不让用户代码依赖于对象的创建或排列方式，避免用户直接使用 new 创建对象。

工厂方法模式、抽象工厂方法模式、生成器模式、原型模式和单例模式

行为型模式 涉及怎样合理的设计对象之间的交互通信，以及合理为对象分配职责，让设计富有弹性、易维护、易复用。

责任链模式、命令模式、解释器模式、迭代器模式、中介者模式、备忘录模式、观察者模式、状态模式、策略模式、模板方法模式和访问者模式

结构型模式 涉及如何组合类和对象以形成更大的结构，和类有关的结构型模式涉及如何合理使用继承机制，和对象有关的结构型模式涉及如何合理的使用对象组合机制。

适配器模式、组合模式、代理模式、享元模式、外观模式、桥接模式和装饰模式

经典设计模式分类

创建型模式 涉及对象的实例化，特点是不让用户代码依赖于对象的创建或排列方式，避免用户直接使用 new 创建对象。

工厂方法模式、抽象工厂方法模式、生成器模式、原型模式和单例模式

行为型模式 涉及怎样合理的设计对象之间的交互通信，以及合理为对象分配职责，让设计富有弹性、易维护、易复用。

责任链模式、命令模式、解释器模式、迭代器模式、中介者模式、备忘录模式、观察者模式、状态模式、策略模式、模板方法模式和访问者模式

结构型模式 涉及如何组合类和对象以形成更大的结构，和类有关的结构型模式涉及如何合理使用继承机制，和对象有关的结构型模式涉及如何合理的使用对象组合机制。

适配器模式、组合模式、代理模式、享元模式、外观模式、桥接模式和装饰模式

Singleton 设计模式

☛ 采用调试方式讲解示例代码

课程配套代码 ⏵ package sample.oop.singleton

Singleton 设计模式

❖ 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。

❖ Singleton 的使用场景

在任何使用到 Single 类的 Java 程序中（这里指的是一次运行中），需要确保只有一个 Single 类的实例存在（如 Web 应用 ServletContext 全局上下文对象），则使用该模式。

Singleton 设计模式

❖ 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。

❖ Singleton 的使用场景

在任何使用到 Single 类的 Java 程序中（这里指的是一次运行中），需要确保只有一个 Single 类的实例存在（如 Web 应用 ServletContext 全局上下文对象），则使用该模式。

Singleton 设计模式

❖ 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。

❖ Singleton 的使用场景

在任何使用到 Single 类的 Java 程序中（这里指的是一次运行中），需要确保只有一个 Single 类的实例存在（如 Web 应用 ServletContext 全局上下文对象），则使用该模式。

Singleton 设计模式

❖ 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。

❖ Singleton 的使用场景

在任何使用到 Single 类的 Java 程序中（这里指的是一次运行中），需要确保只有一个 Single 类的实例存在（如 Web 应用 ServletContext 全局上下文对象），则使用该模式。

Singleton 设计模式

❖ 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。

❖ Singleton 的使用场景

在任何使用到 Single 类的 Java 程序中（这里指的是一次运行中），需要确保只有一个 Single 类的实例存在（如 Web 应用 ServletContext 全局上下文对象），则使用该模式。

接下来…

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 }
```

本节习题

❖ 简答题

1. 为什么建议 Java 类需要编写无参构造方法，哪怕该方法什么都没做？
2. 关键字 static 都可以用来修饰 Java 类的那些成员？

❖ 小编程

1. 练习本节中所有示例代码，理解并掌握其用法。
2. 自行调试单例设计模式程序，学习 Eclipse 的 Debug 方法，练习断点、单步执行、跟进方法等操作，查看内存变化。

THE END

wangxiaodong@ouc.edu.cn