抽象类_接口

@M了个J
李明杰

https://github.com/CoderMJLee
http://cnblogs.com/mjios

码拉松

小码哥教育 SEEMYGO
实力IT教育 www.520it.com

抽象方法（Abstract Method）

- 抽象方法：被 abstract 修饰的实例方法
- 只有方法声明，没有方法实现（参数列表后面没有大括号，而是分号）
- 不能是 private 权限（因为定义抽象方法的目的让子类去实现）
- 只能定义在抽象类、接口中

小码哥教育 @M了个J

抽象类（Abstract Class）

- 抽象类：被 `abstract` 修饰的类
  - 可以定义抽象方法
  - 不能实例化，但可以自定义构造方法
  - 子类必须实现抽象父类中的所有抽象方法（除非子类也是一个抽象类）
  - 可以像非抽象类一样定义成员变量、常量、嵌套类型、初始化块、非抽象方法等
  - ✓ 也就说，抽象类也可以完全不定义抽象方法

- 常见使用场景
  - 抽取子类的公共实现到抽象父类中，要求子类必须要单独实现的定义成抽象方法

小码哥教育 SEEMYGO

抽象类实例 – 父类

```
public abstract class Shape {
    protected double area;
    protected double girth;
    public double getArea() {
        return area;
    }
    public double getGirth() {
        return girth;
    }
    public void show() {
        calculate();
        System.out.println(area + "_" + girth);
    }
    protected abstract void calculate();
}
```

小码哥教育 SEEMYGO

抽象类实例 – 子类

```
public class Rectangle extends Shape {
    private double width;
    private double height;
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }
    @Override
    protected void calculate() {
        area = width * height;
        girth = (width + height) * 2;
    }
}
```

```
Rectangle rectangle = new Rectangle(10, 20);
rectangle.show();
// 200.0_60.0
```

小码哥教育 SEEMYGO

抽象类实例 – 子类

```
public class Circle extends Shape {
    private double radius;
    public Circle(double radius) {
        this.radius = radius;
    }
    @Override
    protected void calculate() {
        double half = Math.PI * radius;
        area = half * radius;
        girth = half * 2;
    }
}
```

```
Circle circle = new Circle(10);
circle.show();
// 314.1592653589793_62.83185307179586
```

接口（Interface）

- 看到“接口”二字首先想到的什么？
  - 网线接口？USB 接口？

- 接口的英文单词是 Interface， 这个单词是否很熟悉？

- API（Application Programming Interface）
  - 应用编程接口，提供给开发者调用的一组功能（无须提供源码）

- Java 中的接口
  - 一系列方法声明的集合
  - 用来定义规范、标准

接口中可以定义的内容

- 可以定义：抽象方法、常量、嵌套类型，从 Java 8 开始可以定义：默认方法、静态方法（类方法）
  - 上述可以定义的内容都是隐式 `public` 的，因此可以省略 `public` 关键字
  - 从 Java 9 开始可以定义 `private` 方法
- 常量可以省略 `static`、`final`
- 抽象方法可以省略 `abstract`
- 不能自定义构造方法、不能定义（静态）初始化块、不能实例化

接口的细节

- 接口名称可以在任何使用类型的地方使用

- 一个类可以通过 implements 关键字实现一个或多个接口
  - 实现接口的类必须实现接口中定义的所有抽象方法，除非它是个抽象类
  - 如果一个类实现的多个接口中有相同的抽象方法，只需要实现此方法一次
  - extends 和 implements 可以一起使用，implements 必须写在 extends 的后面
  - 当父类、接口中的方法签名一样时，那么返回值类型也必须一样

- 一个接口可以通过 extends 关键字继承一个或者多个接口
  - 当多个父接口中的方法签名一样时，那么返回值类型也必须一样

小码哥教育 SEEMYGO

接口的升级问题

- 如果接口需要升级，比如增加新的抽象方法
  - 会导致大幅的代码改动，以前实现接口的类都得改动

- 若想在不改动以前实现类的前提下进行接口升级，从 Java 8 开始，有 2 种方案
  - 默认方法（Default Method）
  - 静态方法（Static Method）

小码哥教育 @M了个J

小码哥教育 SEEMYGO

默认方法（Default Method）

- 用 default 修饰默认方法
- 默认方法只能是实例方法

```
public interface Eatable {
    default void eat(String name) {
        System.out.println("Eatable - eat - " + name);
    }
}
```

```
public class Dog implements Eatable {}
```

```
public class Cat implements Eatable {
    @Override
    public void eat(String name) {
        Eatable.super.eat(name);
        System.out.println("Cat - eat - " + name);
    }
}
```

```
Dog dog = new Dog();
dog.eat("bone");
// Eatable - eat - bone

Cat cat = new Cat();
cat.eat("fish");
// Eatable - eat - fish
// Cat - eat - fish
```

默认方法的使用

- 当一个类实现的接口中有默认方法时，这个类可以
  - 啥也不干，沿用接口的默认实现
  - 重新定义默认方法，覆盖默认方法的实现
  - 重新声明默认方法，将默认方法声明为抽象方法（此类必须是抽象类）

- 当一个接口继承的父接口中有默认方法时，这个接口可以
  - 啥也不干，沿用接口的默认实现
  - 重新定义默认方法，覆盖默认方法的实现
  - 重新声明默认方法，将默认方法声明为抽象方法

小码哥教育 SEEMYGO

默认方法的细节

- 如果父类定义的非抽象方法与接口的默认方法相同时，最终将调用父类的方法

```
public class Animal {
    public void run() {
        System.out.println("Animal - run");
    }
}
```

```
public interface Runnable {
    default void run() {
        System.out.println("Runnable - run");
    }
}
```

```
public class Dog extends Animal implements Runnable {}
```

```
Dog dog = new Dog();
dog.run(); // Animal - run
```

默认方法的细节

- 如果父类定义的抽象方法与接口的默认方法相同时，要求子类实现此抽象方法
- 可以通过 super 关键字调用接口的默认方法

```
public interface Runnable {
    default void run() {
        System.out.println("Runnable - run");
    }
}
```

```
public abstract class Animal {
    public abstract void run();
}
```

```
public class Dog extends Animal implements Runnable {
    @Override
    public void run() {
        Runnable.super.run();
        System.out.println("Dog - run");
    }
}
```

```
Dog dog = new Dog();
dog.run();
// Runnable - run
// Dog - run
```

默认方法的细节

- 如果（父）接口定义的默认方法与其他（父）接口定义的方法相同时，要求子类型实现此默认方法

```
public interface Runnable {
    default void run() {
        System.out.println("Runnable - run");
    }
}
```

```
public interface Walkable {
    default void run() {
        System.out.println("Walkable - run");
    }
}
```

```
public interface Testable extends Runnable, Walkable {
    @Override
    default void run() {
        Runnable.super.run();
        Walkable.super.run();
        System.out.println("Testable - run");
    }
}
```

```
public class Dog
        implements Runnable, Walkable {
    @Override
    public void run() {
        Runnable.super.run();
        Walkable.super.run();
        System.out.println("Dog - run");
    }
}
```

```
Dog dog = new Dog();
dog.run();
// Runnable - run
// Walkable - run
// Dog - run
```

小码哥教育 SEEMYGO

默认方法的细节

```
public interface Animal {
    default String myself() {
        return "I am an animal.";
    }
}
```

```
public interface Fire extends Animal {}
```

```
public interface Fly extends Animal {
    default String myself() {
        return "I am able to fly.";
    }
}
```

```
public class Dragon implements Fly, Fire {}
```

```
Dragon dragon = new Dragon();
System.out.println(dragon.myself());
// I am able to fly.
```

静态方法（Static Method）

- 接口中定义的静态方法只能通过接口名调用，不能被继承

```
public interface Eatable {
    static void eat(String name) {
        System.out.println("Eatable - eat - " + name);
    }
}
```

```
public interface Sleepable {
    static void eat(String name) {
        System.out.println("Sleepable - eat - " + name);
    }
}
```

```
public interface Dog extends Sleepable, Eatable {
    static void eat(String name) {
        System.out.println("Dog - eat - " + name);
    }
}
```

```
Dog.eat("1"); // Dog - eat - 1
Eatable.eat("2"); // Eatable - eat - 2
Sleepable.eat("3"); // Sleepable - eat - 3
```

抽象类与接口对比

- 抽象类和接口的用途还是有点类似，该如何选择?

- 何时选择抽象类?
  - 在紧密相关的类之间共享代码
  - 需要除 `public` 之外的访问权限
  - 需要定义实例变量、非 `final` 的静态变量

- 何时选择接口?
  - 不相关的类实现相同的方法
  - 只是定义行为，不关心具体是谁实现了行为
  - 想实现类型的多重继承

多态（Polymorphism）

- 什么是多态?
  - 具有多种形态
  - 同一操作作用于不同的对象，产生不同的执行结果

- 多态的体现
  - 父类（接口）类型指向子类对象
  - 调用子类重写的方法

- JVM 会根据引用变量指向的具体对象来调用对应的方法
  - 这个行为叫做：虚方法调用（virtual method invocation）
  - 类似于 C++ 中的虚函数调用

小码哥教育 @M了个M

小码哥教育 SEEMYGO

多态实例

```
public class Animal {
    public void speak() {
        System.out.println("Animal - speak");
    }
}

public class Dog extends Animal {
    @Override
    public void speak() {
        System.out.println("Dog - wangwang");
    }
}

public class Cat extends Animal {
    @Override
    public void speak() {
        System.out.println("Cat - miaomiao");
    }
}
```

```
public static void main(String[] args) {
    speak(new Dog()); // Dog - wangwang
    speak(new Cat()); // Cat - miaomiao
}
static void speak(Animal animal) {
    animal.speak();
}
```

小码哥教育 SEEMYGO

多态实例

```
public interface Runnable {
    void run();
}

public class Pig implements Runnable {
    @Override
    public void run() {
        System.out.println("Pig - run");
    }
}

public class Person implements Runnable {
    @Override
    public void run() {
        System.out.println("Person - run");
    }
}
```

```
public static void main(String[] args) {
    run(new Pig()); // Pig - run
    run(new Person()); // Person - run
}
static void run(Runnable runnable) {
    runnable.run();
}
```

小码哥教育 SEEMYGO

类方法的调用细节

```
public class Animal {
    public static void run() {
        System.out.println("Animal - run");
    }
}
```

```
public class Dog extends Animal {
    public static void run() {
        System.out.println("Dog - run");
    }
}
```

```
Dog.run(); // Dog - run
Animal.run(); // Animal - run

Dog dog1 = new Dog();
dog1.run(); // Dog - run

Animal dog2 = new Dog();
dog2.run(); // Animal - run
```

小码哥教育 SEEMYGO

成员变量的访问细节

```java
public class Person {
    public int age = 1;
    public int getPAge() {
        return age;
    }
}
```

```java
public class Student extends Person {
    public int age = 2;
    public int getSAge() {
        return age;
    }
}
```

```java
Student stu1 = new Student();
System.out.println(stu1.age); // 2
System.out.println(stu1.getPAge()); // 1
System.out.println(stu1.getSAge()); // 2

Person stu2 = new Student();
System.out.println(stu2.age); // 1
System.out.println(stu2.getPAge()); // 1
```

```java
public class Teacher extends Person {
    public int age = 3;
    @Override
    public int getPAge() {
        return age;
    }
    public int getTAge() {
        return age;
    }
}
```

```java
Teacher tea1 = new Teacher();
System.out.println(tea1.age); // 3
System.out.println(tea1.getPAge()); // 3
System.out.println(tea1.getTAge()); // 3

Person tea2 = new Teacher();
System.out.println(tea2.age); // 1
System.out.println(tea2.getPAge()); // 3
```

小码哥教育 SEEMYGO

instanceof

- 可以通过 instanceof 判断某个类型是否属于某种类型

```
public class Animal {}

public interface Runnable {}

public class Dog extends Animal implements Runnable {}

Object dog = new Dog();
System.out.println(dog instanceof Dog); // true
System.out.println(dog instanceof Animal); // true
System.out.println(dog instanceof Runnable); // true
System.out.println(dog instanceof String); // false
```

instanceof 使用

```
public class Animal {}

public class Dog extends Animal {
    public void wang() {
        System.out.println("Dog - wang");
    }
}

public class Cat extends Animal {
    public void miao() {
        System.out.println("Cat - miao");
    }
}
```

```
public static void main(String[] args) {
    speak(new Dog()); // Dog - wang
    speak(new Cat()); // Cat - miao
}

static void speak(Animal animal) {
    if (animal instanceof Dog) {
        ((Dog) animal).wang();
    } else if (animal instanceof Cat) {
        ((Cat) animal).miao();
    }
}
```

小码哥教育 SEEMYGO

对象数组的注意点

```
Object obj1 = 11;
Integer obj2 = (Integer) obj1;
System.out.println(obj2);

// Object[] objs1 = new Object[] { 11, 22, 33 };
Object[] objs1 = { 11, 22, 33 };
// java.lang.ClassCastException
// [Ljava.lang.Object; cannot be cast to [Ljava.lang.Integer;
Integer[] objs2 = (Integer[]) objs1;
System.out.println(objs2);
```