



DeepL

订阅DeepL Pro以翻译大型文件。

欲了解更多信息，请访问www.DeepL.com/pro。

JC2002 Java 程序设计

第 4 天：抽象类和接口（人工智能、计算机科学与技术）

JC2002 Java 程序设计

第 4 天，第 1 课时：抽象课堂

参考文献和学习目标

- 今天的会议主要基于
 - Evans, B. and Flanagan, D., 2018.**Java in a Nutshell: 桌面快速参考**》，7th 版。O'Reilly Media.
 - Deitel, H., 2018.**Java 如何编程, 早期对象, 全球版**, 第 11 版。Pearson.
- 今天的课程结束后, 您应该能够
 - 在 Java 程序中使用抽象类和接口
 - 设计适当的类层次结构, 包括抽象类和

界面

抽象类

- 抽象类是不能实例化为对象的类
 - 只在继承层次中作为超类使用，因此它们有时被称为 *抽象超类*
 - 不能用于实例化对象--抽象类 *不完整*
 - 子类必须声明 "缺失的部分"，才能成为 "具体" 类，您才能从中实例化对象；否则，这些子类也将是抽象的
- 抽象类提供了一个超类，其他类可以从其中继承，因此具有共同的设计

抽象类与具体类

- 可用于实例化对象的类是*具体类*
 - 这些类为其声明的每个方法提供实现
(部分实现可以继承)
- 抽象超类过于笼统，无法创建真正的对象--它们只规定了子类之间的共同点
- 具体类提供了使实例化对象变得合理的具体内容
- 并非所有层次结构都包含抽象类

声明抽象类

- 使用关键字 `abstract` 来声明一个**抽象类**
- 抽象类通常包含一个或多个**抽象方法**
 - 抽象方法是用关键字 `abstract` 定义的，例如

```
public abstract void draw(); // 抽象方法
```
 - 抽象方法不提供实现
- 包含抽象方法的类必须是抽象类，即使该类包含一些具体（非抽象）方法

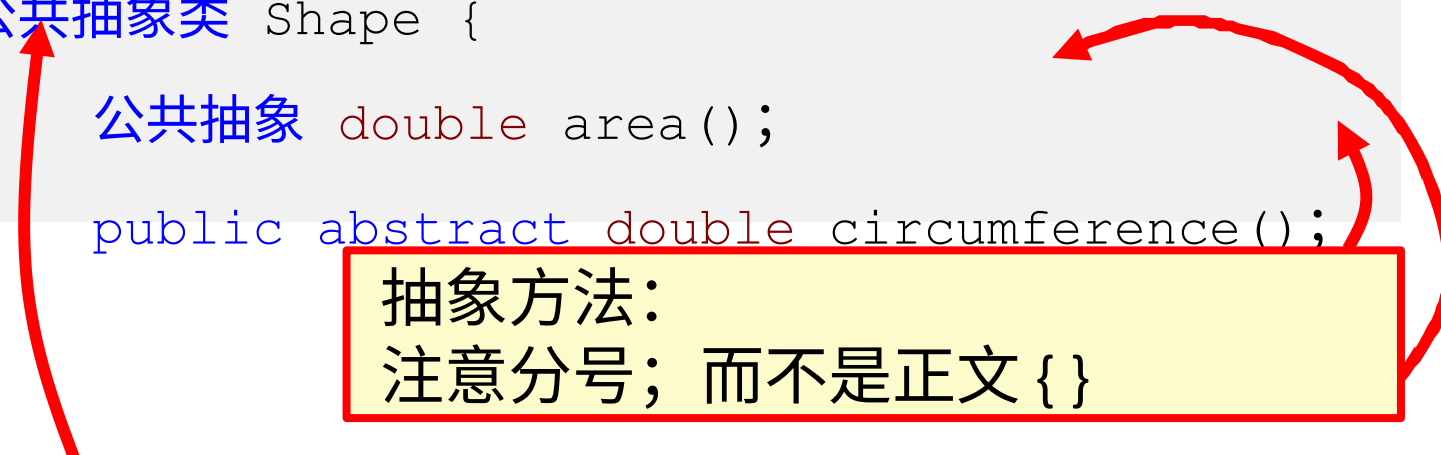
- 抽象超类的每个具体子类也必须提供超类抽象方法的具体实现

定义抽象类示例

```
1 公共抽象类 Shape {  
2  
3      公共抽象 double area();  
4  
    public abstract double circumference();  
}
```

定义抽象类示例

```
1 公共抽象类 Shape {  
2  
3      公共抽象 double area();  
4  
    public abstract double circumference();  
}
```



抽象方法：
注意分号；而不是正文 {}

请注意，公有类必须放在自己的 java 文件中！

扩展抽象类示例 (1)

```
1  class Circle extends Shape { public
2      static final double PI =          3.14159265358979323846;
3
4      protected double r;
5      public Circle(double r) { this.r = r; }
6      public double getRadius() { return r; }
7      public double area() { public return PI*r*r; }
8      double circumference() { return 2*PI*r; }
   }
```

扩展抽象类示例 (2)

```
1  类 Rectangle 扩展 Shape {  
2      受保护的 double w、h;  
3      public Rectangle(double w, double h) {  
4          this.w = w;  
5          this.h = h;  
6      }  
7      public double getWidth()      { return w;      }  
8      public double getHeight()     { return h;      }  
9      public double area()          { return w*h;      }  
10     public double circumference() { return 2*(w+h); }  
11 }
```

测试继承类示例

```
1  类 TestShape {  
2      public static void main(String[] args) {  
3          形状  
4          shape = new Circle(5);  
5          System.out.println("Area: " + shape.area());  
6          shape = 新矩形 (5,10) ;  
7          System.out.println("Area: " + shape.area());  
8      }
```


覆盖抽象方法

- 子类可以覆盖父类的公共非静态方法
 - 如果超类包含抽象方法，具体子类必须覆盖它们!
- **@Override** 注解的使用是可选的
 - 但是，如果不使用 **@Override** 注解，编译器就不会检查你是否真的在覆盖一个现有方法

方法覆盖示例 (1)

```
1 抽象类 动物 { 公共抽象 void  
2      makeSound();  
3  }  
4
```

```
5  
6 类 Cat extends Animal {  
7      public void makeSound() {  
8          System.out.println("Meow!");  
9      };  
10 }  
11
```

```
12 类 Dog extends Animal { public  
13     void makeSound() {  
14         System.out.println("Woff woff!");  
15     };  
16 }  
16
```

```
18 公共类 AnimalTest {  
19     public static void main(String[] args){  
20         Cat cat = new Cat(); cat.makeSound();  
21         Dog dog = new Dog(); dog.makeSound();  
22     }  
22 }
```

Woff Woff!

方法覆盖示例 (2)

```
1  抽象类 动物 { 公共抽象 void  
2      makeSound();  
3  }  
4
```

```
5  
6  类 Cat extends Animal {  
7      public void makeNoise() {  
8          System.out.println("Meow!");  
9      };  
10 }  
11
```

```
12 类 Dog extends Animal { public  
13      void makeSound() {  
14          System.out.println("Woff woff!");  
15      };  
16 }  
16
```

```
18 公共类 AnimalTest {  
19     public static void main(String[] args){  
20         Cat cat = new Cat(); cat.makeSound();  
21         Dog dog = new Dog(); dog.makeSound();  
22     }  
22 }
```

方法覆盖示例 (3)

```
1 抽象类 动物 {
2      public void makeSound() {
3          System.out.println("Burp!");
4      }
5  }
6
7  猫类动物 {
8      public void makeNoise() {
9          System.out.println("Meow!");
10     };
11 }
```

Burp!

Woff Woff!

```
12 类 Dog 扩展动物 {
13     public void makeSound() {
14         System.out.println("Woff woff!");
15     };
16 }
16 公共类 AnimalTest {
18     public static void main(String[] args){
19         Cat cat = new Cat(); cat.makeSound();
20         Dog dog = new Dog(); dog.makeSound();
21     }
22 }
```

方法覆盖示例 (4)

```
1  抽象类 动物 {
2      public void makeSound() {
3          System.out.println("Burp!");
4      }
5  }
6  猫类动物 {
7      @Override
8      public void makeNoise() {
9          System.out.println("Meow!");
10     };
11 }
```

```
12 类 Dog 扩展动物 {
13     public void makeSound() {
14         System.out.println("Woff woff!");
15     };
16 }
16 公共类 AnimalTest {
18     public static void main(String[] args){
19         猫 cat = 新 Cat(); cat.makeSound();
20         Dog dog = new Dog(); dog.makeSound();
21     }
22 }
```

错误：方法没有覆盖或实现超类型的方法

动态绑定

- 动态绑定或（延迟绑定）：例如，Java 决定哪个类的方法在执行时调用，而不是在编译时调用
 - 超类引用只能用于调用超类的方法--子类方法的实现是多态调用的
- 试图在超类引用上直接调用子类专用方法会导致编译错误
- 操作符 **instanceof** 可用于检查对象是否可以铸成某一类型

多态处理示例

```
1  类 TestShape {  
2      public static void main(String[] args) {  
3          Shape[] shapes = new Shape[3];  
4          shapes[0] = new Circle(3.0);  
5          shapes[1] = 新矩形 (5.0,2.0) ;  
6          shapes[2] = 新矩形 (4.0,4.0) ;  
7          double totalArea = 0.0;  
8          for(int i=0; i<shapes.length; i++)  
9              totalArea += shapes[i].area();  
10         System.out.println ("总面积: " + totalArea) ;  
11     }  
12 }
```


使用 instanceof 的示例

```
1  类 TestShapeInstanceof {  
2      public static void main(String[] args) {  
3          Shape shape;  
4          shape = new Circle(5);  
5          if (shape instanceof Rectangle) {  
6              System.out.println("Shape is Rectangle!");  
7          }  
8          if (shape instanceof Circle) {  
9              System.out.println("Shape is Circle!");  
10             ;  
11         }  
12     }  
$ j  
Sha }
```


向子类转换的示例

```
1  类 ShrinkShape2 {  
2      public static void main(String[] args) {  
3          形状 shape = 新矩形 (1.0,3.0) ;  
4          System.out.println("Original area: " + shape.area());  
5          if(shape instanceof Rectangle) {  
6              矩形 rect = (Rectangle)shape;  
7              double w = rect.getWidth();  
8              double h = rect.getHeight();  
9              shape = new Rectangle(w/2, h/2);  
10             System.out.println("New area: " + shape.area());  
11         }  
12     }
```

```
13 }
```

```
$ java ShrinkShape2 原始面
```

```
积: 3.0
```

```
新区 0.75
```

获取班级信息

- 每个对象都*知道自己的类*，并可通过 **getClass()** 方法访问这些信息。
 - getClass 方法会返回一个 **Class**（来自 java.lang 包）类型的对象，其中包含有关对象的类型，包括其类名
 - 请注意，关键字 **class** 和类 **Class** 是不同的东西！
 - getClass 调用的结果用于调用 **getName()**，以获取对象的类名

getClass() 示例

```
1  抽象类 动物 {
2
3      public abstract void makeSound();
4  }
5
6  猫类动物 {
7      public void makeSound() {System.out.println("Meow!");}
8  }
9
10 类 Dog 扩展动物 {
11      public void makeSound() {System.out.println("Woff woff!");}
12  }
13
14公共类 AnimalGetClass {
15    public static void main(String[] args) {
16        Animal animal = new Cat();
17        Class cl = animal.getClass();
18        System.out.println("Animal is " + cl.getName());
19    }
20}
```

最终方法和类

- 超类中的最终方法不能在子类中重写
 - 声明为私有的方法是隐式最终方法，因为它是不可能
 - 声明为静态的方法隐含为最终方法
 - 最终方法的声明永远不会改变，因此所有子类都使用相同的方法实现，对最终方法的调用在编译时就已解决，这就是所谓的静态绑定。
- 最终类不能扩展以创建子类

- 最终类中的所有方法都是隐式最终方法

Java 应用程序接口中的最终类

- 字符串类是最终类的一个例子
 - 如果允许创建字符串的子类，那么该子类的对象就可以在任何需要使用字符串的地方使用了
 - 由于字符串类无法扩展，因此使用字符串的程序可以依赖 Java API 中指定的字符串对象功能。
 - 将类设为最终类还能防止程序员创建子类，从而绕过安全限制

- 请注意，在 JAVA 应用程序接口中，大多数类都**没有**声明为最终类

从构造函数调用方法

- **不要在构造函数中调用可重载的方法**: 在创建子类对象时, 这可能导致在子类对象完全初始化之前就调用了重载方法
 - 回想一下, 当你构造一个子类对象时, 它的构造函数会 **首先**调用直接超类的一个构造函数
 - 如果超类构造函数调用了可重载方法, 子类版本的方法将被超类构造函数调用--在子类构造函数的主体执行之前
 - 如果子类方法依赖于
子类构造函数尚未进行初始化

- 不过，从构造函数中调用静态方法是可以接受的

向子类转换的示例

```
1  抽象类 动物 {
2      公共 动物 ( ) {
3          System.out.println ("调用了构造函数 Animal") ;
4      }
5  }
6  抽象类 Mammal extends Animal {
7      public Mammal () {
8          System.out.println ("调用了构造函数 Mammal") ;
9      }
10 }
```

```
11 类 Cat 扩展哺乳动物 {
12      public Cat () {
13          System.out.println ("调用了构造函数 Cat") ;
14      }
15 }
```

```
16 公共类 ConstructorExample1 {
```

```
17      public static void main(String[] args)
18      {
```

```
18         Cat cat = new Cat();  
19     }  
20 }
```

\$ java 构造函数示例1 调用构造

函数动物 调用构造函数哺乳动物

调用构造函数猫

向子类转换的示例

```
1  抽象类 动物 {
2      public String sound() { return "nothing"; }
3      公共 动物( ) {
4          System.out.println("Animal says " + sound());
5      }
6  }
7  猫类动物 {
8      public String sound() { return "meow"; }
9      public Cat() {
10         System.out.println("Cat says " + sound());
11     }
12 }
13 公共类 ConstructorExample2 {
14     public static void main(String[] args) {
15         Cat cat = new Cat();
16     }
17 }
```


有问题或意见？

JC2002 Java 程序设计

第 4 天，会议 2：界面

界面

- 有了**接口**，互不相关的类就可以实现一套通用方法：人和系统可以通过接口以标准化的方式进行交互
- 举例说明：收音机上的控制按钮是无线电用户 无线电内部组件
 - 提供一套有限的操作（如更换电台、调节音量、在调幅和调频之间进行选择）
 - 不同的无线电设备可能会以不同的方式实现控制（如使用按钮、拨号盘、语音指令）

- 接口规定了无线电必须允许用户控制的操作，但没有规定***如何***进行操作

Java 中的接口

- Java 界面描述了一组可在以下情况下调用的方法物件
- *接口声明*以关键字 **interface** 开头，通常只包含常量和抽象方法
 - 所有接口成员必须是公共的
 - 接口中声明的强制方法隐含为公共抽象方法
 - 所有字段都是隐式公共、静态和最终字段

- 接口不能被实例化，因此它没有定义构造函数

在类中使用接口

- 要使用接口，具体类必须指定它实现了接口，并用指定的签名声明接口中的每个方法
- 没有实现接口所有方法的类是一个抽象类，必须声明为抽象类。
 - 实现接口就像与编译器签订合同：*"我将声明接口指定的所有方法，否则我将声明我的抽象类"*。

使用界面示例

```
1  抽象类 动物 {
2      protected boolean hungry = true;
3  }
4  接口 Feeder {
5      public void feed();
6  }
7  类 Cat extends Animal 实现 Feeder {
8      公共 void feed() {
9          hungry = false;
10     }
11 }
12 公共类 InterfaceExample1 {
13     public static void main(String[] args) {
14         Cat cat = new Cat();
15         cat.feed();
16     }
17 }
```

```
16         System.out.print("Is the cat hungry? ");
17         System.out.println(cat.hungry ? "Yes" : "No");
18     }
19 }
```

```
$ java InterfaceExample1
```

投遞餓了嗎？不餓

Java 界面的新功能

- 从 Java SE 8 开始，接口还可包含具有具体默认实现的公共默认方法，这些默认实现可指定在未重载的情况下如何执行操作
 - 如果一个类实现了这样的接口，该类也会收到接口的默认实现（如果有的话）
 - 要声明默认方法，请在方法的返回类型，并提供一个具体方法的实现
- 从 JAVA SE 8 开始，接口可包含静态方法
- 从 JAVA SE 9 开始，接口也可以包含私有方法、但是，定义受保护方法会导致编译错误

带有默认方法的接口示例

```
1  抽象类 动物 {
2      protected boolean hungry = true;
3  }
4  接口 Feederable {
5      公共默认 void feed() {
6          System.out.println("No method for feeding!");
7      }
8  }
9  类 Cat extends Animal 实现 Feederable {
10 }
11 公共类 InterfaceExample2 {
12      public static void main(String[] args) {
13          Cat cat = new Cat();
14          cat.feed();
15          System.out.print("Is the cat hungry? ");
16      }
17  }
```

```
16         System.out.println(cat.hungry ? "Yes" : "No");
17     }
18 }
```

```
$ java InterfaceExample2
```

没有进料方法!

猫饿了吗? 饿

使用多个接口

- Java 不允许子类从一个以上的超类继承（多重继承）；但是，一个类可以从一个超类继承，*并*根据~~需要~~实现多个接口
- 要实现多个接口，请使用以逗号分隔的接口列表
类声明中关键字 implements 后的名称，如

```
public class Subclass extends Superclass implements  
    FirstInterface, SecondInterface {
```

- Java API 包含大量接口，而许多 Java API
方法接受接口参数并返回接口值

何时使用接口

- 当不同的类（即不相关的类）需要共享共同的方法和常量
 - 通过响应 *相同* 的方法调用，允许多态处理不相关类的对象
 - 您可以创建一个描述所需功能的接口，然后在任何需要该功能的类中实现该接口
- 当没有默认实现可继承时，应使用接口代替抽象类
- 与公共抽象类一样，接口通常也是公共的

- 公共接口必须在与接口名称相同的文件中声明，文件扩展名为
.java

多个界面中的相同方法

- 如果一个类实现了两个接口，而这两个接口都定义了一个同名的默认方法，那么该类必须覆盖该方法并提供一个实现
- 可以使用
语法如下

```
InterfaceName.super.method( );
```

带有默认方法的接口示例

```
1  接口 钢琴家 {  
2      default void play() { System.out.println("Bling blong"); }  
3  }  
4  接口 小提琴家 {  
5      default void play() { System.out.println("Viih vooh"); }  
6  }  
7  类 音乐家 实现 钢琴家、小提琴家 {  
8      public void play() {  
9          Pianist.super.play();  
10     }  
11 }  
12 公共类 MusicianExample {  
13     public static void main(String[] args) {  
14         new Musician().play();
```

```
15     }  
16 }
```

```
$ java InterfaceExampleMusician  
Bling blong
```


扩展接口

- 与类一样，接口也可以扩展
 - 扩展接口继承超级接口的所有方法
- 一个接口可以扩展多个超级接口
- 实现该接口的类必须实现接口直接定义的抽象方法，以及从所有超接口继承的所有抽象方法

扩展接口示例

```
1 interface Scalable { void scale(double scaler); }
2 interface Rotatable { void rotate(); }
3 接口 Transformable extends Scalable, Rotatable {}
4 类 Rectangle 实现 Transformable {
5     public double w, h;
6     public Rectangle(double w, double h) { this.w = w; this.h = h; }
7     public void scale(double scaler) { this.w *= scaler; this.h *= scaler; }
8     public void rotate() {
9         double temp = this.w; this.w = this.h; this.h = temp; }
10 }
11 公共类 TransformableExample {
12     public static void main(String[] args) {
13         Rectangle rect = new Rectangle(10.0, 5.0);
14         rect.scale(0.5);
```

```
15         System.out.printf("New dimensions: %f,%f\n", rect.w, rect.h);
16     }
17 }
```

```
$ java InterfaceExampleTransformable 新
```

```
尺寸: 5.000000,2.500000
```

功能界面

- 从 Java SE 8 开始，任何只包含一个抽象方法的接口都称为 *功能接口*，也称为 SAM（单抽象方法）接口
- 可以使用可选注解 **@FunctionalInterface**
- Java API 中定义的功能接口示例：
 - **比较器** （Deitel 书中的第 16 章） - 实现该接口可定义一种方法，用于比较给定类型的两个对象，以确定第一个对象是否小于、等于或大于第二个对象

- **Runnable**（Deitel 书中第 23 章）--实现该接口可定义与程序其他部分并行运行的任务

功能界面示例

```
1  @FunctionalInterface
2  接口 Talkable {
3      void talk(String msg);
4  }
5  公共类 FunctionalInterfaceExample {
6      public static void main(String[] args) {
7          Talkable person = new Talkable() {
8              public void talk(String msg) {
9                  System.out.println(msg);
10             }
11         };
12         person.talk ("世界你好! ");
13     }
14 }
```

世界你好

Lambda 表达式

- *Lambda 表达式*是 Java SE 8 中引入的一项新功能，允许来表示功能接口的单一方法
- lambda 表达式的格式： **(参数列表) -> { 主体 }**
 - 参数列表可以为空 () 或包含一个或多个参数
 - 主体包含方法的实现
- Lambda 表达式用于 *函数式编程*
 - 我们将在本课程稍后部分详细讨论 lambda 表达式

lambda 表达式示例 (1)

```
1  @FunctionalInterface
2  接口 Talkable {
3      void talk(String msg);
4  }
5  公共类 LambdaExample1 {
6      public static void main(String[] args) {
7          Talkable person = (msg) -> {System.out.println(msg);};
8          person.talk("世界你好!");
9      }
10 }
```


lambda 表达式示例 (2)

```
1  @FunctionalInterface
2  接口 Talkable {
3      void talk(String msg);
4  }
5  公共类 LambdaExample2 {
6      public static void main(String[] args) {
7          Talkable person = (msg) -> {System.out.println(msg);};
8          person.talk("世界你好!");
9          Talkable quietPerson =
10             (msg) -> {System.out.println("Shh!");};
11         quietPerson.talk("Hello world!");
```

```
12     }
```

```
13 }
```

```
$ java LambdaExample2
```

```
Hello world!
```

```
嘘
```

摘要

- 抽象类是包含方法的类，没有具体的执行
 - 抽象方法作为具体方法的 "占位符"
抽象类子类中的实现
 - 有助于将功能的定义和实施分开
 - 接口定义了一系列通用功能，就像抽象类一样
- 界面是一种 "协议"，规定了你的班级能做什么
- Java 不支持多重继承，但类似的效果可以通过通过实施多个接口来实现

- 功能接口是一种完全包含一个抽象类的接口类型

有问题或意见？