福建師範大學
FUJIAN NORMAL UNIVERSITY

# Java语言

学　　期：2025-2026-1

教　　师：赵珊珊

# 第5章面向对象（二）

# 第5章面向对象（二）

- 5.1类的继承
- 5.2抽象类和接口
- 5.3多态

# 5.1类的继承

- 5.1.1继承的概念
  - 利用继承机制。可以先创建一个具有共性的一般类，从一般类再派生出具有特殊性的新类，新类继承一般类的属性和方法，并根据需要增加它自己新的属性和新的方法。

# 5.1类的继承

- 5.1.1继承的概念
  - 类的继承也称类的派生。通常，将被继承的类称为父类或超类，派生出来的类称为子类，从一个父类可以派生出多个子类，子类还可以派生出新的子类，这样就形成了类的层次关系。在Java中一个类只能继承一个父类，称为单继承。但一个父类却可以派生出多个子类，每个子类作为父类又可以派生出多个子类，从而形成具有树形结构的类的层次体系。

# 5.1类的继承

- 1.Java中，子类继承父类的关键字：extends
- Java语言只支持单继承，不允许多重继承，即一个子类只能继承一个父类
- Java语言虽然不支持多重继承，但它支持多层继承，即一个类的父类可以继承另外的父类。

# 5.1类的继承

- 2.父类成员的访问权限
  - 子类可以继承父类中的属性成员和除构造方法以外的方法成员，但不能继承父类的构造方法。而且，并不是对父类的所有属性成员和方法成员都具有访问权限，即并不是在子类声明的方法中能够访问父类中的所有属性成员和方法成员。

# 5.1类的继承

- 2.父类成员的访问权限
  - 子类对父类的private成员没有访问权限。既不能直接引用父类中的private属性成员，也不能调用父类中的private方法成员，如果需要访问父类的成员，可以通过父类中的非private成员方法来引用父类的成员。
  - 子类对父类的public或protected成员具有访问权限。
  - 子类对父类的默认权限成员的访问分为两种情况：一是对同一包中父类的默认权限成员具有访问权限；二是对其他包中父类的默认权限成员没有访问权限。

# 5.1类的继承

- 5.1.2成员变量的隐藏
- 类的继承使得子类从父类中既继承了有用的属性成员，也会继承一些不需要或不恰当的属性成员。当父类中的属性不适合子类需要时，子类可以把从父类继承来的属性成员进行重新定义。由于在子类中也定义了与父类中名字相同的成员变量，因此父类中的成员变量在子类中就不可见了，这就是成员变量的隐藏。这时在子类中若想引用父类中的同名变量，可以用关键词super作为前缀来引用父类中同名变量。即：
- super.属性名

# 5.1类的继承

- 5.1.2成员变量的隐藏

```
class Construct{
    public String Floor="1";//楼层号
    protected void printFloor() {
        System.out.println("楼层为："+Floor);
    }
}
public class Building extends Construct{
    String Floor;//房间号
    Building(String Floor){
        System.out.println("超类的楼层号为"+super.Floor);
        this.Floor = Floor;
    }
    public void printFloor() {
        System.out.println("房间号为："+Floor);
    }
    public static void main(String[] args) {
        Building b=new Building("101");
        b.printFloor();
    }
}
```

```
超类的楼层号为1
房间号为：101
```

# 5.1类的继承

- 5.1.3方法的重写
- 在继承关系中，子类从父类中继承了可访问的方法，但有时从父类继承下来的方法不能完全满足子类需要，这时就需要在子类方法中修改父类方法，即子类重新定义从父类中继承的成员方法，这个过程称为方法重写或覆盖。

# 5.1类的继承

- 5.1.3方法的重写
- 子类方法覆盖父类的方法时，方法头要与父类一样，即两个方法要具有完全相同的方法名、返回类型、参数表。方法体要根据子类的需要重写，从而满足子类功能的要求。与类中使用父类被隐藏的属性成员类似，如果子类中需要调用被覆盖的父类中的同名方法，通过super关键字作前缀来调用，即：
- super.方法名()

# 5.1类的继承

- 方法重载与方法重写的区别：
  - 方法重载是在同一个类中，方法重写是在子类与父类中。
  - 方法重载要求：方法名相同，参数个数或参数类型不同。
  - 方法重写要求：子类与父类的方法名、返回值类型和参数列表相同。

# 5.1类的继承

- 5.1.4super关键字
- Java用关键字super表示父类对象，因此在子类中使用super作为前缀可以引用被子类隐藏的父类的变量和被子类覆盖的父类的方法。
- 当子类中没有声明与父类同名的成员变量时，引用父类的成员变量可以不使用super。当子类中没有声明与父类中同名的成员方法时，调用父类的成员方法也可以不使用super。

# 5.1类的继承

- 5.1.4super关键字
- 其语法格式如下：
  - super.成员变量
  - super.成员方法
  - super([参数列表])

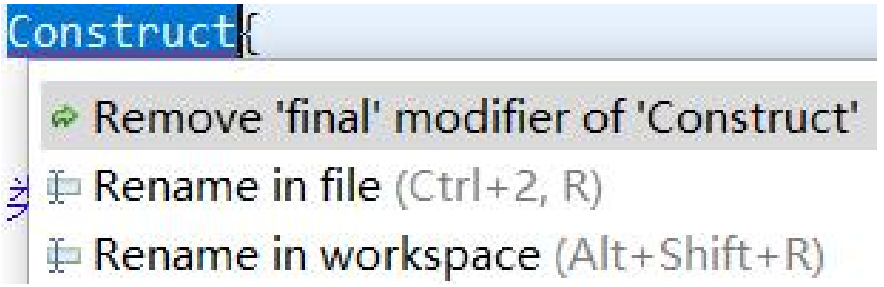
super([参数列表])

必须放在子构造方法的第一行

# 5.1类的继承

- 5.1.5final关键字
  - final修饰的类不能被继承。
  - final修饰的方法不能被子类重写。
  - final修饰的变量是常量，初始化后不能再修改。

```
final class Construct{
    public String Floor="1";//楼层号
    protected void printFloor() {
        System.out.println("楼层为："+Floor);
    }
}
public class Building extends Construct{
```




The type Building cannot subclass the final class Construct

# 5.2抽象类和接口

- 5.2.1抽象类
- Java用abstract关键字，表示抽象的意思，用abstract修饰的方法，称为抽象方法，是一个不完整的方法，只有方法的声明，没有方法体。用abstract修饰的类，称为抽象类，语法如下：

```
[权限修饰符] abstract class 类名{
 [权限修饰符] abstract 返回值类型 方法名(参数列表);
}
```

# 5.2抽象类和接口

- 5.2.1抽象类
  - 抽象方法声明只需给出方法头，不需要方法体，直接以“;”结束。
  - 构造方法不能声明为抽象方法。
  - 在抽象类中，可以包含抽象方法，也可以不包含抽象方法，但类中如果有抽象方法，此类必须声明为抽象类。
  - 抽象类不能被实例化，即使抽象类中没有抽象方法，也不能被实例化。
  - 子类必须实现抽象父类中所有抽象方法，否则子类必须要声明为抽象类。
  - 抽象方法不能用static、final或private来修饰。

# 5.2抽象类和接口

- 5.2.2接口
- 接口是全局常量和公共抽象方法的集合，接口可被看作一种特殊的抽象类，也属于引用类型。每个接口都被编译成独立的字节码文件。Java提供interface关键字，用于声明接口，其语法格式如下：

```
interface 接口名{
    全局常量声明;
    抽象方法声明;
}
```

# 5.2抽象类和接口

- 5.2.2接口
  - 可以在抽象类中定义方法的默认行为，但是接口中的方法不能有默认行为。
  - 如果没有指定接口方法和变量的访问权限，Java将其默认为public。
  - exntends表示的是一种单继承关系，而一个类却可以实现多个接口，表示的是一种多继承。
  - 接口中定义的变量和方法都包含默认的修饰符，其中定义的变量默认声明为“public static final”，即全局常量。定义的方法默认声明为“public abstract”，即抽象方法。
  - 如果一个类实现了某个接口，那么必须实现这个接口所有的抽象方法，否则这个类是抽象类。

# 5.2抽象类和接口

- 5.2.2接口
- 1.接口的实现
- Java提供implements关键字，用于实现多个接口，其语法格式如下：

```
class 类名 implements 接口{
    属性和方法
}
```

# 5.2抽象类和接口

- 5.2.2接口
- 2.接口的继承
- 接口支持多重继承，即一个接口可以继承多个父接口。其语法格式如下：

```
interface 接口名 extends 接口名1, 接口名2{
    全局常量声明
    抽象方法声明
}
```

```
class 类名 implements 接口名1, 接口名2{
    属性和方法
}
```

```
class 类名 extends 类{
    属性和方法
}
```

# 5.2抽象类和接口

- 5.2.2接口
- 3.使用接口的好处
  - （1）声明引用时要使用接口类型。
  - （2）方法的参数要声明成接口类型。
  - （3）方法的返回值要声明成接口类型。

# 5.2抽象类和接口

- 5.2.3抽象类和接口的关系

| 区别 | 接口 | 抽象类 |
| --- | --- | --- |
| 含义 | 接口通常用于描述一个类的外围能力，而不是核心特征。类与接口之间是-able或can do的关系。 | 抽象类定义了它的继承类的核心特征。派生类与抽象类之间是is-a的关系。 |
| 方法 | 接口只提供方法声明 | 抽象类可以提供完整方法、默认构造方法以及用于覆盖的方法声明 |
| 变量 | 只包含public static final常量，常量必须在声明时初始化 | 可以包含实例变量和静态变量 |
| 多重继承 | 一个接口可以继承多个接口 | 一个类只能继承一个抽象类 |
| 实现类 | 类可以实现多个接口 | 类只从抽象类派生，必须重写 |
| 适用性 | 所有的实现只能共享方法签名 | 所有实现大同小异，并且共享状态和行为 |

# 5.2抽象类和接口

```
建筑面积: 1500.0 平方米, 高度: 100.0 米
通过接口引用访问 - 面积: 1500.0
```

```java
// 定义水平建设接口
interface HorizonConstruct {
    double getArea(); // 获取建设用地面积
}
// 定义垂直建设接口
interface VerticalConstruct {
    double getHeight(); // 获取建筑高度
}
// Building 类实现两个接口
class Building implements HorizonConstruct, VerticalConstruct {
    private double length;
    private double width;
    private double height;
    public Building(double length, double width, double height) {
        this.length = length;
        this.width = width;
        this.height = height;
    }
    // 实现 HorizonConstruct 接口的方法
    public double getArea() {return length * width; }
    // 实现 VerticalConstruct 接口的方法
    public double getHeight() { return height; }
    // Building 类自己的方法
    public void displayInfo() {
        System.out.println("建筑面积: " + getArea() + " 平方米, 高度: " + getHeight() + " 米");
    }
}
public class Main {
    public static void main(String[] args) {
        Building myBuilding = new Building(50, 30, 100);
        myBuilding.displayInfo(); // 输出: 建筑面积: 1500.0 平方米, 高度: 100.0 米
        System.out.println("通过接口引用访问 - 面积: " + myBuilding.getArea());
        }
}
```

```
内侧建筑面积: 1275.0 平方米, 高度: 100.0 米
通过抽象类引用访问 - 面积: 1275.0
```

```java
// 定义抽象建设类
abstract class Construct {
    protected double length;
    protected double width;
    protected double height;
    public Construct(double length, double width, double height) {
        this.length = length;
        this.width = width;
        this.height = height;
    }
    // 抽象类可以包含具体实现的方法
    public double getArea() { return length * width; }
    // 抽象类定义抽象方法，留给子类实现
    public abstract double getHeight();
}
class Building extends Construct {
    public Building(double length, double width, double height) {
        // 调用父类构造方法初始化继承的属性
        super(length, width, height);
    }
    // 实现抽象方法
    public double getHeight() { return height; }
    // 可以重写父类方法
    public double getArea() {
        return super.getArea() * 0.85;
    }
    // Building 类自己的方法
    public void displayInfo() {
        System.out.println("内侧建筑面积: " +getArea() +" 平方米, 高度: " + getHeight() + " 米");
    }
}
public class Main {
    public static void main(String[] args) {
        Building myBuilding = new Building(50, 30, 100);
        myBuilding.displayInfo();
        // 抽象类引用可以指向子类对象
        Construct construct = myBuilding;
        System.out.println("通过抽象类引用访问 - 面积: " + construct.getArea());
    }
}
```

# 5.3多态

- 5.3.1多态的概念
- 多态意为一个名字可具有多个语义。在程序设计语言中，多态性是指“一种定义，多种实现”。

# 5.3多态

- 5.3.1多态的概念
- 多态有下面几个特点：
  - 对象类型不可变，引用类型可变。
  - 只能调用引用对应的类型中定义的方法。
  - 运行时会运行子类覆盖的方法。
- 多态实现的三个必要条件
  - 要有继承（实现implements）
  - 要有重写（overWrite&overRide)
  - 父类引用指向子类对象

  P102: 动态绑定

  只有实例方法表现出多态作用。

# 5.3多态

- 5.3.2向上转型和向下转型
  - 1.向上转型：是从子类到父类的转换，也称隐式转换。
  - 2.向下转型：是从父类到子类的转换，也称显式转换。

```
// 父类
class Construct {
    public void build() {
        System.out.println("正在进行基础建设...");
    }
}
// 子类
class Building extends Construct {
    public void build() { // 重写父类方法
        System.out.println("正在建造一栋大楼...");
    }
    public void operate() { // 子类特有方法
        System.out.println("大楼投入使用：办公、住宿、商业...");
    }
}
public class Main {
    public static void main(String[] args) {
        Construct construct = new Building(); // 子→父，发生了向上转型
        construct.build(); // 输出：正在建造一栋大楼...
        // construct.operate(); // 编译错误！父类引用无法调用子类特有方法
    }
}
```

```
正在建造一栋大楼...
```

```
Construct construct = new Building(); // 子→父，发生了向上转型
construct.build(); // 输出：正在建造一栋大楼...
//向下转型：将父类引用强制转回子类类型
Building building = (Building) construct; // 使用强制类型转换符 (Building)
building.build(); // 输出：正在建造一栋大楼...（仍然是重写后的方法）
building.operate(); // 输出：大楼投入使用：办公、住宿、商业...（成功调用子类特有方法）
```

```
正在建造一栋大楼...
大楼投入使用：办公、住宿、商业...
```

```
// 安全的向下转型：先检查，再转换
if (construct instanceof Building) { // 判断construct引用是否实际指向Building对象
    Building building = (Building) construct; // 安全地向下转型
    building.operate(); // 调用子类特有方法
} else {
    System.out.println("construct引用指向的对象不是Building类型，无法安全向下转型。");
}
```

```
正在建造一栋大楼...
正在建造一栋大楼...
大楼投入使用：办公、住宿、商业...
```

# 5.3多态

- 5.3.3Object类
- Java中提供了一个Object类，是所有类的父类，如果一个类没有显式地指定继承类，则该类的父类默认为Object。

# 5.3多态

- 5.3.3Object类
  - 1.toString()方法:调用一个对象的toString()方法会默认返回一个描述该对象的字符串，它由该对象所属类名、@和对象十六进制形式的内存地址组成。
  - 2.equals()方法:equals()方法是用于测试两个对象是否相等，比较地址。

```
public static void main(String[] args) {
    Construct construct = new Construct();
    Construct change = new Building(); // 子→父，发生了向上转型
    Building build = new Building();
    System.out.println("construct="+construct.toString());
    System.out.println("build → construct="+change.toString());
    System.out.println("build="+build.toString());
}
```

```
construct=Construct@3d012ddd
change=Building@626b2d4a
build=Building@5e91993f
```

```
class Building{
    private String add;
    private double area;
    Building(String add, double area){
        this.add = add;
        this.area =area;
    }
}
public class Main {
    public static void main(String[] args) {
        Building b1 = new Building("知名楼1-206",40);
        Building b2 = new Building("知名楼1-206",40);
        System.out.println(b1.equals(b2));
        System.out.println(b1==b2);
    }
}
```

```
false
false
```

# 5.3多态

- 5.3.4工厂设计模式
- 工厂模式（Factory）主要用来实例化有共同接口的类，它可以动态决定应该实例化哪一个类，不必事先知道每次要实例化哪一个类。工厂模式主要有三种形态：简单工厂模式、工厂方法模式和抽象工厂模式。