

讲义制作：海风
参考《Java疯狂讲义（第3版）》

第3章 面向对象编程 上

韩 慧

hanhuie@126.com

A solid green horizontal bar at the bottom of the slide.

本章导读

- 3.1 面向对象基础 OOP
- 3.2 类和对象 Class & Object
- 3.3 包 Package
- 3.4 访问权限 Access Privileges
- 3.5 综合作业及延伸

本章导读

✓ 3.1 面向对象基础 OOP

□ 3.2 类和对象 Class & Object

□ 3.3 包 Package

□ 3.4 访问权限 Access Privileges

□ 3.5 综合作业及延伸

3.1 面向对象编程基础

Object Oriented Programming, OOP

封装

- 将数据（属性）和对数据的操作（功能）封装在一起
- 成员变量，成员方法，类

继承

- 子类可以继承父类的属性和功能，同时又可以增加子类独有的属性和功能

多态

- 多个操作具有相同的名字，但是接受的消息类型必须不同
- 同一个操作被不同类型的对象调用时产生不同的行为

本章导读

- 3.1 面向对象基础 OOP
- ✓ 3.2 类和对象 Class & Object
- 3.3 包 Package
- 3.4 访问权限 Access Privileges
- 3.5 综合作业及延伸

3.2 类和对象

- 3.2.1 类的定义
- 3.2.2 对象的声明与创建
- 3.2.3 初窥构造器
- 3.2.4 深入成员变量
- 3.2.5 方法详解
- 3.2.6 初始化块
- 3.2.7 static小结

3.2 类和对象Class & Object

- 面向对象程序设计OOP中，有两个重要的概念：**类**（class）与**对象**（object，也被称为**实例**，instance），其中类是某一批对象的抽象，对象才是一个具体存在的实体。
- 类是由描述状态的属性（变量）和用来实现对象行为的方法组成。
 - 几种叫法：

◦ 属性 property — 域 field — 成员变量	状态
◦ 方法 method — 成员方法	行为

3.2.1 类的定义_自定义Java类示例

```
[修饰符] class 类名{  
    // 零个到多个成员变量  
    // 零个到多个初始化块  
    // 零个到多个构造器  
    // 零个到多个成员方法  
    // 零个到多个内部类  
  
    ...  
}
```

```
/**  
 * 自定义Java类的示例  
 */  
class MyClass {  
    // 公有字段  
    public String Information = "";  
  
    // 自定义公有Java实例方法  
    public void myMethod(String argu) {  
        System.out.println(argu);  
    }  
  
    // 定义属性：私有字段+get方法+set方法  
    private int value;  
    public int getValue() {  
        return value;  
    }  
    public void setValue(int value) {  
        this.value = value;  
    }  
}
```

ClassAndObjectTest

3.2.1 类的定义_类的成员类型

一个类内部的成员类型分为以下五种：

1. 构造器 Constructor
2. 成员变量 Property / Field
3. 成员方法 Method
4. 初始化块 Initialization Block
5. 内部类 Inner Class（详见 第4章讲义）

3.2.1 类的定义_使用自定义类

```
public class ClassAndObjectTest {  
  
    public static void main(String[] args) {  
        //创建类的实例，定义一个对象变量引用这一实例  
        MyClass obj = new MyClass();  
        //通过对象变量调用类的公有方法  
        obj.myMethod("Hello");  
        //给属性赋值  
        obj.setValue(100);  
        //输出属性的当前值  
        System.out.println(obj.getValue());  
        //直接访问对象公有字段  
        obj.Information = "Information";  
        //输出对象公有字段的当前值  
        System.out.println(obj.Information);  
    }  
}
```

ClassAndObjectTest

3.2.2 对象的声明与创建

① 对象的声明

类的名字 对象名字;

```
Rect rectangleOne; //此时 rectangleOne = null
```

- rectangleOne是一个空对象，它不能访问成员变量和成员方法

② 创建对象

对象名=new 构造器名 (参数列表);

也可以把声明对象和创建对象合在一起进行

```
rectangleOne=new Rect( );
```

```
rectangleOne=new Rect(10, 20);
```

```
Rect rectangleOne=new Rect(10,20);
```

3.2.3 初窥构造器 Constructor

- 构造器 Constructor，也称为构造方法，但二者实际有较大的差异，因此**推荐使用构造器这一叫法**，以与普通方法作区分。
- 类创建对象时，需要使用构造器完成对象的初始化工作。
- 构造器的名称必须与类名相同。
- 构造器声明不写返回值类型，与方法的重大不同！
- 一个类中可以有若干个构造器（名称相同），但是构造器的参数必须不同（构造器重载，我们将在this关键字部分详细讨论构造器的重载）
- 如果类中没有构造器，系统为类定义一个默认的构造器，该构造器没有参数，类体为空，访问权限与类的访问权限相同。

3.2.3 初窥构造器_构造器重载

```
1  class Rect {
2      double sideA, sideB;
3      Rect() {} //无参构造器
4      Rect(double a, double b) { //有参构造器
5          sideA=a;
6          sideB=b;
7      }
8
9      //...
10 }
```

一个类中可以有若干个构造器（名称相同），但是构造器的参数必须不同，这称之为构造器重载

3.2.3 初窥构造器

再看自定义类——默认无参构造器

```
/**
 * 自定义Java类的示例
 */
class MyClass {
    // 公有字段
    public String Information = "";

    // 自定义公有Java实例方法
    public void myMethod(String argu) {
        System.out.println(argu);
    }

    // 定义属性：私有字段+get方法+set方法
    private int value;
    public int getValue() {
        return value;
    }
    public void setValue(int value) {
        this.value = value;
    }
}
```

如果类中没有构造器，系统为类定义一个默认的构造器，该构造器没有参数，类体为空，访问权限与类的访问权限相同。

3.2.3 初窥构造器

- 以下代码为何无法通过编译？哪儿出错了？

```
public class Test {  
  
    public static void main(String[] args) {  
        Foo obj1=new Foo();  
    }  
}  
  
class Foo{  
    int value;  
    public Foo(int initValue) {  
        value=initValue;  
    }  
}
```

如果类定义中显式地给出了构造器，系统就不会再提供默认的空参构造器

3.2.4 深入成员变量_Property / Field

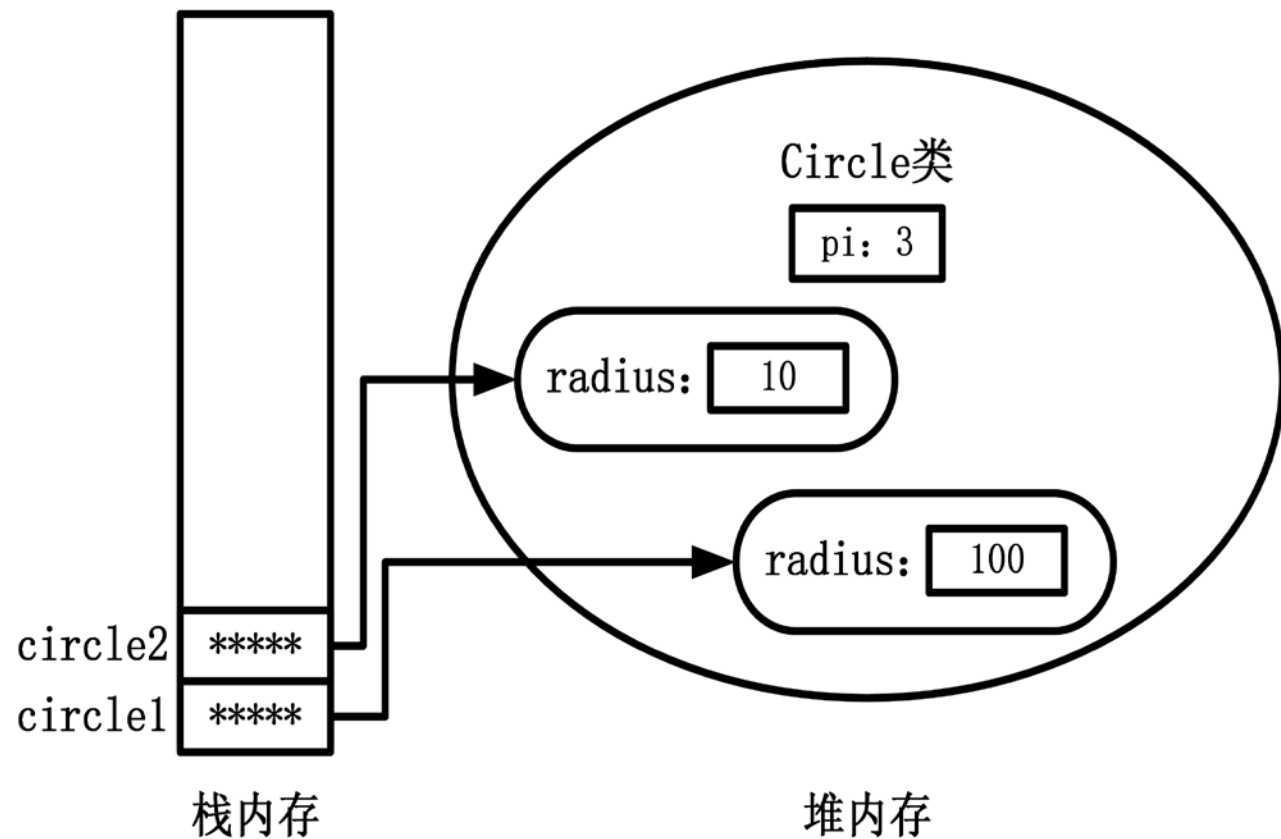
- 成员变量分为两种
 - 实例变量
 - 不用关键字static修饰
 - 一个类中不同对象的实例变量将被分配不同的存储空间
 - 只能通过对象访问实例变量
 - 类变量（静态变量）
 - 用关键字static修饰
 - 一个类中所有对象的某个类变量被分配同一个内存，所有对象共享这个类变量
 - 可以通过类名访问类变量，也可以通过某个对象访问类变量

3.2.4 深入成员变量

类变量与实例变量的内存分配比较

```
class Circle {  
    static double pi; //类变量  
    double radius; //实例变量  
}  
  
public class Test{  
    public static void main(String[] args) {  
        Circle.pi=3;  
        Circle circle1=new Circle();  
        circle1.radius=10;  
        Circle circle2=new Circle();  
        circle2.radius=100;  
    }  
}
```

Property



3.2.4 深入成员变量

final成员变量

- final成员变量
 - 如果一个成员变量修饰为final，就是**常量**。（类似C语言的const）
 - **常量的名字习惯用大写字母。**
 - final int **MAX**=100;
- final修饰的成员变量不占用内存，**声明时必须初始化，本质类似宏变量。**
- 如果final变量是基本数据类型，则其值不能发生变化。
- 如果final变量是引用数据类型，则其指向对象的引用不能发生变化。

3.2.4 深入成员变量

final成员变量

- final修饰的成员变量**必须由程序员显式地指定初始值。**
- 类变量：必须在类初始化块或声明该类变量时指定初始值。（2选1）
- 实例变量：必须在非静态初始化块、声明该实例变量时或构造器中指定初始值。（3选1）

3.2.4 深入成员变量

思考

- Tom类的两个实例，cat1和cat2在内存中成员变量的内存如何分配。

```
class Tom{
    final int MAX=100; //实例final成员变量
    static final int MIN=20; //静态final成员变量
}
public class Test{
    public static void main(String args[ ])
    {
        System.out.println(Tom.MIN);
        Tom cat1 = new Tom();
        Tom cat2 = new Tom();
        int x = Tom.MIN + cat1.MAX + cat2.MAX;
        System.out.println(x);
    }
}
```

3.2.4 深入成员变量

加深理解引用数据类型常量

- 如果final变量是引用数据类型，则其指向对象的引用不能发生变化。
- 思考，下图所示的测试代码，能否成功编译，若可以，请猜测运行结果；若不可以，请说出原因。

```
class IdCard{
    final String cardNO;
    String policeStation;
    IdCard(String cardNO, String policeStation){
        this.cardNO = cardNO;
        this.policeStation = policeStation;
    }
}

class Person{
    final IdCard idCard;
    Person(IdCard idCard){
        this.idCard = idCard;
    }
}
```

```
public class Test{
    public static void main(String[] args){
        IdCard idCard = new IdCard("320924*****2145", "****派出所");
        Person stu = new Person(idCard);
        System.out.println(stu.idCard);
        System.out.println(stu.idCard.policeStation);
        stu.idCard.policeStation = "北京市海淀区东升派出所";
        System.out.println(stu.idCard);
        System.out.println(stu.idCard.policeStation);
    }
}
```

FinalReferenceTest

3.2.5 方法详解_Method

- 定义方法的语法格式

```
[修饰符] 方法返回值类型 方法名 (形参列表) {  
    // 由零条到多条可执行性语句组成的方法体  
    ...  
}
```

- 实例方法和类方法

- 实例方法：方法声明中不用static修饰，必须通过对象来调用
- 类方法（静态方法）：方法声明中用static修饰，可以通过类名和对象来调用

- 由于类方法属于类，因此强烈推荐使用类名调用类方法

3.2.5 方法详解

类方法与成员方法

访问方法	类变量或方法	实例变量或方法
类方法	✓	✗
实例方法	✓	✓

```
class A {  
    float a,b;  
    void sum(float x,float y) { //实例方法  
        a=max(x,y); //实例方法调用类方法  
        b=min(x,y); //实例方法调用实例方法  
    }  
    static float getMaxSqrt(float x,float y) { //类方法  
        return max(x,y)*max(x,y); //类方法调用类方法  
    }  
    static float max(float x,float y) { //类方法  
        return x>y?x:y;  
    }  
    float min(float x,float y) { //实例方法  
        return x<y?:y;  
    }  
}
```

3.2.5 方法详解_参数传递

- 问题1: C语言的函数, 参数传递有哪些机制? 请简述。
- 问题2: Java语言的类型系统主要分为哪两类? 请简述。

3.2.5 方法详解_参数传递

- 问题1 C语言的函数，参数传递有哪些机制？请简述。
 - **值传递**，向函数传递的是实参的拷贝副本，函数中对形参的修改，不会影响实参。
 - **引用传递（指针传递）**，向函数传递的是实参的地址，函数中对形参指向的内存进行修改，会影响实参的值。
- 问题2 Java语言的类型系统主要分为哪两类？请简述。
 - **基本数据类型**（char, byte, short, int, long, float, double, boolean）
 - **引用数据类型**（class, interface, array, enum, @interface）

3.2.5 方法详解_参数传递

- 虽然Java有引用数据类型，但与C语言不一样的是，**Java的参数传递机制只有一种，那就是值传递！！**
- 基本数据类型参数的值传递
 - 实参和形参占有不同的内存空间，形参的改变不影响实参
- 引用数据类型参数的值传递
 - 引用类型数据包括对象、数组以及后面将要学习的接口和泛型等
 - 引用数据类型实参和形参在传递过程中，传递的也是副本，分别占有不同的内存，都指向共同的内存
 - 实参和形参指向共同的内存，改变形参引用的实体，会导致实参引用的实体发生同样的变化

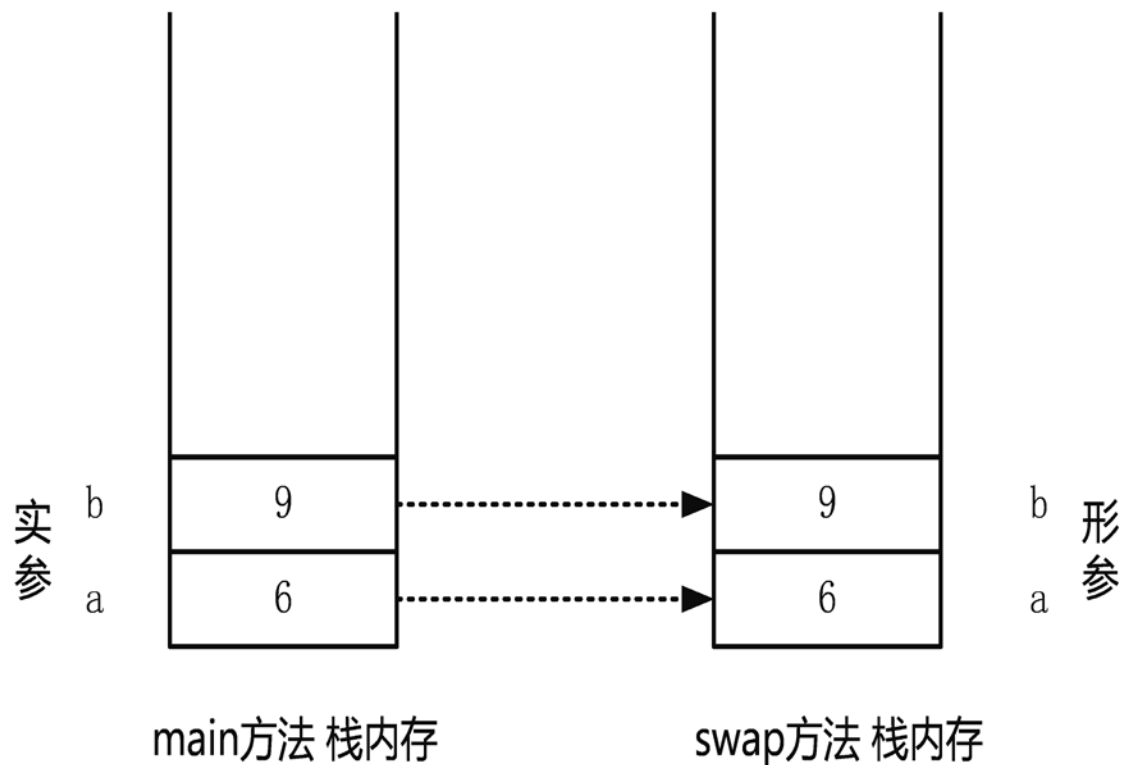
3.2.5 方法详解

基本数据类型的值传递

```
class PrimitiveTransferTest {  
    public static void swap(int a , int b) {  
        // 下面三行代码实现a、b变量的值交换。  
        int tmp = a; // 定义一个临时变量来保存a变量的值  
        a = b; // 把b的值赋给a  
        b = tmp; // 把临时变量tmp的值赋给a  
        System.out.println("swap方法里，a的值是"  
            + a + "；b的值是" + b);  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        int a = 6, b = 9;  
        PrimitiveTransferTest.swap(a , b);  
        System.out.println("交换结束后，变量a的值是"  
            + a + "；变量b的值是" + b);  
    }  
}
```

PrimitiveTransferTest

基本数据类型实参和形参占有不同的内存空间，形参的改变不影响实参。



3.2.5 方法详解

引用数据类型的值传递 (1)

```
class DataWrap {
    int a;
    int b;
}

class ReferenceTransferTest {
    public static void swap(DataWrap dw) {
        // 下面三行代码实现dw的a、b两个成员变量的值交换。
        int tmp = dw.a; // 定义一个临时变量来保存dw对象的a成员变量的值
        dw.a = dw.b; // 把dw对象的b成员变量值赋给a成员变量
        dw.b = tmp; // 把临时变量tmp的值赋给dw对象的b成员变量
        System.out.println("swap方法里，a成员变量的值是"
            + dw.a + "；b成员变量的值是" + dw.b);
    }
}

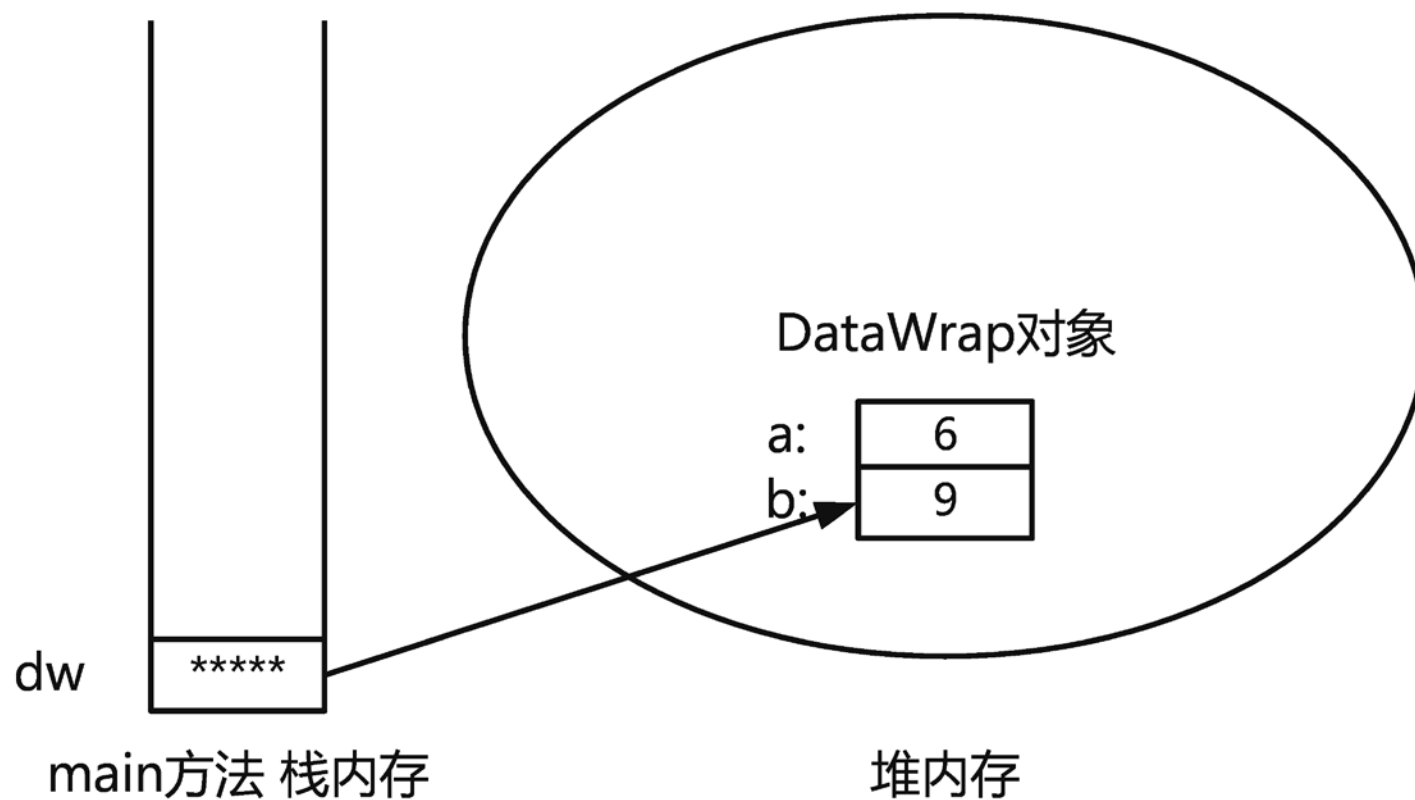
public class Test {
    public static void main(String[] args) {
        DataWrap dw = new DataWrap();
        dw.a = 6;
        dw.b = 9;
        ReferenceTransferTest.swap(dw);
        System.out.println("交换结束后，a成员变量的值是"
            + dw.a + "；b成员变量的值是" + dw.b);
    }
}
```

引用数据类型的实参和形参指向共同的内存，改变形参引用的实体，会导致实参引用的实体发生同样的变化。

3.2.5 方法详解

引用数据类型的值传递（1）——内存详解

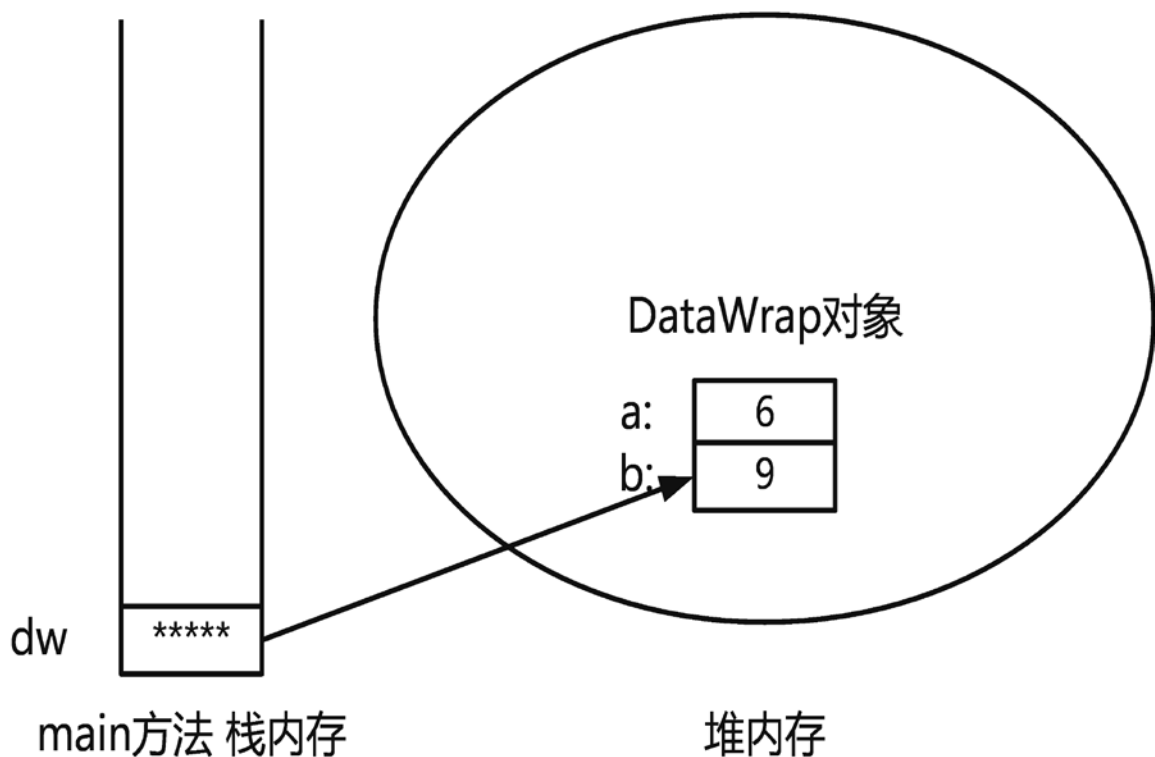
1. main()方法中创建了DataWrap对象后的存储示意图



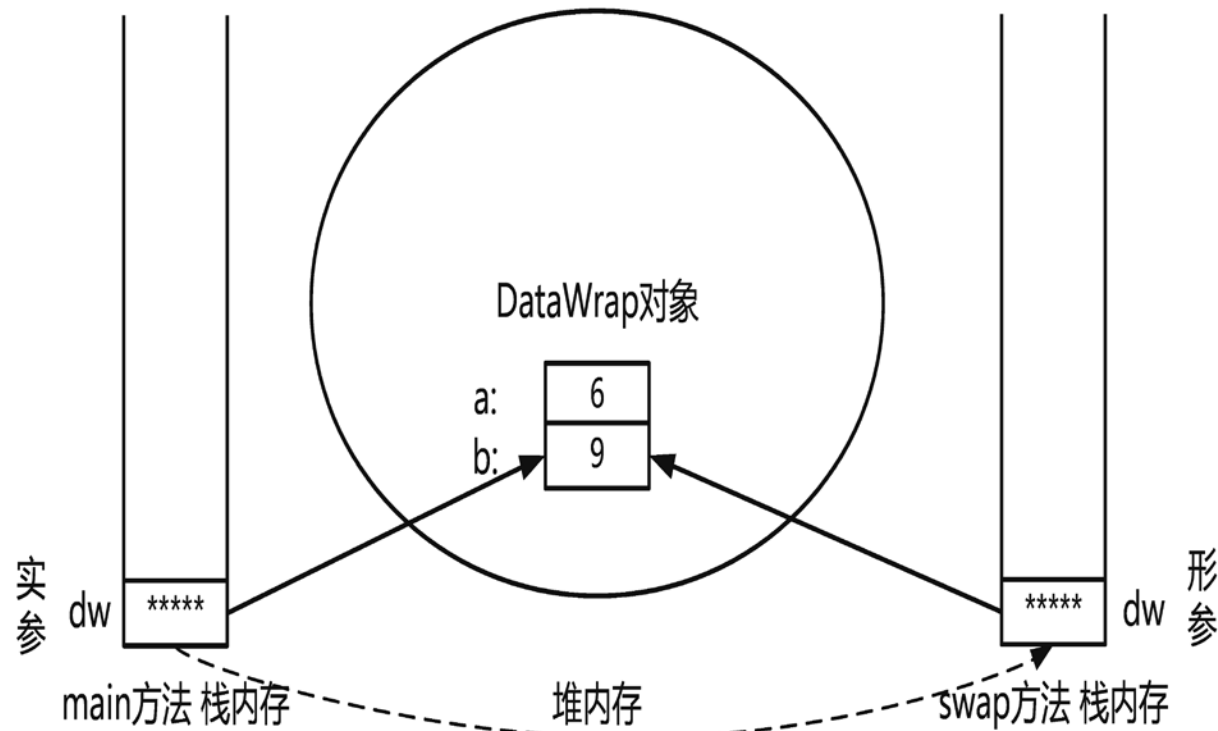
3.2.5 方法详解

引用数据类型的值传递 (1) ——内存详解

1. main()方法中创建了DataWrap对象后的存储示意图



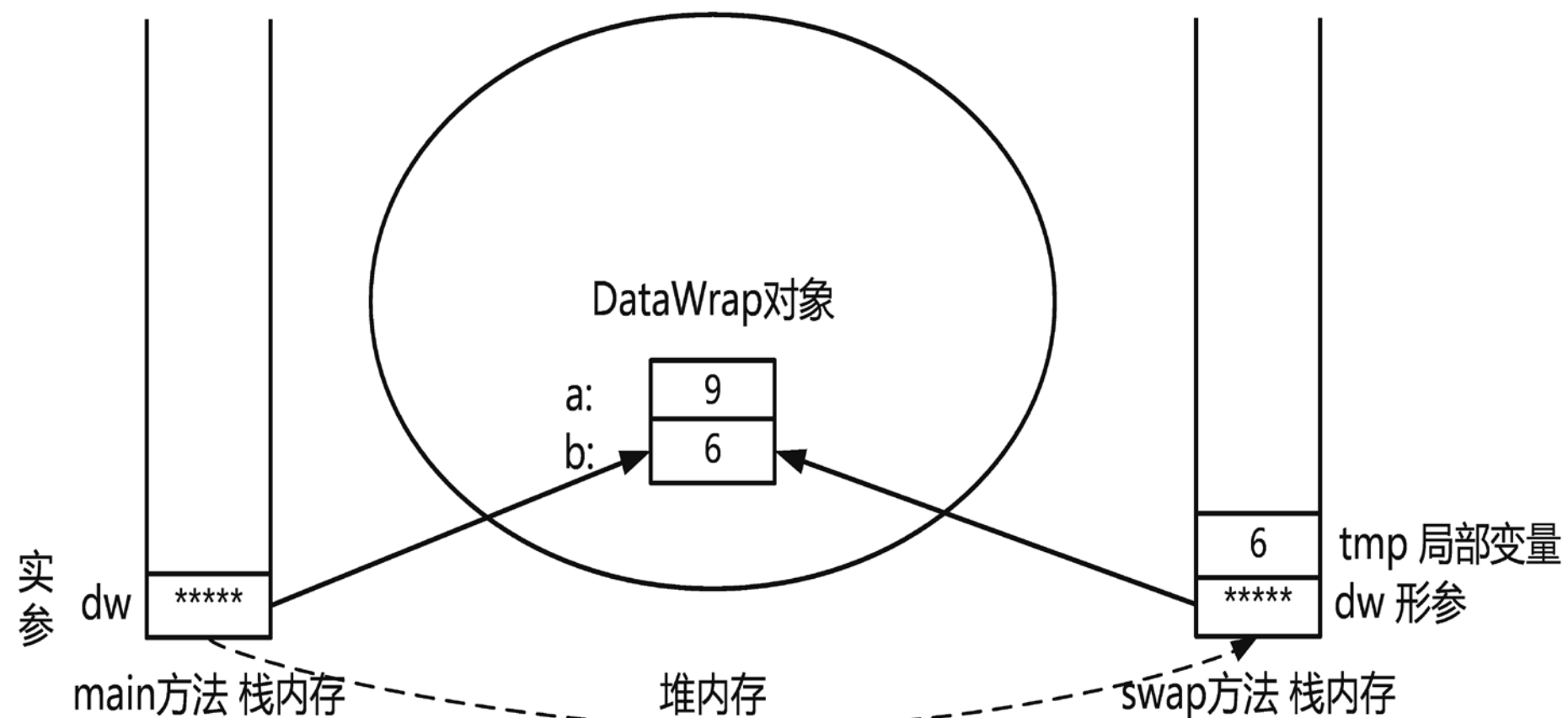
2. main()方法中的dw对象传入swa()方法后内存示意图



3.2.5 方法详解

引用数据类型的值传递（1）——内存详解

3. swap()方法执行完毕后，退出swap()方法前的内存示意图



引用数据类型的实参和形参指向共同的内存，改变形参引用的实体，会导致实参引用的实体发生同样的变化。

3.2.5 方法详解

思考

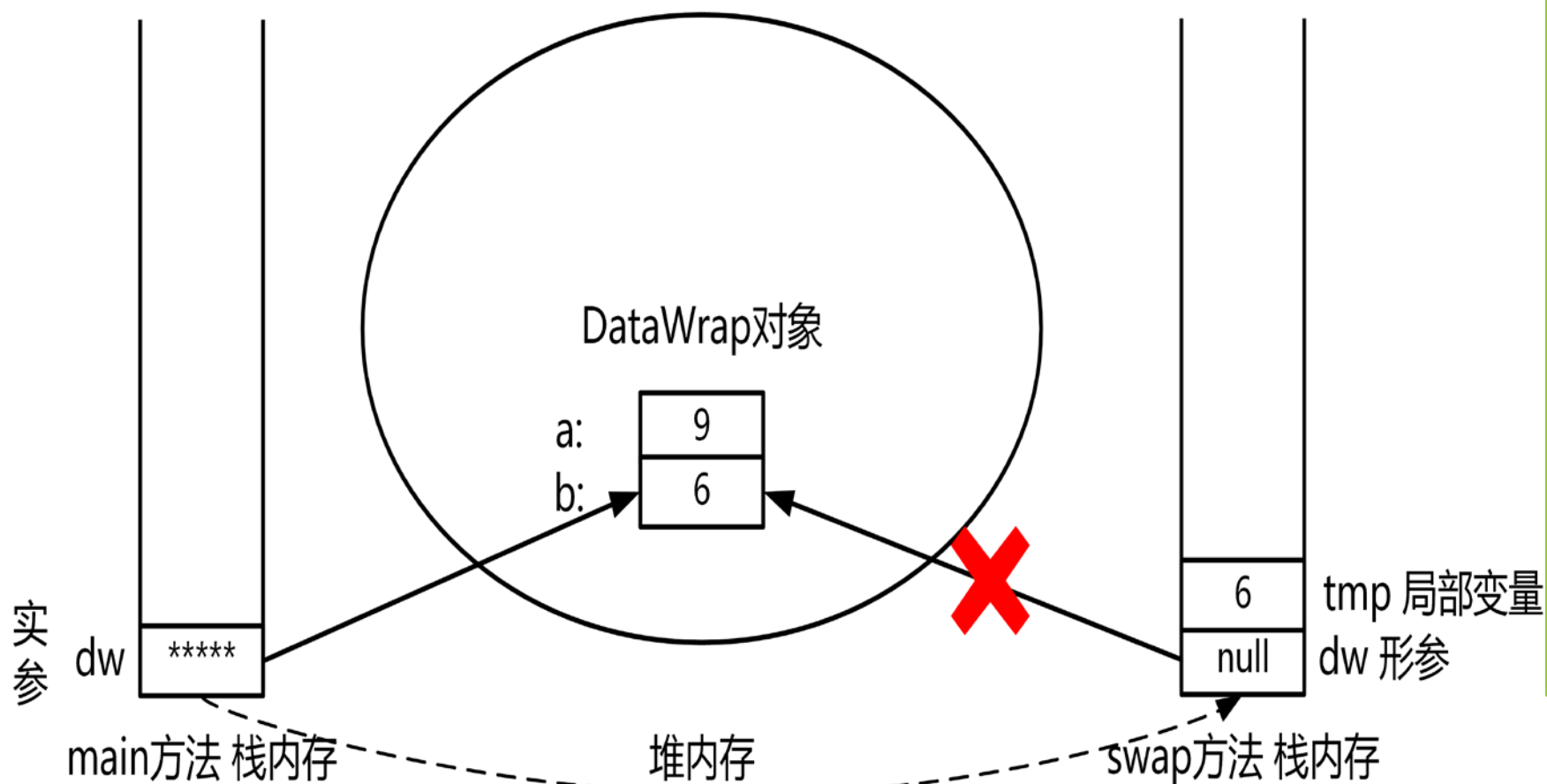
- 如右图所示的测试代码，在swap()方法的最后，添加了一行，将形参dw赋为了null，使它不再指向任何有效地址。
- 此时运行main()方法，会得到怎样的输出？

```
class DataWrap {  
    int a;  
    int b;  
}  
  
class ReferenceTransferTest {  
    public static void swap(DataWrap dw) {  
        // 下面三行代码实现dw的a、b两个成员变量的值交换。  
        int tmp = dw.a; // 定义一个临时变量来保存dw对象的a成员变量的值  
        dw.a = dw.b; // 把dw对象的b成员变量值赋给a成员变量  
        dw.b = tmp; // 把临时变量tmp的值赋给dw对象的b成员变量  
        System.out.println("swap方法里，a成员变量的值是"  
            + dw.a + "，b成员变量的值是" + dw.b);  
        dw = null; // 把dw直接赋为null，让它不再指向任何有效地址。  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        DataWrap dw = new DataWrap();  
        dw.a = 6;  
        dw.b = 9;  
        ReferenceTransferTest.swap(dw);  
        System.out.println("交换结束后，a成员变量的值是"  
            + dw.a + "，b成员变量的值是" + dw.b);  
    }  
}
```


3.2.5 方法详解

引用数据类型的值传递（2）——内存详解

将swap()方法中的dw赋值为null后的内存示意图



引用数据类型实参和形参在传递过程中，传递的也是副本，分别占有不同的内存，都指向共同的内存。让形参指向新的地址，不会改变实参所指向的地址。

3.2.5 方法详解

形参个数可变的方法

- 从JDK1.5之后，Java允许定义形参个数可变的参数，从而允许为方法指定数量不确定的形参。如果在定义方法时，在最后一个形参的类型后增加三点（...），则表明该形参可以接受多个参数值，**多个参数值被当成数组传入**。
- 长度可变的形参只能处于形参列表的最后，一个方法中最多只能包含一个长度可变的形参。
- **长度可变的形参本质就是一个数组类型的形参**，因此调用包含一个长度可变形参的方法时，这个长度可变的形参既可以传入多个参数，也可以传入一个数组。

3.2.5 方法详解

形参个数可变的方法——代码示例

```
class Varargs {  
    // 定义了形参个数可变的方法  
    public static void test(int a , String... books) {  
        // books被当成数组处理  
        for (String tmp : books) {  
            System.out.println(tmp);  
        }  
        // 输出整数变量a的值  
        System.out.println(a);  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        Varargs.test(5 , "疯狂Java讲义" , "轻量级Java EE企业应用实战");  
    }  
}
```

VarArgsMethod

3.2.5 方法详解

形参个数可变的方法——代码示例

```
class Varargs {  
    // 定义了形参个数可变的方法  
    public static void test(int a, String... books) {  
        // books被当成数组处理  
        for (String tmp : books) {  
            System.out.println(tmp);  
        }  
        // 输出整数变量a的值  
        System.out.println(a);  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        Varargs.test(5, new String[] {"疯狂Java讲义", "轻量级Java EE企业应用实战"});  
    }  
}  
  
public {  
    public static void main(String[] args) {  
        Varargs.test(5, "疯狂Java讲义", "轻量级Java EE企业应用实战");  
    }  
}
```

长度可变的形参本质就是一个数组类型的形参，因此调用包含一个长度可变形参的方法时，这个长度可变的形参既可以传入多个参数，也可以传入一个数组。

3.2.5 方法详解

this关键字

- this关键字总是**指向调用该方法的对象**，根据this出现的位置不同，this作为对象的默认引用有两种情形。
 - 构造器中引用该构造器正在初始化的对象
 - 在方法中引用调用该方法的对象
- this可以出现在实例方法和构造器中，但是不可以出现在类方法中。（直观理解，类方法属于类，而this对象代表调用方法的对象，因此this不能出现在类方法中）
- 技巧：使用this区分成员变量和局部变量（**在这里，局部变量是指在构造器或成员方法的参数列表或者在方法体中定义的变量**）

3.2.5 方法详解

this关键字

- 在构造器中使用this：代表使用该构造器所创建的对象，this可以省略

```
public class Tom{
    int leg;
    Tom(int n) { //构造器
        leg=n;
        this.cry(); //this可以省略，等价于“cry();”
    }
    void cry() {
        System.out.println("Tom有"+leg+"条腿");
    }
    public static void main(String args[ ]) {
        Tom cat=new Tom(4); //调用构造方法Tom时，this就是对象cat
        cat.cry()
    }
}
```

3.2.5 方法详解

this关键字

- 在实例方法中使用this：代表使用该方法的当前对象，this可以省略

```
class Circle {  
    double pi=3.1416;  
    double radius;  
    double computeDiameter() {  
        return 2*this.radius; // this.成员变量  
    }  
    double computeGirth() {  
        double girth=this.pi*this.computeDiameter(); //this.成员方法  
    }  
}
```

3.2.5 方法详解

this关键字

- 使用this区分成员变量和局部变量
 - 如果成员变量和局部变量名称相同，则成员变量被隐藏，即成员变量在方法内暂时失效
 - 如果想在方法中使用成员变量，对于实例方法，应使用“this.成员变量”，对于类方法，应使用“类名.成员变量”
 - this不能省略

```
public class Tom{  
    int leg;  
    Tom(int leg) { //构造器  
        this.leg = leg;  
    }  
  
    // ...  
}
```


3.2.5 方法详解

this关键字

- 使用this区分成员变量和局部变量

```
class Tom
{
    int x=188,y; //成员变量
    void f()
    {
        int x=3; //局部变量
        y=x; //y的值是3，不是188
    }
}
```

```
class Tom
{
    int x=188,y;
    void f()
    {
        int x=3;
        y=this.x; //y的值是188
    }
}
```

3.2.5 方法详解

this关键字——二窥构造器之构造器重载

- 同一个类里具有多个构造器，多个构造器的形参列表不同，即被称为构造器重载。构造器重载允许Java类里包含多个初始化逻辑，从而允许使用不同的构造器来初始化Java对象。

```
class ConstructorOverload{
    public String name;
    public int count;
    // 提供无参数的构造器
    public ConstructorOverload(){
        // 提供带两个参数的构造器，
        // 对该构造器返回的对象执行初始化
        public ConstructorOverload(String name , int count){
            this.name = name;
            this.count = count;
        }
    }
}

public class Test {
    public static void main(String[] args) {
        // 通过无参数构造器创建ConstructorOverload对象
        ConstructorOverload oc1 = new ConstructorOverload();
        // 通过有参数构造器创建ConstructorOverload对象
        ConstructorOverload oc2 = new ConstructorOverload(
            "轻量级Java EE企业应用实战", 300000);
        System.out.println(oc1.name + " " + oc1.count);
        System.out.println(oc2.name + " " + oc2.count);
    }
}
```

ConstructorOverload

3.2.5 方法详解

this关键字——二窥构造器之构造器重载

- 如果一个类中包含了多个构造器，其中一个构造器完全包含另一个构造器的执行体，可以在构造器A中使用this关键字来调用构造器B。
- 系统会根据this后括号里的实参来调用形参列表与之对应的构造器。

```
public class Apple {  
    public String name;  
    public String color;  
    public double weight;  
    public Apple(){}  
    // 两个参数的构造器  
    public Apple(String name , String color) {  
        this.name = name;  
        this.color = color;  
    }  
    // 三个参数的构造器  
    public Apple(String name , String color , double weight) {  
        // 通过this调用另一个重载的构造器的初始化代码  
        this(name , color);  
        // 下面this引用该构造器正在初始化的Java对象  
        this.weight = weight;  
    }  
}
```

3.2.5 方法详解

方法重载 overload

- 方法重载是**多态**性的一种，是指一个类中可以有多具有**相同的名字**，但是，**参数数量或者参数类型不同**的方法。
- 方法的其他部分，如方法的返回值类型、修饰符和参数的名字不参与比较，与方法重载没有任何关系。
- 构造器重载属于方法重载的一种。

3.2.5 方法详解

方法重载——代码示例

```
class Overload {  
    // 下面定义了两个test()方法，但方法的形参列表不同  
    // 系统可以区分这两个方法，这种被称为方法重载  
    public void test() {  
        System.out.println("无参数");  
    }  
    public void test(String msg) {  
        System.out.println("重载的test方法 " + msg);  
    }  
}  
  
public class Test {  
    public static void main(String[] args){  
        Overload ol = new Overload();  
        // 调用test()时没有传入参数，因此系统调用上面没有参数的test()方法。  
        ol.test();  
        // 调用test()时传入了一个字符串参数，  
        // 因此系统调用上面带一个字符串参数的test()方法。  
        ol.test("hello");  
    }  
}
```

OverloadTest

3.2.5 方法详解

思考——再探“参数个数可变的方法”

- 如右图所示的参数个数可变的方法的方法重载的代码示例中，请思考1~4条test()方法依次会调用哪个test()方法。

```
class OverloadVarargs {  
    public void test(String msg) {  
        System.out.println("只有一个字符串参数的test方法");  
    }  
    // 因为前面已经有了一个test()方法，test()方法里有一个字符串参数。  
    // 此处的长度可变形参里不包含一个字符串参数的形式  
    public void test(String... books) {  
        System.out.println("****形参长度可变的test方法****");  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        OverloadVarargs olv = new OverloadVarargs();  
        olv.test();  
        olv.test("aa");  
        olv.test(new String[]{"aa"});  
        olv.test("aa", "bb");  
    }  
}
```

OverloadVarargs

3.2.6 初始化块 Initialization Block

- 类的初始化块不接受任何的参数，而且只要一创建类的对象，他们就会被执行。因此，适合于封装那些“**对象创建时必须执行的代码**”。
- 初始化块的语法格式如下：

```
[修饰符]{  
    // 初始化块的可执行代码  
    ...  
}
```

初始化块的修饰符只能是static，使用static修饰的初始化块被称为类初始化块。

- 初始化块中可以包含任何可执行性语句，包括定义局部变量，调用其他对象的方法等。

3.2.6 初始化块 代码示例

- 请猜测一下，如右图所示的示例代码的运行结果。
- 根据运行结果，你可以得到哪些结论？

```
class Person {  
    // 下面定义一个初始化块  
    {  
        int a = 6;  
        if (a > 4)  
        {  
            System.out.println("Person初始化块：局部变量a的值大于4");  
        }  
        System.out.println("Person的初始化块");  
    }  
    // 定义第二个初始化块  
    {  
        System.out.println("Person的第二个初始化块");  
    }  
    public Person() {    // 定义无参数的构造器  
        System.out.println("Person类的无参数构造器");  
    }  
}  
public class Test {  
    public static void main(String[] args) {  
        new Person();  
    }  
}
```

InitializationBlock

3.2.6 初始化块

代码示例

```
class Person {  
    // 下面定义一个初始化块  
    {  
        int a = 6;  
        if (a > 4)  
        {  
            System.out.println("Person初始化块: 局部变量a的值大于4");  
        }  
        System.out.println("Person的初始化块");  
    }  
    // 定义第二个初始化块  
    {  
        System.out.println("Person的第二个初始化块");  
    }  
    public Person() {    // 定义无参数的构造器  
        System.out.println("Person类的无参数构造器");  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        new Person();  
    }  
}
```

结论1：初始化块是Java类的一种成员，但它没有名字，也就无法使用类名或对象来调用初始化块。普通初始化块只在创建Java对象时隐式执行，而且**在构造器之前执行**。

结论2：一个类可以有多个初始化块，相同类型的初始化块之间有顺序：前面定义的初始化块先执行，后面定义的初始化块后执行。

3.2.6 初始化块

初始化顺序 (1)

- 请猜测右图所示的程序的运行结果。
- 根据运行结果你又可以得到怎样的结论？

```
class InstanceInitTest1{
{
    a = 6;
}
int a = 9;
}
class InstanceInitTest2{
int a = 9;
{
    a = 6;
}
}
public class Test {
    public static void main(String[] args) {
        System.out.println(new InstanceInitTest1().a);
        System.out.println(new InstanceInitTest2().a);
    }
}
```

InstanceInitTest

3.2.6 初始化块

初始化顺序 (1)

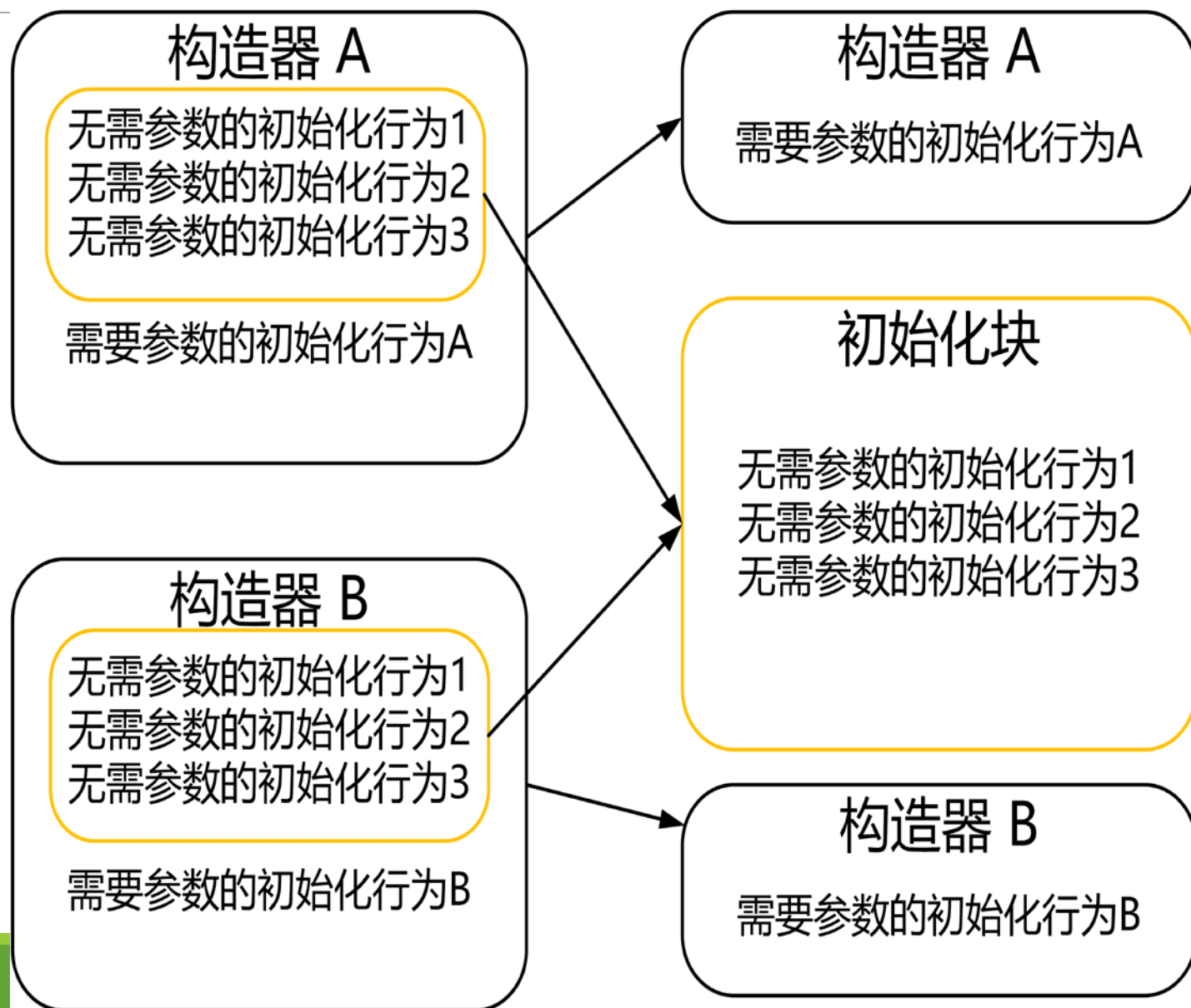
```
class InstanceInitTest1{
{
    a = 6;
}
int a = 9;
}
class InstanceInitTest2{
int a = 9;
{
    a = 6;
}
}
public class Test {
    public static void main(String[] args) {
        System.out.println(new InstanceInitTest1().a);
        System.out.println(new InstanceInitTest2().a);
    }
}
```

普通初始化块、声明实例变量指定的默认值都可认为是对象的初始化代码，它们的执行顺序与源程序中的排列顺序相同。

3.2.6 初始化块

普通初始化块的基本用法

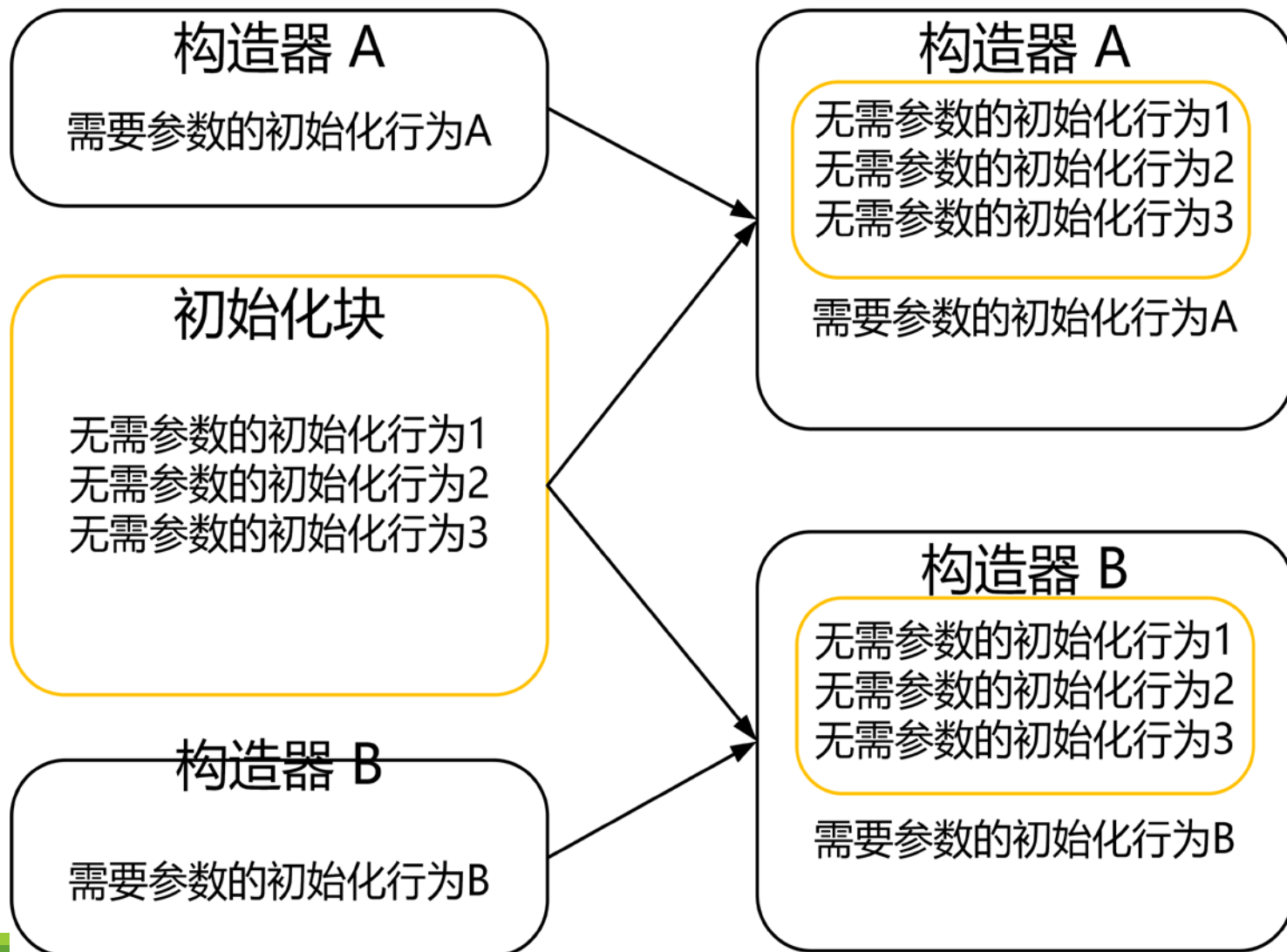
- 从某种程度上来看，初始化块是构造器的补充，初始化块总是在构造器执行之前执行。
- 与构造器不同的是，初始化块是一段固定的代码，不能接收任何参数。因此初始化块对同一个类的所有对象所进行的初始化处理完全相同。如果一个类的多个构造器重载过程中有一段初始化处理代码对所有对象完全相同，且无需接收任何参数，就可以把这段代码提取到初始化块中，使得代码更加优雅简洁。



3.2.6 初始化块

普通初始化块本质

●实际上初始化块是一个假象，使用javac命令编译java类后，该Java类中的初始化块“消失”——初始化块中的代码会被“还原”到每个构造器中，且位于构造器所有代码的前面。



3.2.6 初始化块

类初始化块

- 类初始化块，也被称为静态初始化块，普通初始化块负责对对象执行初始化，类初始化块则负责对类进行初始化。静态初始化块是类相关的，系统将在类初始化阶段执行类初始化块，而不是在创建对象时才执行。因此类初始化块总是比普通初始化块先执行。
- 类初始化块，属于类的静态成员，同样需要遵循静态成员不能访问非静态成员的规则，因此静态初始化块不能访问非静态成员，包括不能访问实例变量和实例方法。
- 类初始化块在诸如Spring这样的流行java web开发框架中，有大量应用，负责在类的初始化阶段，初始化并加载框架。
- 在第4章，我们将结合继承，再探初始化顺序。

3.2.6 初始化块

类初始化块——代码示例

- 当JVM第一次主动使用某个类时，系统会在类的准备阶段为该类的所有类成员变量分配内存；在初始化阶段则负责初始化这些类成员变量，初始化类成员变量就是执行类初始化块或者声明类成员变量时指定的初始值，它们的执行顺序与源代码中的排列顺序相同。

```
class StaticInitTest1 {  
    static {  
        a = 6;  
    }  
    static int a = 9;  
}  
class StaticInitTest2 {  
    static int a = 9;  
    static {  
        a = 6;  
    }  
}  
public class Test {  
    public static void main(String[] args) {  
        System.out.println(StaticInitTest1.a);  
        System.out.println(StaticInitTest2.a);  
    }  
}
```

StaticInitTest

3.2.7 static小结

- 虽然绝大部分资料都喜欢把static称为静态，但实际上这种说法很模糊，无法直观说明static的真正作用。**static的真正作用是区分成员变量、方法、内部类、初始化块这4中成员到底属于类本身还是属于实例。**
- 在类中定义的成员，static相当于一个标志，有static修饰的成员属于类本身，没有static修饰的成员属于该类的实例。
- 类的static成员，需要**遵循static成员不能访问非static成员的规则**，例如，不能在类方法、类初始化块中访问实例变量。
- Java中虽然允许通过实例来访问static修饰的成员，但这是一个设计上的失误。希望同学们今后可以通过类名去访问类成员，这样程序的可读性、明确性都会大大提高。

本章导读

- 3.1 面向对象基础 OOP
- 3.2 类和对象 Class & Object
- ✓ 3.3 包 Package
- 3.4 访问权限 Access Privileges
- 3.5 综合作业及延伸

3.3 包 Package

- Java中的可复用软件资源以包的形式提供
- 每个包都针对某个领域：如网络，GUI（图形用户界面）等。
- 包中存放了彼此在功能上互补的多个类。
- 包类似于C++中的名字空间（name space）
- 为何要引入包？
 - 1 解决类的同名问题
 - 2 便于代码复用与维护

3.3 包_包的声明

- package 包名
 - package语句是Java源文件的第一条有效语句（忽略注释）
 - 如果源文件中没有package语句，则其中的类被默认为在无名包中
 - 包名可以是标识符，也可以是若干标识符加 ‘ ’ . ‘ ’ 组成，建议用小写
 - package adder;
 - package com.sun.jdk;

为了避免包名冲突，建议开发者用自己单位的域名的倒置作为包名

3.3 包_Import语句

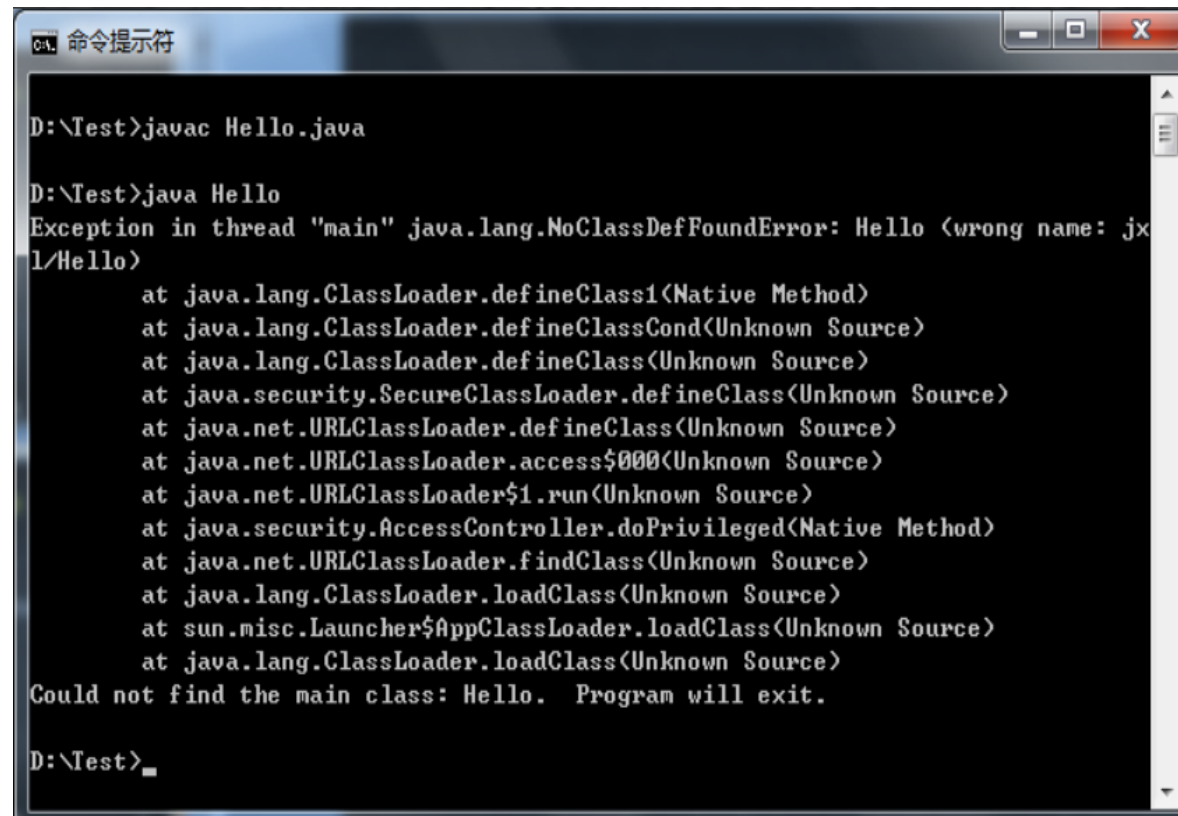
- 使用import语句可以引入包中的类
- import语句必须写在package语句和源文件中类的定义之间
- 引入一个包中的全部类, “*”被称为通配符
 - import java.applet.*;
 - import java.awt.*;
- 引入一个包中的某个类
 - import java.util.Arrays;

作业

- 在如下左图的源文件中添加package语句后，使用命令行运行出现了如右图所示的Error，请思考原因，并且通过查阅资料，找出正确运行的办法，下节课会随机抽查。

```
package jxl;
```

```
public class Hello  
{  
    public static void main(String[] args)  
    {  
        System.out.println("Hello World!");  
    }  
}
```



```
命令提示符  
D:\Test>javac Hello.java  
  
D:\Test>java Hello  
Exception in thread "main" java.lang.NoClassDefFoundError: Hello (wrong name: jxl/Hello)  
    at java.lang.ClassLoader.defineClass1(Native Method)  
    at java.lang.ClassLoader.defineClassCond(Unknown Source)  
    at java.lang.ClassLoader.defineClass(Unknown Source)  
    at java.security.SecureClassLoader.defineClass(Unknown Source)  
    at java.net.URLClassLoader.defineClass(Unknown Source)  
    at java.net.URLClassLoader.access$000(Unknown Source)  
    at java.net.URLClassLoader$1.run(Unknown Source)  
    at java.security.AccessController.doPrivileged(Native Method)  
    at java.net.URLClassLoader.findClass(Unknown Source)  
    at java.lang.ClassLoader.loadClass(Unknown Source)  
    at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)  
    at java.lang.ClassLoader.loadClass(Unknown Source)  
Could not find the main class: Hello.  Program will exit.  
  
D:\Test>
```

3.3 包_Java编程过程中的“路径”问题小结

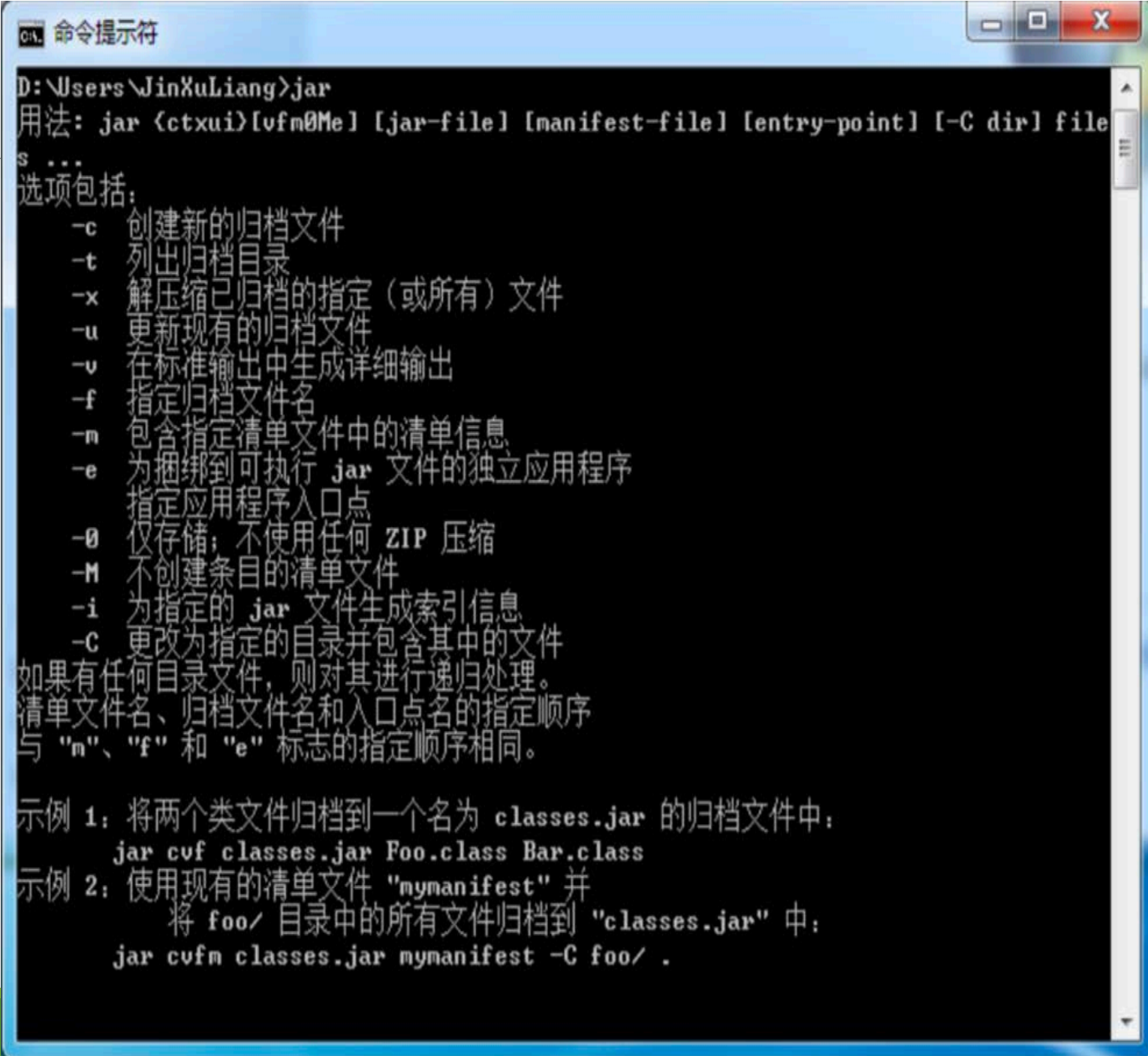
- Java公有类必须放入同名的源文件中，编译生成.class文件，如果使用package语句指定了类所属的包名，则.class文件应该放入包所对应的文件夹下。
- 如果代码中使用了另外包中的类，必须保证此类的 .class 文件可以在与包名匹配的路径中找到。
- 可以使用 CLASSPATH 环境变量指明 .class 文件的查找路径。
- 为降低复杂程度，可以使用 **Jar** 将所有相关文件都打包到同一个压缩包中。

3.3 包_Jar文件

- Jar文件其实是一个压缩包，遵循于Zip数据压缩标准，可以使用WinRAR等解压软件打开。
- Jar文件中包容一个清单（manifest）文件，包容一些重要信息，例如指定Main-Class。
- **可执行的Jar包**，（可以在windows资源管理器中“双击”执行的jar包），其清单文件必须指明包中哪个类是“主类（main class）”，从而让JVM知道应该从哪个类中的main方法开始执行。

作业

- 了解Jar命令，如何创建归档文件。
- 用Java编写一个MathOpt类，其中使用Java提供的Math类，自己编写一些一些代码，封装一些数学函数功能，把此类放入MathPackage包中，和提供的测试类一起使用jar命令打包成一个可运行的Jar包。



```
命令提示符
D:\Users\JinXuLiang>jar
用法: jar <ctxui>[vfm0Me] [jar-file] [manifest-file] [entry-point] [-C dir] file
s ...
选项包括:
-c 创建新的归档文件
-t 列出归档目录
-x 解压缩已归档的指定(或所有)文件
-u 更新现有的归档文件
-v 在标准输出中生成详细输出
-f 指定归档文件名
-m 包含指定清单文件中的清单信息
-e 为捆绑到可执行 jar 文件的独立应用程序指定应用程序入口点
-0 仅存储; 不使用任何 ZIP 压缩
-M 不创建条目的清单文件
-i 为指定的 jar 文件生成索引信息
-C 更改为指定的目录并包含其中的文件
如果有任意目录文件, 则对其进行递归处理。
清单文件名、归档文件名和入口点名的指定顺序与 "m"、"f" 和 "e" 标志的指定顺序相同。

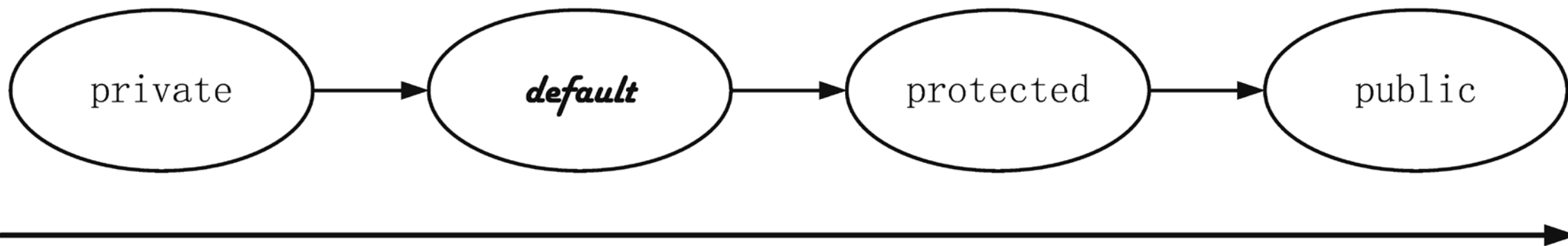
示例 1: 将两个类文件归档到一个名为 classes.jar 的归档文件中:
jar cvf classes.jar Foo.class Bar.class
示例 2: 使用现有的清单文件 "mymanifest" 并
将 foo/ 目录中的所有文件归档到 "classes.jar" 中:
jar cvfm classes.jar mymanifest -C foo/ .
```


本章导读

- 3.1 面向对象基础 OOP
- 3.2 类和对象 Class & Object
- 3.3 包 Package
- ✓ 3.4 访问权限 Access Privileges
- 3.5 综合作业及延伸

3.4 访问权限Access Privileges

- Java提供了3个访问控制符：private、protected和public，分别代表了3个访问控制级别，另外还有一个不加任何访问控制的访问控制级别（default）。因此，Java一共提供了4最种访问控制级别。
- Java通过包机制与访问控制权限来实现面向对象的“封装”。达到把该隐藏的隐藏起来，该暴露的暴露出来的目的。



访问控制级别由小到大

3.4 访问权限_private（当前类访问权限）

- 如果类里的一个成员（包括成员变量、方法、构造器、内部类）使用private访问控制符来修饰，则这个成员只能在当前类的内部被访问。
- 该访问控制符用于修饰成员变量最合适，使用它来修饰成员变量就可以把成员变量隐藏在该类的内部。
- 对某个类的私有静态方法或私有静态变量，其他类不能用类名对它们直接访问。

3.4 访问权限_default（包访问权限）

- 如果类里的一个成员（包括成员变量、方法、构造器、内部类）或者一个外部类不使用任何访问控制符修饰，就称它是包访问权限的，default访问控制的成员或外部类可以被相同包下的其他类访问。

3.4 访问权限_protected（子类访问权限）

- 如果类里的一个成员（包括成员变量、方法、构造器）使用protected访问控制符来修饰，则这个成员既可以被同一个包中的其他类访问，也可以被不同包中的子类访问。
- 通常情况下，如果使用protected来修饰一个方法，一般是**希望其子类来重写这个方法**。（关于父类、子类的介绍请参考第4章）

3.4 访问权限_public（公共访问权限）

- 这是最为宽松的一个访问控制级别，如果类的一个成员（包括成员变量、成员方法、构造器、内部类）或者一个外部类使用public访问控制符修饰，那么这个成员或者外部类就可以被所有类访问，不管访问类与被访问类是否处于同一个包，是否具有父子继承关系。
- 一个类中，用public修饰的成员变量和成员方法称为公有变量和公有方法
- 一般配合 private 隐藏成员变量，使用 public 暴露其对应的getter和setter方法，来访问以及修改对应的成员变量的值。

```
/**
 * 自定义Java类的示例
 */
class MyClass {
    // 公有字段
    public String Information = "";

    // 自定义公有Java实例方法
    public void myMethod(String argu) {
        System.out.println(argu);
    }

    // 定义属性：私有字段+get方法+set方法
    private int value;
    public int getValue() {
        return value;
    }
    public void setValue(int value) {
        this.value = value;
    }
}
```

3.4 访问权限

修饰符	是否有访问权限			
	同类	同包	非同包子类	全局
private	✓			
default	✓	✓		
protected	✓	✓	✓	
public	✓	✓	✓	✓

本章导读

- 3.1 面向对象基础 OOP
- 3.2 类和对象 Class & Object
- 3.3 包 Package
- 3.4 访问权限 Access Privileges
- ✓ 3.5 综合作业及延伸

实战——梦回C++

- 请使用C++语言编写一个Student类，要求声明和实现分离，具体要求如下：
 - 1. 至少包含以下字段：姓名、学号、成绩单（包括课程名称、学时、分数，可以使用数组或者链表实现，由于每个学生选修的课程数量不一致，成绩单的项数不固定）、学分积（要求根据成绩单中各课程的学时数来计算分数的加权平均）
 - 2. 方法：构造函数以及析构函数不能省略，可以根据要求自行添加所需要的方法，至少有一组函数重载。
 - 3. 具备良好的封装，不允许直接操作对象的属性，必须通过相应的方法对对象的属性进行添加、删除、修改、查询。

举个“栗子”

_Student.h文件

- 回忆！这个头文件中涉及到了哪些cpp的知识点？

```
#include<string>

using std::string;

class Student {
private:
    typedef struct _course_ {
        string _courseNumber;
        string _courseName;
        int _courseHour;
        double _score;
        struct _course_ *next;
    }course, *courseList;
private:
    string name;
    int NO;
    double averScore;
    courseList report;

private:
    void calcAverScore();

public:
    Student();
    Student(string name, int NO, double averScoer);
    ~Student();

    void setName(string);
    void setNO(int);
    string getName();
    int getNO();
    double getAverScore();

    bool addCourse(string NO, string name, int hour, double score);
    bool delCourse(string NO);
    bool updateCourse(string NO, double score);
    void printReport();
    void printReport(string NO);
};
```

举个“栗子” Student.h文件

●回忆！这个头文件中涉及到了哪些cpp的知识点？

构造函数&析构函数

构造函数重载

方法重载

链表

访问权限控制

.....

```
#include<string>

using std::string;

class Student {
private:
    typedef struct _course_ {
        string _courseNumber;
        string _courseName;
        int _courseHour;
        double _score;
        struct _course_ *next;
    }course, *courseList;
private:
    string name;
    int NO;
    double averScore;
    courseList report;

private:
    void calcAverScore();

public:
    Student();
    Student(string name, int NO, double averScoer);
    ~Student();

    void setName(string);
    void setNO(int);
    string getName();
    int getNO();
    double getAverScore();

    bool addCourse(string NO, string name, int hour, double score);
    bool delCourse(string NO);
    bool updateCourse(string NO, double score);
    void printReport();
    void printReport(string NO);
};
```

穿越时空——Java版本

```
class Course {  
    private String courseNumber;  
    private String courseName;  
    private int courseHour;  
    private double score;  
    private Course next;  
  
    public String getCourseNumber() {  
        return courseNumber;  
    }  
  
    public void setCourseNumber(String courseNumber) {  
        this.courseNumber = courseNumber;  
    }  
  
    // other getter and setter method  
}
```

```
public class Student {  
    private String name;  
    private int NO;  
    private double averScore;  
    private Course report;  
  
    public Student() {  
        // ...  
    }  
  
    public Student(String name, int NO, double averScore) {  
        this.name = name;  
        this.NO = NO;  
        this.averScore = averScore;  
        this.report = new Course();  
        this.report.setNext(null);  
    }  
  
    public static void main(String[] args) {  
        // 测试程序  
    }  
  
    private void calcAverScore() {  
        // ...  
    }  
  
    // some getter and setter method  
  
    public boolean addCourse(String NO, String name, int hour, double score) {  
        // ...  
    }  
  
    public boolean delCourse(String NO) {  
        // ...  
    }  
  
    public boolean updateCourse(String NO, double score) {  
        // ...  
    }  
  
    public void printReport() {  
        // ...  
    }  
  
    public void printReport(String NO) {  
        // ...  
    }  
}
```

综合练习

- 我们给出了Student类的cpp和Java版本的框架，请完成具体的代码实现！
- 测试类课程文件中已给出。

```
public class Student {
    private String name;
    private int NO;
    private double averScore;
    private Course report;

    public Student() {
        // ...
    }

    public Student(String name, int NO, double averScore) {
        this.name = name;
        this.NO = NO;
        this.averScore = averScore;
        this.report = new Course();
        this.report.setNext(null);
    }

    public static void main(String[] args) {
        // 测试程序
    }

    private void calcAverScore() {
        // ...
    }

    // some getter and setter method

    public boolean addCourse(String NO, String name, int hour, double score) {
        // ...
    }

    public boolean delCourse(String NO) {
        // ...
    }

    public boolean updateCourse(String NO, double score) {
        // ...
    }

    public void printReport() {
        // ...
    }

    public void printReport(String NO) {
        // ...
    }
}
```

延伸_private访问权限

- private访问控制符可以修饰一个类的构造器，请同学思考，这个特性是否有必要？
 - 若无必要，请至少给出三点理由，若有必要，请写出示例代码，需要提交你的结论！
 - 参考：设计模式之单例模式
- private访问控制符修饰一个类的实例方法的时候，又是在什么样的场景下，请童鞋们进行思考，很快，我们会在实验中遇到这一场景！

延伸_栈内存与堆内存

●栈内存：

在函数中定义的一些基本类型的变量和对象的引用变量都在函数的栈内存中分配。栈内存主要存放的是基本类型类型的数据 如 (int, short, long, byte, float, double, boolean, char) 和对象句柄。在栈内存的数据的大小及生存周期是必须确定的、其优点是寄存速度快、栈数据可以共享、缺点是数据固定、不够灵活。

●堆内存：

堆内存用来存放所有new 创建的对象和数组的数据

延伸_栈和堆共享的比较

- 请思考如下两段程序的运行结果：

- 栈的共享：

```
String str1 = "myString";  
String str2 = "myString";  
System.out.println(str1 == str2 );
```

- 堆的共享：

```
String str1 = new String ("myString");  
String str2 = new String ("myString");  
System.out.println(str1 == str2);
```


延伸_栈和堆共享的比较

- 请思考如下两段程序的运行结果：
- 栈的共享：

```
String str1 = "myString";  
  
String str2 = "myString";  
  
System.out.println(str1 == str2 );
```

```
jshell> String str1 = "mystring"  
str1 ==> "mystring"  
| 已创建 变量 str1 : String  
  
jshell> String str2 = "mystring"  
str2 ==> "mystring"  
| 已创建 变量 str2 : String  
  
jshell> str1 == str2  
$3 ==> true  
| 已创建 暂存变量 $3 : boolean
```

上述代码的原理是，首先在栈中创建一个变量为str1的引用，然后查找栈中是否有myString这个值，如果没找到，就将myString存放进来，然后将str1指向myString。接着处理String str2 = “myString”，在创建完str2 的引用变量后，因为在栈中已经有myString这个值，便将str2 直接指向myString。这样，就出现了str1与str2 同时指向myString。

延伸_栈和堆共享的比较

- 请思考如下两段程序的运行结果：
- 堆的共享：

```
String str1 = new String ("myString");  
String str2 = new String ("myString");  
System.out.println(str1 == str2);
```

```
jshell> String str1 = new String("mystring")  
str1 ==> "mystring"  
| 已修改 变量 str1 : String  
| 更新已覆盖 变量 str1 : String  
  
jshell> String str2 = new String("mystring")  
str2 ==> "mystring"  
| 已修改 变量 str2 : String  
| 更新已覆盖 变量 str2 : String  
  
jshell> str1 == str2  
$6 ==> false  
| 已创建暂存变量 $6 : boolean
```

创建了两个引用，创建了两个对象。两个引用分别指向不同的两个对象。

延伸_栈和堆共享的比较

- 顺其自然，我们就会有如下的问题：

```
String str1 = new String ("myString");  
  
String str2 = "myString";  
  
System.out.println(str1 ==str2 );
```

可见只要是用new()来新建对象的，都会在堆中创建，而且其字符串是单独存值的，即使与栈中的数据相同，也不会与栈中的数据共享。

```
jshell> String str1 = new String("mystring")  
str1 ==> "mystring"  
| 已创建 变量 str1 : String  
  
jshell> String str2 = "mystring"  
str2 ==> "mystring"  
| 已创建 变量 str2 : String  
  
jshell> str1 == str2  
$3 ==> false  
| 已创建 暂存变量 $3 : boolean
```

延伸_字符串比较 与 ==运算符

- 本章最后的思考！ 嗯（加个复数）
- 相信此时很多童鞋心中有所疑问，明明str1与str2的字符串完全相同，为什么使用==运算符进行比较的时候，返回的却是false？
- 虽然此时，应该模糊的知道，**==运算符在比较引用类型变量时，只有在两个引用都指向了同一个对象时才返回真值。**
- 那么，字符串的字面值比较该如何做呢？请同学们查找Java API给出答案，并给出示例程序。
- 同时，是否可以稍稍理解，为什么直到Java 7版本switch才支持使用string表达式进行判断，下节课，会随机找三名同学来谈谈对此的看法哦~