

✧ 1 ✧ 高级类特性

基本信息

课程名称： Java 应用与开发

授课教师： 王晓东

授课时间： 第四周

参考教材： 本课程参考教材及资料如下：

- 陈国君主编，Java 程序设计基础（第 5 版），清华大学出版社，2015.5
- Bruce Eckel, Thinking in Java (3rd)

教学目标

1. 掌握抽象类和接口的概念、特性及定义方法
2. 理解抽象类和接口的异同和作用
3. 了解嵌套类的分类，掌握嵌套类中静态嵌套类和匿名嵌套类的概念
4. 掌握匿名内部类的特征、继承和接口实现的用法
5. 掌握枚举类型的使用方法

授课方式

理论课： 多媒体教学、程序演示

实验课： 上机编程

教学内容

1.1 抽象类

1.1.1 抽象类的概念

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。

1.1.2 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 }
```

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 }

```

1.1.3 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。**问题** 抽象类能否有构造方法？

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。

- 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。

1.2 接口

1.2.1 接口（interface）的概念

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。

1.2.2 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 }

```

1.2.3 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如下：

```

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 }

```

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求继承各自的父类。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

1.2.4 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。

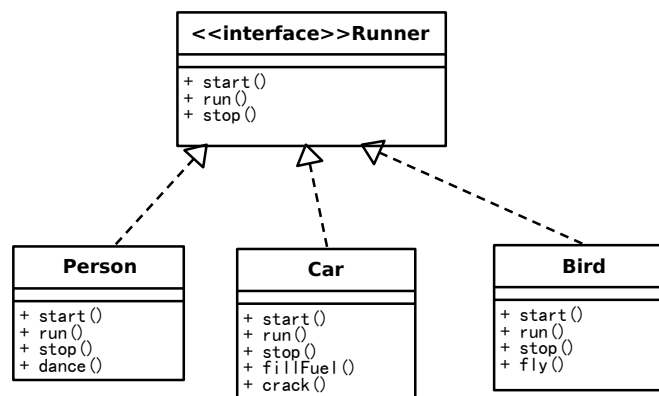


图 1.1 接口实现示例

```

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。

1.2.5 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。

1.3 抽象类和接口剖析

1.3.1 语法层面的区别

概念差异——语法差异——用法差异——设计哲学

- 抽象类可以提供成员方法的实现细节，而接口中只能存在 `public abstract` 方法
- 抽象类中的成员变量可以为各种类型，而接口中的成员变量只能是 `public static final` 类型
- 抽象类可以有静态代码块和静态方法，接口中不能含有静态代码块以及静态方法
- 一个类只能继承一个抽象类，而一个类却可以实现多个接口

1.3.2 设计层面的区别

- 抽象类是对类的抽象（可以抽象但不宜实例化），而接口是对行为的抽象。
- 抽象类是对类整体进行抽象，包括属性、行为，但是接口却是对类局部（行为）进行抽象。
- 抽象类作为很多子类的父类，它是一种模板式设计。**模板式设计：模板代表公共部分，公共部分需要改的则改动模板即可。**
- 而接口是一种行为规范，它是一种辐射式设计。**辐射式设计：接口进行了变更，则所有实现类都必须进行相应的改动。**

1.3.3 怎样才是合理的设计？（门和警报的示例）

以门和警报设计作为示例，一般来说，门都有 `open()` 和 `close()` 这两个动作。通过抽象类和接口来定义这个抽象概念：

```
1 abstract class Door {  
2     public abstract void open();  
3     public abstract void close();  
4 }
```

```
1 interface Door {  
2     public abstract void open();  
3     public abstract void close();  
4 }
```

问题

如果现在我们需要门具有报警 `alarm()` 的功能该如何实现？

思路一

将这三个功能都放在抽象类里面，这样一来所有继承这个抽象类的子类都具备了报警功能，但是有的门并不一定需要具备报警功能。**不合理抽象**

思路二

将这三个功能都放在接口里面，但需要用到报警功能的类就需要实现这个接口中的 `open()` 和 `close()`，也许这个类根本就不具备 `open()` 和 `close()` 这两个功能，比如火灾报警器。**不合理规划**

`Door` 的 `open()`、`close()` 和 `alarm()` 根本就属于两个不同范畴内的行为：

- `open()` 和 `close()` 属于门本身固有的行为特性。
- `alarm()` 属于延伸的附加行为。

更为合理的思路

❶ 单独将报警设计为一个接口，包含 `alarm()` 行为；❷ `Door` 设计为单独的抽象类，包含 `open()` 和 `close()` 两种行为；❸ 设计一个报警门继承 `Door` 类和实现 `Alarm` 接口。

课程配套代码 ▶ `package sample.advance.door`

1.4 嵌套类

1.4.1 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。

1.4.2 静态嵌套类

静态嵌套类的特征

- 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

1.4.3 匿名内部类

匿名内部类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。

1.4.4 使用匿名内部类

Code: [Person.java](#)

```

1  public abstract class Person {
2      private String name;
3      private int age;

5      public Person() {}

7      public Person(String name, int age) {
8          this.name = name;
9          this.age = age;
10     }

12     public String getInfo() {
13         return "Name: " + name + "\t Age: " + age;
14     }

16     public abstract void work();
17 }

```

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          };

10         sp.work();
11     }
12 }

```

对上述代码的解释如下：

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，再将其地址保存到引用变量 sp 中待用。

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 };

```

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: [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    }
14 }

```

1.5 枚举类型

1.5.1 枚举类型的概念

Java SE 5.0 开始，引入了一种新的引用数据结构**枚举类型**。枚举类型均自动继承 `java.lang.Enum` 类，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 }

```

无枚举类型前如何解决上述需求？

一般采用声明多个整型常量的做法实现枚举类的功能。

```

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

```

1.5.2 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 }

```

1.5.3 组合使用枚举类型与 switch

课程配套代码  package sample.advance.enumclass

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。

实验设计

❖ 2 ❖ 泛型

基本信息

课程名称： Java 应用与开发

授课教师： 王晓东

授课时间： 第四周

参考教材： 本课程参考教材及资料如下：

- 陈国君主编，Java 程序设计基础（第 5 版），清华大学出版社，2015.5
- Bruce Eckel, Thinking in Java (3rd)

教学目标

1. 理解泛型的概念，掌握其基本应用
 - 集合框架中的泛型
 - 泛型的向后兼容性
2. 掌握自定义泛型类和泛型方法
 - 理解类型参数
 - 理解差异性并能够定义自己的泛型类和泛型方法
 - 受限制的类型参数
3. 学会处理泛型类型，包括使用通配符实现泛型容器遍历和操作

授课方式

理论课： 多媒体教学、程序演示

实验课： 上机编程

教学内容

2.1 泛型概念

2.1.1 泛型的概念

泛型机制自 JDK 5.0 开始引入，其实质是**将原本确定不变的数据类型参数化**。作为对原有 Java 类型体系的扩充，使用泛型可以提高 Java 应用程序的类型安全、可维护性和可靠性。

传统的集合容器为了提供广泛的适用性，会将所有加入其中的元素当作 Object 类型来处理。基于此原因，在实际使用时，我们必须将从集合中取出的元素值再强制转换（造型）为所期望的类型。

无泛型机制的集合容器

```
1 Vector v = new Vector();
2 v.addElement(new Person("Tom", 18));
3 Person p = (Person) v.elementAt(0);
4 p.showInfo();
```

2.1.2 集合框架中的泛型

- 泛型允许编译器实施由开发者设定的附加类型约束，将类型检查从**运行时挪到编译时**进行，这样类型错误就可以在编译时暴露出来，而不是在运行时才发作（抛出 ClassCastException 运行异常）。
- 创建集合容器时规定其允许保存的元素类型，然后由编译器负责添加元素的类型合法性检查，在取用集合元素时则不必再进行造型处理。

在 Vector 中使用泛型

课程配套代码 ♦ sample.generics.VectorGenericsSample.java

在 Hashtable 中使用泛型

课程配套代码 ♦ sample.generics.HashtableGenericsSample.java

泛型的向后兼容性

- Java 语言中的泛型是维护向后兼容的，完全可以不采用泛型、而继续沿用过去的做法。
- 这些未加改造的旧式代码将无法使用泛型带来的便利和安全性。

未启用泛型机制的代码在高版本编译器中会输出如下形式的编译提示信息：

output

注: VectorGenericsSample.java 使用了未经检查或不安全的操作。

注: 有关详细信息, 请使用 `-Xlint:unchecked` 重新编译。

可以使用 `SuppressWarnings` 注解关闭编译提示或警告信息：

```
1  @SuppressWarnings({"unchecked"})
2  public class VectorGenericsSample {
3      ...
4  }
```

2.2 泛型类与泛型方法

类型参数

形如 `Vector<String>`，其中，尖括号括起来的部分被称为**类型参数**，而这种由类型参数修饰的类型则被称为**泛型类**。

注意

应在声明泛型类变量和创建对象时均给出类型参数，且两者应保持一致。

使用类型参数 *E* 进行泛型化处理的 `java.util.Vector` 类的定义代码摘要如下：

```
1  public class Vector<E> --- {
2      public void addElement(E obj) { --- }
3      public E elementAt(int index) { --- }
4  }
```

这里的 *E* 也称为“形式类型”参数。在实际使用该泛型类时，我们需要指定相应的具体类型，即实际类型参数。

```
1  Vector<String> v = new Vector<String>();
```

编译器遇到 `Vector<String>` 类型变量时，即知道此 `Vector` 变量/对象的类型参数 *E* 已经被绑定为 `String` 类型，进而也就确定其 `addElement()` 方法的参数和 `elementAt()` 方法的返回值均为 `String` 类型。

符号	意义
K	键，比如映射的键
V	值，比如 List 和 Set 的内容，或者 Map 中的值
E	元素，比如 Vector<E>
T	泛型

2.2.1 定义泛型类

课程配套代码 `package sample.generics.userdefined`

代码示例中的泛型类 `PersonG` 可以在使用时通过类型参数 `T` 指定其属性 `secrecy` 的具体类型（以及该属性相应存/取方法的参数和返回值类型），进而提供了通用的信息存储能力。

形式类型参数的编程惯例

使用受限制的类型参数

泛型机制允许开发者对类型参数进行附加约束。

```

1  import java. utils .Number;

3  public class Point<T extends Number> {
4      private T x;
5      private T y;
6      public Point() {}
7      public Point(T x, T y) {
8          this.x = x;
9          this.y = y;
10     }
11     ...
12 }
```

类型参数不能为基本数据类型，而 `java.lang.Number` 是所有数值型封装类（如 `Integer`、`Float`、`Double` 等）的父类型，于是限制泛型类 `Point` 的类型参数必须为 `Number` 或其子类类型，并使用 `extends` 关键字来标明这种继承层次上限制。

2.2.2 定义泛型方法

与泛型类的情况类似，**方法也可以被泛型化，且无论其所属的类是否为泛型类。**

Code: 泛型方法示例

```

1 public class Tool {
2     public <T>T evaluate(T a, T b) {
3         if (a.equals(b))
4             return a;
5         else
6             return null;
7     }
8 }

```

- 上述代码中方法 `evaluate()` 声明中的“<T>”用于标明这是一个泛型方法。
- 类型 `T` 是可变的，不必显示的告知编译器 `T` 具体取何值，但出现多处（两个形参、一个返回值类型）的这些值必须都相同。

使用泛型方法，而不是定义泛型类的原则

1. 不涉及到类中的其他方法时，则应将之定义为泛型方法，因为泛型方法的类型参数是局部性的，这样可以简化其所在类型的声明和处理开销。
2. 要施加类型约束的方法为静态方法时，只能将之定义为泛型方法，因为静态方法不能使用其所在类的类型参数。

对泛型的理解

- 泛型类可以理解为具有广泛适用性、尚未最终定型的类型。
- `Person<String>` 和 `Person<Double>` 属于同一个类，但确是不同的类型。
- 同一个泛型类与不同的类型参数复合而成的类型间并不存在继承关系，即使是类型参数间存在继承关系时也是如此。
如：`Vector<String>` 不是 `Vector<Object>` 的子类。

2.3 处理泛型类型

2.3.1 遍历泛型 Vector 集合

遍历 `Vector<String>` 类型集合

```

1 public void overview(Vector<String> v) {
2     for (String o: v) {
3         String.out.println(o);

```

```
4     }  
5 }
```

遍历其他类型参数的 Vector 集合元素该怎么办？

难道要定义 `overview(Vector<Person> v)`、`overview(Vector<Integer> v)` ... 显然过于繁琐，引用泛型机制后代码的通用性似乎不如从前？

2.3.2 泛型类型的处理方法

可能的处理方法（不要使用）

将遍历方法的形参定义为不带任何类型参数的原型类型 `Vector`，但这样会破坏已有的类型安全性。

```
1 public void overview(Vector v) {  
2     for(Object o: v) {  
3         String.out.println(o);  
4     }  
5 }
```

使用通配符

为了解决类似泛型遍历的问题，Java 泛型机制中引入了通配符 “?”。

```
1 public void overview(Vector<?> v) {  
2     for(Object o: v) {  
3         System.out.println(o);  
4     }  
5 }
```

类型通配符

使用类型通配符的好处包括：

1. `Vector<?>` 是任何泛型 `Vector` 的父类型，因此可以将 `Vector<String>`、`Vector<Integer>`、`Vector<Object>` 等作为实参传给 `overview(Vector<?> v)` 方法处理；
2. `Vector<?>` 类型的变量在调用方法时是受到限制的——凡是必须知道具体类型参数才能进行的操作均被禁止。

```
1 Vector<String> vs = new Vector<String>();
2 vs.add("Tom");
3 Vector<?> v = vs;
4 v.add("Billy"); // 非法, 编译器不知道具体类型参数
5 Object e = v.elementAt(0); // 合法, 允许检索元素, 此时读取的元素均当作 Object
   类型处理
6 System.out.println(e);
```

上述限制不等同于将 `Vector<?>` 变为“只读”，在不需要编译器确定类型参数的情况下也是可以修改集合内容的，例如：

```
1 Vector<String> vs = new Vector<String>();
2 vs.add("Tom");
3 vs.add("Billy");
4 Vector<?> v = vs;
5 v.remove(new Integer(200)); // 形参为 Object 类型, 运行不受影响
6 v.clear(); // 不需要参数, 运行不受影响
```

实验设计
