福建師範大學
FUJIAN NORMAL UNIVERSITY

# Java语言

学　　期：2025-2026-1

教　　师：赵珊珊

# 第4章面向对象（一）

# 第4章面向对象（一）

| | |
|---|---|
| • 4.1面向对象概念 | 封装性、继承性和多态性 |
| • 4.2类与对象的概念 | 类是封装对象属性和行为的载体，对象是类抽象出来的一个实例，<br>对象分静态（属性或变量）、动态（方法） |
| • 4.3方法 | 成员方法与构造方法的定义、调用和重载 |
| • 4.4关键字this的使用 | 调用成员变量、成员方法、其他构造方法 |
| • 4.5关键字static的使用 | 静态与实例的属性（变量）、方法 、代码块 |
| • 4.6内部类 | 成员内部类、静态内部类、方法内部类、匿名内部类 |
| • 4.7包 | 包的定义和使用，import 语句 |
| • 4.8类及成员的访问权限 | 类的访问权限：public和default默认<br>成员的访问权限：public、protected、default、private |

# 4.1面向对象概念

- 面向对象的特点主要可以概括为封装性、继承性和多态性

```
// 建筑用地基类Construction（封装性体现）
public class Construction {

}
```

```
double Area = 1000;//面积，m2
String Floor="1",RoomNO1="01",RoomNO2="02";//定义楼层，教室编号
int seetNumber1=30,seetNumber2=40;//教室座位数
String getNO() {
    return("屋面包括"+Floor+RoomNO1+"、"+Floor+RoomNO2);
}
```

```
//将Construction作为超类，设计Building子类来继承它（继承性体现）
class Building extends Construction{

}
```

多态性

```
String getNO(int no1,int no2) {
    return "教室总座位为"+(no1+no2)+"个";
}
public static void main(String[] args) {
    Building b =new Building();
    System.out.println(b.getNO());
    System.out.println(b.getNO(b.seetNumber1, b.seetNumber2));
}
```

# 4.2类与对象的概念

- 类实质上就是封装对象属性和行为的载体，而对象则是类抽象出来的一个实例。

```
class Building{
}
```

```
Building b =new Building();
```

# 4.2类与对象的概念

- 4.2.1类的定义
  - 类是对一个特定类型对象的描述，它定义了一种新类型，即“类”是对象的定义，用户也可以把它看做是对象的蓝图。
  - 类中可以包含有关对象属性和方法的定义。其中，属性是存储数据的变量，可以是任何类型，用户通过这些数据区分类的不同对象；方法定义了用户对类的各种操作，也就是类的对象可以做的事情，通常，方法是对属性进行操作的。

```
class Construction{
    double Area = 1000;//面积，m2
    String Floor="1",RoomNO1="01",RoomNO2="02";
    int seetNumber1=30,seetNumber2=40;
    String getNO() {
        return("屋面包括"+Floor+RoomNO1+"、"+Floor+RoomNO2);
    }
}
```

# 4.2类与对象的概念

- 4.2.2对象的定义和引用
  - 类是对象的抽象，为对象定义了属性和行为，但类本身既不带任何数据，也不存在于内存空间中。而对象是类的一个具体存在，既拥有独立的内存空间，也存在独特的属性和行为，属性还可以随着自身的行为而发生改变。
  - 类名 对象名=new 类名();

```
Building b =new Building();
System.out.println(b.getNO());
System.out.println(b.getNO(b.seetNumber1, b.seetNumber2));
```

# 4.2类与对象的概念

## • 4.2.2对象的定义和引用

- new创建对象时，每个对象的内存空间是独立的。
- 一个对象可以被多个变量引用，当对象不被任何变量引用时，该对象会成为垃圾数据，不再被使用。

```
public class Building{
    double Area = 1000;//面积，m2
    String Floor,RoomNO;//定义楼层，单层教室编号  多态性
    int seetNumber;//教室座位数
    public void BuildingInfo() {
        System.out.println("面积="+Area+";教室号="+Floor+RoomNO+";座位数="+seetNumber);
    }
    Building(String Floor,String RoomNO,int seetNumber){
        this.Floor=Floor;
        this.RoomNO=RoomNO;
        this.seetNumber = seetNumber;
    }
    public static void main(String[] args) {
        Building b1 = new Building("1","01",30);
        Building b2 = new Building("1","02",40);
        b2.Area=800;
        b1.BuildingInfo();
        b2.BuildingInfo();
        b2=b1;
        b1.BuildingInfo();
        b2.BuildingInfo();
    }
}
```

```
面积=1000.0;教室号=101;座位数=30
面积=800.0;教室号=102;座位数=40
面积=1000.0;教室号=101;座位数=30
面积=1000.0;教室号=101;座位数=30
```

# 4.2类与对象的概念

- 4.2.3类的设计
  - 由于封装性是面向对象的特征之一，因此通常将类设计成一个黑匣子，使用者只能通过类所提供的公共方法来实现对内部成员的操作和访问，不能看见方法的实现细节，也不能直接访问对象内部成员。类的封装可以隐藏类的实现细节，促使用户只能通过方法去访问数据，这样就可以增强程序的安全性。

```
class Building{

}
```

# 4.3方法

- 方法（method）是数行代码的集合，可以操作类中的属性，用于解决特定问题。在程序中多次使用相同的代码，重复地编写及维护比较麻烦，因此可以将此部分代码定义成一个方法，以供程序反复调用。

```
Building(String Floor,String RoomNO,int seetNumber){
    this.Floor=Floor;
    this.RoomNO=RoomNO;
    this.seetNumber = seetNumber;
}

public void BuildingInfo() {
    System.out.println("面积="+Area+";教室号="+Floor+RoomNO+";座位数="+seetNumber);
}
```

# 4.3方法

- 4.3.1成员方法
  - 1.方法的定义：Java中的方法定义在类中，一个类可以声明多个方法。方法包括方法头和方法体两部分，其中方法头确定方法的名字、形式参数的名字及类型、返回值的类型和访问权限等，方法体是具体完成的操作。
  - 修饰符 返回值类型 方法名([参数类型 参数名1, 参数类型 参数名2]){
  -  方法体
  - }

# 4.3方法

- 4.3.1成员方法
  - 2.方法的调用：方法在调用时执行方法中的代码，因此要执行方法，必须调用方法。如果方法有返回值，通常将方法调用作为一个值来处理。如果方法没有返回值，方法调用必须是一条语句。
  - 实参的值传递给方法的形参，称为值传递。

```
public void BuildInfo() {
    System.out.println("暂无建设用地信息！ ");
}
public void BuildInfo(double area) {
    System.out.println("本建设用地面积为:"+area+"m2");
}
public void BuildInfo(double area, int Floor) {
    System.out.println("本建设用地面积为:"+area+"m2,楼层为"+Floor+"层。 ");
}
```

```
暂无建设用地信息！
本建设用地面积为:100.0m2
本建设用地面积为:100.0m2,楼层为1层。
```

# 4.3方法

- 4.3.2构造方法
  - 构造方法必须以类名作为方法的名称，不返回任何值，也就是说构造方法是以类名为名称的特殊方法。

# 4.3方法

- 4.3.2构造方法
  - 在Java中，最少要有一个构造方法。
  - 类的构造方法可以显示定义也可以隐式定义，显示定义的意思是说在类中已经写好了构造方法的代码；隐式定义是指如果在一个类中没有定义构造方法，系统在解释时会分配一个默认的构造方法，这个构造工作方法只是一个空壳，没有参数，也没有方法体，类的所有属性系统将根据其数据类型默认赋值。
  - 类的构造方法是必须的，但其代码可以不编写。总之，如果在类中已经实现了构造方法，系统不会分配构造方法，如果没有实现，系统会自动分配。

# 4.3方法

- 4.3.2构造方法
- 构造方法的特点如下：
  - 构造方法必须与类同名。
  - 构造方法没有返回类型，也不能用void。
  - 构造方法不能由编程人员调用，是系统自动调用。
  - 一个类中可以定义多个构造方法，即构造方法的重载。但如果没有定义构造方法，系统会自动分配一个无参的默认的构造方法。

# 4.3方法

- 4.3.3方法的重载
- 1.成员方法的重载
  - 每一成员方法都有其签名，方法的签名包含方法的名称及它的形参的数量、每个形参的类型组成。
  - 方法签名不包含返回类型。在类中如果声明有多个同名的方法，但它们的签名不同，则称为方法的重载。

# 4.3方法

- 4.3.3方法的重载
- 2.构造方法的重载
  - 类的定义中如果有两个以上参数个数或类型不同的构造方法时，称为构造方法的重载。

```
public class Building{
    double area ;//面积，m2
    String Floor,RoomNO;//定义楼层，单层教室编号
    int seetNumber;//教室座位数
    Building() {
        System.out.println("暂无建设用地信息！ ");
    }
    Building(double area) {
        this.area = area;
        System.out.println("本建设用地面积为:"+area+"m2");
    }
    Building(double area, String Floor) {
        this.area = area;
        this.Floor = Floor;
        System.out.println("本建设用地面积为:"+area+"m2,楼层为"+Floor+"层。");
    }
    public static void main(String[] args) {
        Building b0 = new Building();
        Building b1 = new Building(100);
        Building b2 = new Building(100,"1");
    }
}
```

```
暂无建设用地信息！
本建设用地面积为:100.0m2
本建设用地面积为:100.0m2,楼层为1层。
```

# 4.4关键字this的使用

- 每个对象都有一个名为this的引用，它指向当前对象本身
- 使用this调用类中的属性，也就是类中的成员变量。
- 使用this调用成员方法
- 使用this调用其他构造方法：必须位于首行，且只能出现一次。

```
Building() {
    System.out.println("暂无建设用地信息！");
}
Building(double area) {
    this();
    this.area = area;
    System.out.println("本建设用地面积为:"+area+"m2");
}

public static void main(String[] args) {
    Building b0 = new Building();
    Building b1 = new Building(100);
}
```

```
暂无建设用地信息！
暂无建设用地信息！
本建设用地面积为:100.0m2
```

# 4.5关键字static的使用

- Java的类中可以包含两种成员：实例成员和静态成员。
  - 实例成员：一般在类中定义的成员是每个由此类产生的对象拥有的，因此可以称之为实例成员或对象成员
  - 静态成员：如果需要让类的所有对象在类的范围内共享某个成员，而这个成员不属于任何由此类产生的对象，它是属于整个类的，这种成员称为静态成员或类成员。

# 4.5关键字static的使用

- 4.5.1静态属性与实例属性
  - 静态属性：使用static修饰的属性，称为静态属性或类属性，它被类的所有对象共享，属于整个类所有，因此可以通过类名直接来访问。
  - 实例属性：而未使用static修饰的属性称为实例属性，它属于具体对象独有，每个对象分别包含一组该类的所有实例属性。

```
class Construct{
    int count;
    public Construct() {
        count++;
    }
}
public class Building{
    public static void main(String[] args) {
        Construct c1 = new Construct();
        Construct c2 = new Construct();
        Construct c3 = new Construct();
        System.out.println(c3.count);
    }
}
```

1

```
class Construct{
    static int count;
    public Construct() {
        count++;
    }
}
public class Building{
    public static void main(String[] args) {
        Construct c1 = new Construct();
        Construct c2 = new Construct();
        Construct c3 = new Construct();
        System.out.println(Construct.count);
    }
}
```

3

# 4.5关键字static的使用

- 4.5.2静态方法与实例方法
  - 静态方法：使用static修饰的成员方法，称为静态方法，无须创建类的实例就可以调用静态方法，静态方法可以通过类名调用。
  - 实例方法：没有用static修饰的方法，称为实例方法。

```
class Construct{
    static int count;
    public Construct() {
        count++;
    }
    public static void ConstructTime() {
        System.out.println("实例化次数为："+count);
    }
}
public class Building{
    public static void main(String[] args) {
        Construct c1 = new Construct();
        Construct c2 = new Construct();
        Construct c3 = new Construct();
        Construct.ConstructTime();
    }
}
```

```
实例化次数为：3
```

# 4.5关键字static的使用

- 4.5.3静态成员和实例成员的区别
- 1.静态成员的特征
- 2.实例成员的特征
- 3.访问权限

# 4.5关键字static的使用

- 1.静态成员的特征
  - （1）一个静态属性只标识一个存储位置。无论创建了多少个类的对象，永远都只有静态属性的一个副本。
  - （2）静态方法不在某个特定对象上操作，在这样的方法中引用this是错误的。

# 4.5关键字static的使用

- 2.实例成员的特征
  - （1）类的每个对象分别包含一组该类的所有实例属性。类的每个对象都为每个实例属性建立一个副本。也就是说类的每个对象的实例属性的存储位置都是不同的。
  - （2）实例方法在类的给定对象上操作，此对象可以作为this访问。

# 4.5关键字static的使用

- 3.访问权限
  - （1）静态方法可以访问静态成员，但是不能访问实例成员。
  - （2）实例成员可以访问静态成员，也可以访问实例成员。

# 4.5关键字static的使用

- 4.5.4代码块
  - 1.动态代码块：动态代码块就是直接定义在类中的代码块，它没有任何前缀、后缀及关键字修饰。
  - 2.静态代码块：静态代码块就是使用static关键字修饰的代码块，它是最早执行的代码块。

```
class Construct{
    static int count;
    public Construct() {
        count++;
    }
    public static void ConstructTime() {
        System.out.println("实例化次数为："+count);
    }
    static {
        System.out.println("实例化次数为："+count);
    }
}
public class Building{
    public static void main(String[] args) {
        Construct c1 = new Construct();
        Construct c2 = new Construct();
        Construct c3 = new Construct();
        Construct.ConstructTime();
    }
}
```

```
实例化次数为：0
实例化次数为：3
```

# 4.6内部类

- 在Java中，类中除了可以定义成员变量与成员方法外，还可以定义类，该类称作内部类，内部类所在的类称作外部类。
- 内部类的3点共性
- 根据内部类的位置、修饰符和定义的方式可分为成员内部类、静态内部类、方法内部类以及匿名内部类4种。

# 4.6内部类

- 4.6.1成员内部类
- 成员内部类是指类作为外部类的一个成员，能直接访问外部类的所有成员，但在外部类中访问内部类，则需要在外部类中创建内部类的对象，使用内部类的对象来访问内部类中的成员。同时，若要在外部类外要访问内部类，则需要通过外部类对象去创建内部类对象，在外部类外创建一个内部类对象的语法格式如下：
- 外部类名.内部类名 变量名=new 外部类名().内部类名()

# 4.6内部类

- 4.6.2静态内部类
- 如果不需要外部类对象与内部类对象之间有联系，那么可以将内部类声明为static，用static关键字修饰的内部类称为静态内部类。静态内部类可以有实例成员和静态成员，它可以直接访问外部类的静态成员，但如果想访问外部类的实例成员，就必须通过外部类的对象去访问。另外，如果在外部类外访问静态内部类成员，则不需要创建外部类对象，只需创建内部类对象即可。创建内部类对象的语法格式如下：
- 外部类名.内部类名 变量名=new 外部类名.内部类名()

# 4.6内部类

- 4.6.3方法内部类
- 方法内部类是指在成员方法中定义的类，它与局部变量类似，作用域为定义它的代码块，因此它只能在定义该内部类的方法内实例化，不可以在此方法外对其实例化。

# 4.6内部类

- 4.6.4匿名内部类
- 匿名内部类就是没有名称的内部类。创建匿名内部类时会立即创建一个该类的对象，该类定义立即消失，匿名内部类不能重复使用。

# 4.6内部类

```java
class Outer {
    private String outerField = "Outer Field"; // 外部类的私有字段
    // 成员内部类
    class Inner {
        void show() {// 内部类可以直接访问外部类的私有成员
            System.out.println("访问外部类字段: " + outerField);
        }
    }
}
public class Main {
    public static void main(String[] args) {
        Outer outer = new Outer(); // 首先创建外部类对象
        Outer.Inner inner = outer.new Inner(); // 通过外部类对象创建内部类对象
        inner.show(); // 输出：访问外部类字段: Outer Field
    }
}
```

```java
class Outer {
    private static String staticOuterField = "Static Outer Field"; // 外部类的静态字段
    // 静态内部类
    static class StaticInner {
        void show() {
            System.out.println("访问外部类静态字段: " + staticOuterField); //只能访问外部类的静态成员
            // System.out.println(outerField); // 错误！不能访问外部类的非静态成员
        }
    }
}
public class Main {
    public static void main(String[] args) {
        // 创建静态内部类对象无需先创建外部类对象
        Outer.StaticInner staticInner = new Outer.StaticInner();
        staticInner.show(); // 输出：访问外部类静态字段: Static Outer Field
    }
}
```

```java
class Outer {
    void outerMethod() {
        final String localVar = "Local Variable"; // 方法内的局部变量，必须是 final或有效 final
        // 方法内部类
        class MethodInner {
            void show() {
                System.out.println("访问方法局部变量: " + localVar); //可以访问所在方法的final局部变量
            }
        }
        MethodInner methodInner = new MethodInner(); // 只能在方法内部创建实例
        methodInner.show(); // 输出：访问方法局部变量: Local Variable
    }
}
public class Main {
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.outerMethod();
    }
}
```

```java
abstract class AnonymousInner{
    public abstract void show();
}
class Outer {
    void outerMethod() {
        AnonymousInner ano = new AnonymousInner() {
            public void show() {
                System.out.println("访问匿名内部类");
            }
        };
        ano.show();
    }
}
public class Main {
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.outerMethod();
    }
}
```

访问匿名内部类

# 4.7包

- 4.7.1包的定义和使用
  - 包（package）是Java提供的一种区别类的名字空间的机制，是类的组织方式，是一组相关类和接口的集合，它提供了访问权限和命名的管理机制。
  - 使用package语句声明包，其语法格式如下：
  - package 包名

# 4.7包

- 4.7.2 import语句
  - 当类进行打包操作后，同一个包内的类默认引入，当需要使用其他包中的类时，需要在程序的开头写上import语句，指出要导入哪个包的哪些类，然后才可以使用这些类。
  - 引入包的语法如下：import 包名.*;

# 4.8类及成员的访问权限

- Java为类中的成员设置了4种访问权限，为类本身设置了2种访问权限。

# 4.8类及成员的访问权限

- 4.8.1类的访问权限
  - Java提供了2中类的访问权限，分别是public和默认。
- 4.8.2类成员的访问权限
  - Java提供的4种成员的访问权限分别是：public（公有）、protected（保护）、默认和private（私有）

# 4.8类及成员的访问权限

- private（类访问权限）：被private修饰的成员，只能被当前类中其他成员访问，不能在类外被访问。
- default（包访问权限）：如果一个类或类的成员前没有任何访问权限修饰，则表示默认访问权限，即类或类的成员可以被同一包中的所有类访问。
- protected（子类访问权限）：被protected修饰的成员，可以被同一个包中的任何类或不同包中的子类访问。
- public（公共访问权限）：被public修饰的类或类的成员，可以被任何包中的类访问。

# 4.8类及成员的访问权限

| 访问权限 | private | default | protected | public |
| --- | --- | --- | --- | --- |
| 同一类中成员 | √ | √ | √ | √ |
| 同一包中其他类 | | √ | √ | √ |
| 不同包中子类 | | | √ | √ |
| 不同包的非子类 | | | | √ |