

JAVA 应用与开发

高级类特性

让我们愉快的 *Coding* 起来吧...

王晓东

中国海洋大学信息学院计算机系

SEPTEMBER 30, 2018



学习目标

1. 抽象类
2. 接口
3. 内部类
4. 枚举类型

1 抽象类

2 接口

3 嵌套类

4 枚举类型

抽象类

什么是抽象类

抽象类

在面向对象的概念中，所有的对象都是通过类来描绘的，但是反过来，并不是所有的类都是用来描绘对象的。如果一个类中没有包含足够的信息来描绘一个具体的对象，这样的类就是抽象类。

什么是抽象类

抽象类

在面向对象的概念中，所有的对象都是通过类来描绘的，但是反过来，并不是所有的类都是用来描绘对象的。如果一个类中没有包含足够的信息来描绘一个具体的对象，这样的类就是抽象类。

抽象类往往用来表征对问题领域进行分析、设计中得出的抽象概念，是对一系列看上去不同但是本质上相同的具体概念的抽象。



图：我很抽象

定义抽象类

- 在定义 Java 方法时可以只给出方法头，而不必给出方法的实现细节，这样的方法被称为**抽象方法**。
- 抽象方法必须用关键字**abstract**修饰。
- 包含抽象方法的类必须声明为抽象类，用关键字**abstract**修饰。

CODE ▶ 抽象类示例

```
1 public abstract class Animal { //定义为抽象类
2     private int age;
3
4     public void setAge(int age) {
5         this.age = age;
6     }
7
8     public int getAge(){
9         return age;
10    }
11
12    public abstract void eat(); //抽象方法
13 }
```

定义抽象类

CODE ▶ 抽象类继承

```
1 public class Person extends Animal {
2     private String name;
3     public void setName(String name) {
4         this.name = name;
5     }
6     public String getName() {
7         return name;
8     }
9     public void eat() { //重写方法
10        System.out.println("洗手→烹饪→摆餐具→吃喝→收摊儿");
11    }
12 }
```

```
1 public class Bird extends Animal {
2     public void fly(){
3         System.out.println("我心飞翔!");
4     }
5     public void eat(){ //重写方法
6         System.out.println("直接吞食!");
7     }
8 }
```


抽象类的特性与作用

❖ 抽象类的特性

- 子类必须实现其父类中的所有抽象方法，否则该子类也只能声明为抽象类。

抽象类的特性与作用

❖ 抽象类的特性

- 子类必须实现其父类中的所有抽象方法，否则该子类也只能声明为抽象类。
- 抽象类不能被实例化。

抽象类的特性与作用

❖ 抽象类的特性

- 子类必须实现其父类中的所有抽象方法，否则该子类也只能声明为抽象类。
- 抽象类不能被实例化。

抽象类的特性与作用

❖ 抽象类的特性

- 子类必须实现其父类中的所有抽象方法，否则该子类也只能声明为抽象类。
- 抽象类不能被实例化。

❖ 抽象类的作用

抽象类主要是通过继承由其子类发挥作用，包括两方面：

代码重用 子类可以重用抽象类中的属性和非抽象方法。

规划 子类中通过抽象方法的重写来实现父类规划的功能。

抽象类的其他特性

- 抽象类中可以不包含抽象方法。主要用于当一个类已经定义了多个更适用的子类时，为避免误用功能相对较弱的父类对象，干脆限制其实例化。

抽象类的其他特性

- 抽象类中可以不包含抽象方法。主要用于当一个类已经定义了多个更适用的子类时，为避免误用功能相对较弱的父类对象，干脆限制其实例化。
- 子类中可以不全部实现抽象父类中的抽象方法，但此时子类也只能声明为抽象类。

抽象类的其他特性

- 抽象类中可以不包含抽象方法。主要用于当一个类已经定义了多个更适用的子类时，为避免误用功能相对较弱的父类对象，干脆限制其实例化。
- 子类中可以不全部实现抽象父类中的抽象方法，但此时子类也只能声明为抽象类。
- 父类不是抽象类，但在子类中可以添加抽象方法，此情况下子类必须声明为抽象类。

抽象类的其他特性

- 抽象类中可以不包含抽象方法。主要用于当一个类已经定义了多个更适用的子类时，为避免误用功能相对较弱的父类对象，干脆限制其实例化。
- 子类中可以不全部实现抽象父类中的抽象方法，但此时子类也只能声明为抽象类。
- 父类不是抽象类，但在子类中可以添加抽象方法，此情况下子类必须声明为抽象类。
- 多态性对于抽象类仍然适用，可以将引用类型变量（或方法的形参）声明为抽象类的类型。

抽象类的其他特性

- 抽象类中可以不包含抽象方法。主要用于当一个类已经定义了多个更适用的子类时，为避免误用功能相对较弱的父类对象，干脆限制其实例化。
- 子类中可以不全部实现抽象父类中的抽象方法，但此时子类也只能声明为抽象类。
- 父类不是抽象类，但在子类中可以添加抽象方法，此情况下子类必须声明为抽象类。
- 多态性对于抽象类仍然适用，可以将引用类型变量（或方法的形参）声明为抽象类的类型。
- 抽象类中可以声明 **static** 属性和方法，只要访问控制权限允许，这些属性和方法可以通过 **< 类名 >.< 类成员 >** 的方法进行访问。

接口

接口（INTERFACE）

接口

在科技辞典中，“接口”被解释为“两个不同系统（或子程序）交接并通过它彼此作用的部分。”

接口（INTERFACE）

接口

在科技辞典中，“接口”被解释为“两个不同系统（或子程序）交接并通过它彼此作用的部分。”

在 Java 语言中，通过接口可以了解对象的交互界面，即明确对象提供的功能及其调用格式，而不需要了解其实现细节。

接口（INTERFACE）

接口

在科技辞典中，“接口”被解释为“两个不同系统（或子程序）交接并通过它彼此作用的部分。”

在 Java 语言中，通过接口可以了解对象的交互界面，即明确对象提供的功能及其调用格式，而不需要了解其实现细节。

接口是抽象方法和常量值的定义的集合。从本质上讲，接口是一种特殊的**抽象类**，这种抽象类中**只包含常量定义和方法声明，而没有变量和方法的实现。**

定义接口

接口中定义的属性必须是 `public static final` 的，而接口中定义的方法则必须是 `public abstract` 的，因此这些关键字可以部分或全部省略。

CODE ▶ 接口示例（未简化）

```
1 public interface Runner {  
2     public static final int id = 1;  
3     public abstract void start();  
4     public abstract void run();  
5     public abstract void stop();  
6 }
```

CODE ▶ 与上述代码等价的标准定义

```
1 public interface Runner {  
2     int id = 1;  
3     void start();  
4     void run();  
5     void stop();  
6 }
```

接口的实现

和继承关系类似，类可以**实现**接口，且接口和实现类之间也存在多态性。

❖ 类继承和接口实现的语法格式

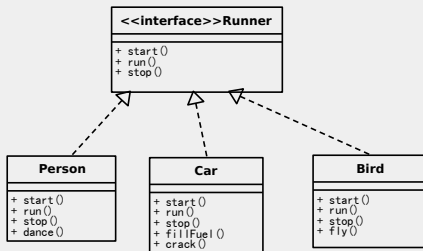
```
1  [ <modifier> ] class <name> [ extends <superclass> ] [ implements <interface> [, <interface>]* ] {
2      <declarations>*
3  }
```

CODE ▶ 接口实现示例

```
1 public class Person implements Runner {
2     public void start() {
3         System.out.println("弯腰、蹬腿、咬牙、瞪眼、开跑");
4     }
5     public void run(){
6         System.out.println("摆动手臂、维持直线方向");
7     }
8     public void stop(){
9         System.out.println("减速直至停止、喝水");
10    }
11 }
```


接口的实现

通过接口可以指明多个类需要实现的方法，而这些类还可以根据需
要继承各自的父类。或者说，通过接口可以实现不相关类的相同行
为，而不需要考虑这些类之间的层次关系。



👉 类允许实现多重接口

课程配套代码 📌 `package sample.advance.interfacesample`

接口间的继承

与接口的多重实现情况类似，由于不担心方法追溯调用上的不确定性，接口之间的继承允许“多重继承”的情况。

```
1  interface A {  
2      public void ma();  
3  }  
4  interface B {  
5      public int mb(int i);  
6  }  
7  interface C extends A,B { //接口的多重继承  
8      public String mc();  
9  }  
10 class D implements C {  
11     public void ma() {  
12         System.out.println("Implements_method_ma()!");  
13     }  
14     public int mb(int i) {  
15         return 2000 + i;  
16     }  
17     public String mc() {  
18         return "Hello!";  
19     }  
20 }
```

上述代码中的 D 类缺省继承了 Object 类，直接实现了接口 C，间接实现了接口 A 和 B，由于多态性的机制，将来 D 类的对象可以当作 Object、C、A 或 B 等类型使用。

接口特性总结

- 通过接口可以实现不相关类的相同行为，而不需要考虑这些类之间的层次关系。

接口特性总结

- 通过接口可以实现不相关类的相同行为，而不需要考虑这些类之间的层次关系。
- 接口可以被多重实现。

接口特性总结

- 通过接口可以实现不相关类的相同行为，而不需要考虑这些类之间的层次关系。
- 接口可以被多重实现。
- 接口可以继承其它的接口，并添加新的属性和抽象方法，接口间支持多重继承。

嵌套类

什么是嵌套类

Java 语言支持类的嵌套定义，即允许将一个类定义在其他类的内部，其中内层的类被称为嵌套类。

❖ 嵌套类的分类

静态嵌套类 (Static Nested Class) 使用 `static` 修饰的嵌套类

内部类 (Inner Class) 非 `static` 的嵌套类

普通内部类 在类中的方法或语句块外部定义的非 `static` 类。

局部内部类 定义在方法或语句块中的类，也称局部类。

匿名内部类 定义在方法或语句块中，该类没有名字、只能在其所在之处使用一次。

静态嵌套类

- 静态嵌套类不再依赖/引用外层类的特定对象，只是隐藏在另一个类中而已。
- 由于静态嵌套类的对象不依赖外层类的对象而独立存在，因而可以直接创建，进而也就无法在静态嵌套类中直接使用其外层类的非 static 成员。

课程配套代码 ▶ `sample.advance.nestedclass.StaticNestedClassSample.java`

匿名内部类可以被认为是局部类的一种简化。

当我们只在一处使用到某个类型时，可以将之定义为局部类，进而如果我们只是创建并使用该类的一个实例的话，那么连类的名字都可以省略。

使用匿名内部类①

CODE ◆ JavaSE_03/TestAnonymousInnerClass.java

CODE ◆ Person.java

```
1 public abstract class Person {  
2     private String name;  
3     private int age;  
4     public Person() {}  
5     public Person(String name, int age) {  
6         this.name = name;  
7         this.age = age;  
8     }  
9     public String getInfo() {  
10        return "Name:␣" + name + "\t␣Age:␣" + age;  
11    }  
12    public abstract void work();  
13 }
```

使用匿名内部类①

CODE ♦ TestAnonymous.java

```
1 public class TestAnonymous {
2     public static void main(String[] args) {
3         Person sp = new Person() { \\ 匿名内部类
4             public void work() {
5                 System.out.println("个人信息: " + this.getInfo());
6                 System.out.println("I am sailing.");
7             }
8         };
9         sp.work();
10    }
11 }
```

上述代码的解释

定义一个新的 Java 内部类，该类本身没有名字，但继承了指定的父类 Person，并在此匿名子类中重写了父类的 work() 方法，然后立即创建了一个该匿名子类的对象，再将其句柄保存到引用变量 sp 中待用。

使用匿名内部类①

由于匿名类没有类名，而构造方法必须与类同名，所以匿名类不能显式的定义构造方法，但系统允许在创建匿名类对象时将参数传给父类构造方法（使用父类的构造方法）。

```
1 Person sp = new Person("Kevin", 30) {  
2     public void work() {  
3         System.out.println("个人信息: " + this.getInfo());  
4         System.out.println("I am sailing.");  
5     }  
6 };
```

使用匿名内部类②

匿名类除了可以继承现有父类之外，还可以实现接口，但不允许实现多个接口，且实现接口时就不能再继承父类了，反之亦然。

CODE 🌸 JavaSE_03/TestAnonymousInnerClass02.java

CODE 🌸 Swimmer.java

```
1 public interface Swimmer {  
2     public abstract void swim();  
3 }
```

CODE 🌸 TestAnonymous2.java

```
1 public class TestAnonymous2 {  
2     public static void main(String[] args) {  
3         TestAnonymous2 ta = new TestAnonymous2();  
4         ta.test(new Swimmer() { // 匿名类实现接口  
5             public void swim() {  
6                 System.out.println("I am swimming.");  
7             }  
8         });  
9  
10        public void test(Swimmer swimmer) {  
11            swimmer.swim();  
12        }  
13    }
```

使用匿名内部类②

上述程序 `main()` 方法中的代码相当于：

```
1 public static void main(String[] args) {  
2     TestAnonymous2 ta = new TestAnonymous2();  
3     class Person implements Swimmer {  
4         public void swim() {  
5             System.out.println("I am swimming.")  
6         }  
7     }  
8     ta.test(new Person());  
9 }
```

枚举类型

枚举类型

Java SE 5.0 开始，引入了一种新的引用数据结构——枚举 (Enum)。

Java 语言中枚举类型均自动继承了 `java.lang.Enum` 类（该类继承自 `Object` 类）。枚举类型使用一组常量值来表示特定的数据集合，该集合中数据的数目确定（通常较少），且这些数据只能取预先定义的值。

```
1 public enum Week {  
2     MON, TUE, WED, THU, FRI, SAT, SUN  
3 }
```

❖ 之前我们如何解决上述需求？

之前实现枚举类的功能，Java 开发者一般采用声明多个整型常量的做法。

```
1 public class Week {  
2     public static final int MON = 1;  
3     public static final int TUE = 2;  
4     ...  
5 }
```


使用枚举类型

```
1 public enum Week {  
2     MON, TUE, WED, THU, FRI, SAT, SUN  
3 }
```

```
1 public class TestEnum {  
2     public static void main(String[] args) {  
3         TestEnum te = new TestEnum();  
4         te.work(Week.SUN);  
5     }  
6     public void work(Week day) {  
7         if (day.equals(Week.SAT)) {  
8             System.out.println("购物!");  
9         } else if (day.equals(Week.SUN)) {  
10            System.out.println("祈祷!");  
11        } else {  
12            System.out.println("工作!");  
13        }  
14    }  
15 }
```

遍历枚举类型常量值

可以使用静态方法 `values()` 遍历枚举类型常量值。

CODE 🍷 ListEnum.java

```
1 public class ListEnum {  
2     public static void main(String[] args) {  
3         Week[] days = Week.values();  
4         for(Week d: days) {  
5             System.out.println(d);  
6         }  
7     }  
8 }
```

组合使用枚举类型与 SWITCH

使用枚举类型通常是为了实现多路分支性结构。

```
1 public class TestEnumInSwitch {  
2     public static void main(String[] args) {  
3         TestEnumInSwitch teis = new TestEnumInSwitch();  
4         teis.work(Week.FRI);  
5     }  
6     public void work(Week day) {  
7         switch(day) {  
8             case MON:  
9             case TUE:  
10            case WED:  
11            case THU:  
12            case FRI:  
13                System.out.println("工作日, 去上班! ");  
14                break;  
15            case SAT:  
16                System.out.println("星期六, 去购物! ");  
17                break;  
18            case SUN:  
19                System.out.println("礼拜天, 去教堂! ");  
20                break;  
21            default:  
22                System.out.println("你有没有搞错! ");  
23        }  
24    }  
25 }
```

❖ 注意

1. case 字句必须省略其枚举类型的前缀，即只需要写成 case SUN:，而不允许写成 case Week.SUN:，否则编译出错。
2. 不必担心系统无法搞清这些常量名称的出处，因为 switch 后的小括号中的表达式已经指明本次要区分处理的是 Week 类型常量。

THE END

wxd2870@163.com