

第1周 环境搭建与语法入门

主要内容

- Java简介
- 开发环境搭建
- 开发第一个Java程序
- 使用Eclipse进行Java开发

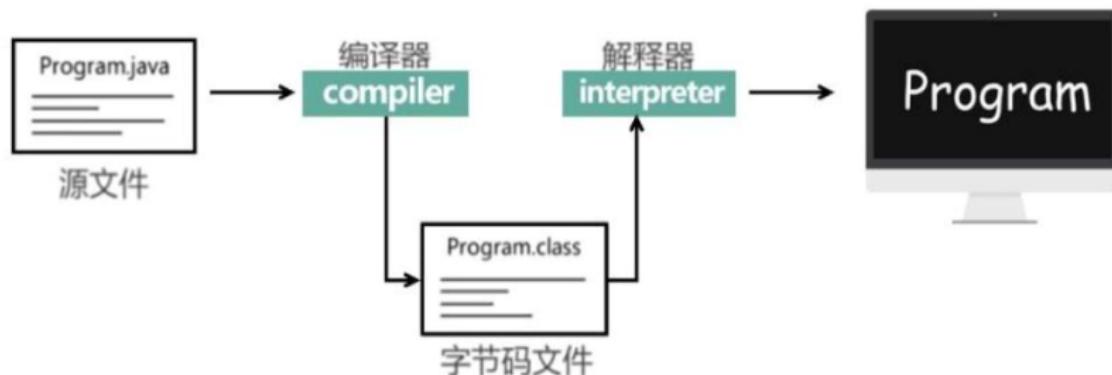
Java简介

- Java是一门面向对象的程序设计语言
- 1995年由sun公司发布
- 2010年被Oracle公司收购
- 现在的JDK版本是 8.0

JVM

- JVM(Java Virtual Machine), Java虚拟机
- JVM是Java平台无关性实现的关键

Java程序执行过程



JDK

- JDK (Java Development Kit) , Java语言的软件开发工具包。
- 两个主要组件：
 - javac -编译器，将源程序转成字节码
 - -java -运行编译后的java程序 (.class后缀的)

JRE

- JRE(Java Runtime Environment)
- 包括Java虚拟机（JVM）、Java核心类库和支持文件
- 如果只需要运行Java程序，下载并安装JRE即可
- 如果要开发Java软件，需要下载JDK

- 在JDK中附带有JRE

JDK、JRE和JVM三者的关系



Java平台



Java程序的结构

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

JDK环境搭建

JDK的下载和安装

Java程序的执行流程

变量与常量

主要内容

- 标识符
- 关键字
- 变量
- 数据类型
- 类型转换
- 常量

标识符

标识符的命名规则

- 标识符可以由字母、数字、下划线（_）和美元符(\$) 组成，不能以数字开头
- 标识符严格区分大小写
- 标识符不能是Java关键字和保留字
- 标识符的命名最好能反映出其作用

关键字

abstract	boolean	break	byte	case	catch
char	class	continue	default	do	double
else	extends	false	final	finally	float
for	if	implements	import	native	int
interface	long	instanceof	new	null	package
private	protected	public	return	short	static
super	switch	synchronized	this	throw	throws
transient	true	try	void	volatile	while

变量

变量的三个元素：变量类型、变量名和变量值。

变量名的命名规则

- 满足标识符命名规则
- 符合驼峰法命名规范
- 尽量简单，做到见名知意
- 变量名的长度没有限制

类的命名规则

- 满足Pascal命名法规范(与骆驼命名法类似只不过骆驼命名法是首字母小写，而帕斯卡命名法是首字母大写)

数据类型



123	int类型的字面值
123L 123l	long类型的字面值
1.23	double类型的字面值
1.23f 1.23F	float类型的字面值
true false	boolean类型字面值

基本数据类型

数据类型		字节
byte		1
short		2
int		4
long		8
float		4
double		8
char		2
boolean		1

进制表示

- 八进制：以0开头，包括0-7的数字
如：037, 056
- 十六进制表示：以0x或0X开头，包括0-9的数字，及字母a-f, A-F
如：0x12, 0xabcf, 0XABCFF

整型字面值

Java中有三种表示整数的方法：十进制、八进制、十六进制

如：123, 023, 0x1357, 0X3c, 0x1abcL

变量声明

- 格式：数据类型 变量名；
例：
int n; 声明整型变量n
long count; 声明长整型变量count

赋值

- 使用“=”运算符进行赋值
- “=”叫作赋值运算符，将运算符右边的值赋给左边的变量
 - 例：int n; 定义int型变量n
 - n=3; 将3赋值给n
- 可以在定义变量的同时给变量赋值，即变量的初始化。
 - 例：int n=3;
 - 数据类型 变量名=变量值;

变量定义

- int octal=037; //定义int类型变量存放八进制数据
- long longNumber=0xa2cdf3ffL; //定义变量存放十六进制长整型数据
- short shortNumber=123; //定义变量存放短整型数据
- byte b=10; //定义变量存放byte类型数据

说明：

整型字面值默认情况下是int类型，如果表示长整型则在末尾加L或l

语句

- 以分号结束
- 不能换到一行写

浮点型字面值

- 浮点型字面值默认情况下表示double类型，也可以在值后加d或D

如：123.43d 或 123.43D

如表示float类型，则需要在字面值后加f或F

如：23.4f 或 23.4F

基本数据类型变量的存储

- 类型

- 数据类型分为基本数据类型和引用数据类型

。 - 引用数据类型包括数组和类等

- 类定义的变量又叫对象

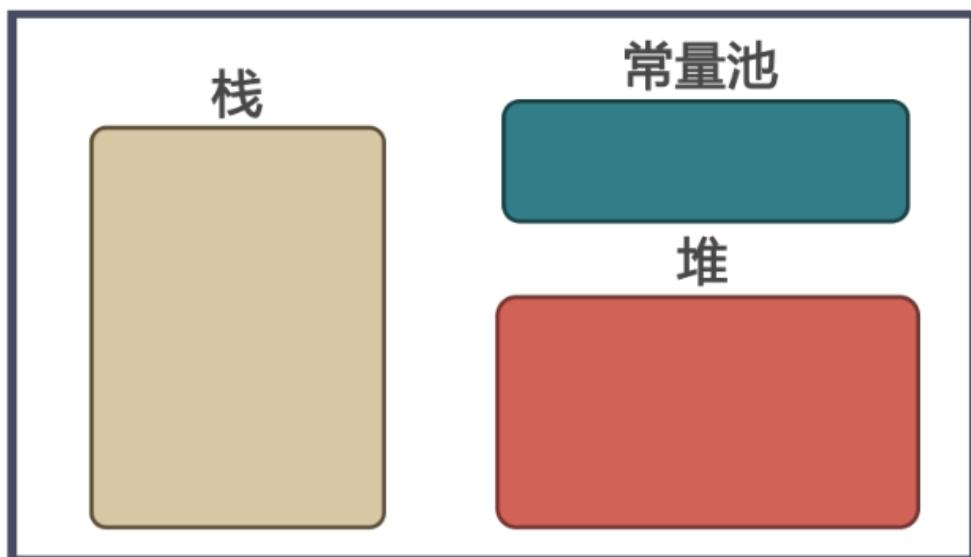
- 范围

按照作用范围分为：

◦ - 类级、对象实例级、方法级、块级

•

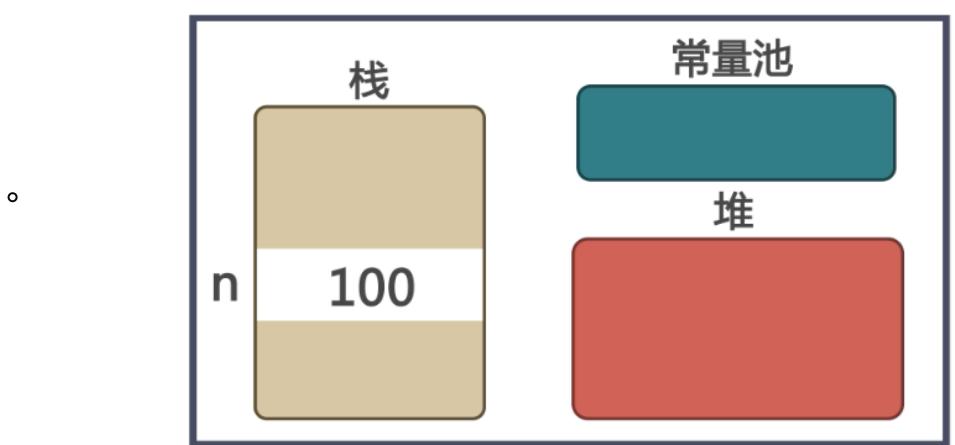
内存



- 局部变量的存储

`int n=100;`

内存



字符型字面值

- 字符型字面值用单引号内的单个字符表示
如: 'a', 'b', '\$'
- 如何定义字符型变量?
`char a='a';`
`char ch=65;`

ASCII码

- ASCII (American Standard Code for Information Interchange, 美国标准信息交换代码)
- 基于拉丁字母的一套电脑编码系统
- 主要用于显示现代英语和其他西欧语言

ASCII码表示

- 使用7位或8位二进制数组合来表示128（标准）或256（拓展）种可能的字符。
- 标准ASCII码使用7位二进制来表示所有的大写字母和小写字母，数字0-9，标点符号，以及在美式英语中使用的控制字符。
- 后128个称为扩展ASCII码，用于表示特殊符号、外来语字母和图形符号。

Unicode编码

- `char c='\\u005d';` (四位)
- Unicode表示法，在值前加前缀\u

布尔类型字面值

- 布尔值只能定义为true和false
例：`boolean b=true;`

字符串字面值

- 双引号引起的0个或多个字符。
- 类

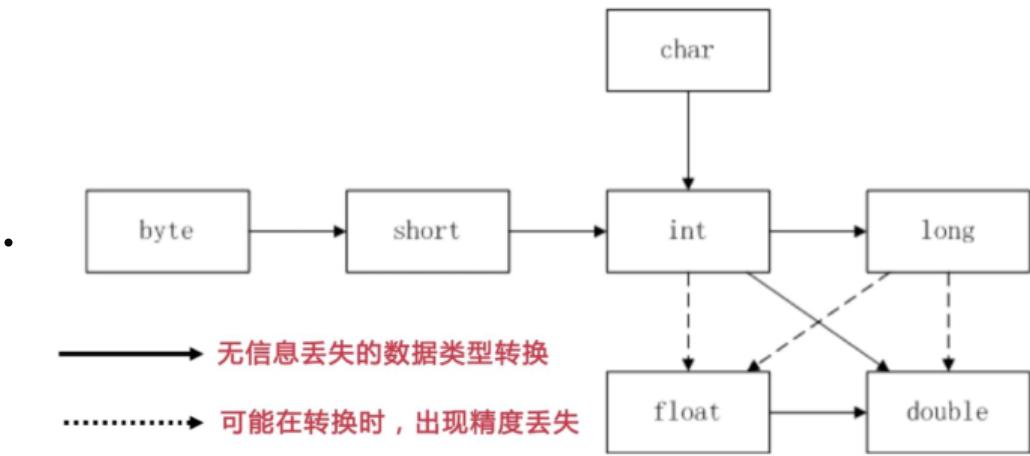
转义字符

转义字符	描述
<code>\\xxxx</code>	四位16进制数所表示的字符
<code>'</code>	单引号字符
<code>"</code>	双引号字符
<code>\\</code>	反斜杠字符
<code>\\r</code>	回车
<code>\\n</code>	换行
<code>\\t</code>	横向跳格
<code>\\b</code>	退格

类型转换

- 类型转换分为自动类型转换和强制类型转换

自动类型转换顺序



强制类型转换

- 如果A类型的数据表示范围比B类型大，则将A类型的值赋值给B类型，需要强制类型转换
如：double d=123.4;
float f=(float)d;

常量

- `final int n=5;`

第3周 面向对象之封装与继承

第1节 Java面向对象

什么是对象

- 万物皆对象、客观存在的事物
- 对象：用来描述客观事物的一个实体，由一组属性和方法构成

什么是面向对象

- 人关注对象
- 人关注事物信息

什么是类

- 类是模子，确定对象将会拥有的特征（属性）和行为（方法）
- 类的特点
 - 类是对象的类型
 - 具有相同属性和方法的一组对象的集合

什么是对象的属性和方法

- 属性：对象具有的各种静态特征→“有什么”
- 方法：对象具有的各种动态行为→“能做什么”

类和对象的关系

- 类是抽象的概念，仅仅是模板
- 对象是一个你能够看得到、摸得着的具体实体
- 类是对象的类型
- 对象是特定类型的数据
- 具体开发过程中，先定义类再实例化对象

单一职责原则

- 单一职责原则，也称为单一功能原则
- 英文Single Responsibility Principle 缩写SRP
- 是面向对象设计中的一个重要原则
- 一个类，应该有且只有一个引起变化的原因
- 在程序设计中，尽量把不同的职责，放在不同的职责中，即把不同的变化原因，封装到不同的类中。

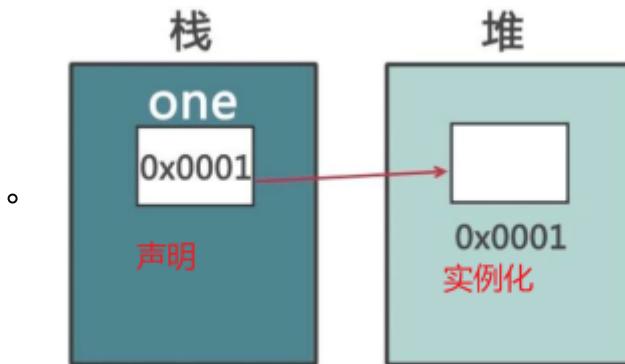
单一职责原则 (SRP: Single responsibility principle) 又称单一功能原则，面向对象五个基本原则 (SOLID: SRP 单一责任原则、OCP 开放封闭原则、LSP 里氏替换原则、DIP 依赖倒置原则、ISP 接口分离原则) 之一。它规定一个类应该只有一个发生变化的原因。

对象实例化

- 实例化对象的过程可以分为两部分：

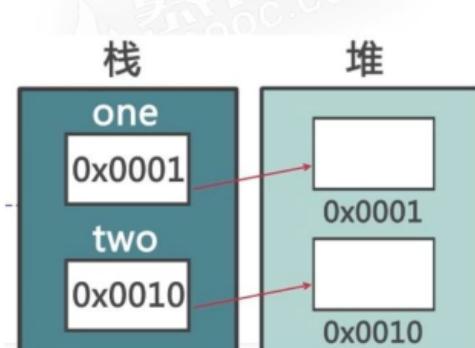
1. 声明对象 Cat one
2. 实例化对象 new Cat();

- 声明与实例化的存储



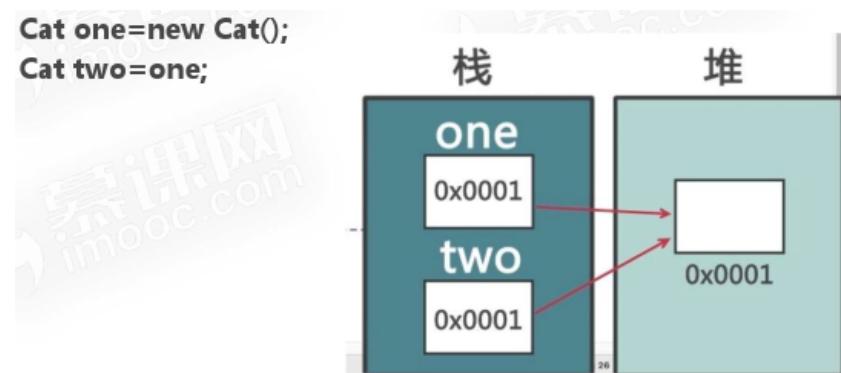
- 多个对象可以指向同一块实例化空间有或者是每次new对象会产生新的实例化对象
 - 实例化新的对象

```
Cat one=new Cat();
Cat two=new Cat();
```



- 同一个实例空间

```
Cat one=new Cat();
Cat two=one;
```



备忘:

- 对象必须被实例化之后才能使用
- 对象间的引用传递，实际上传递的是堆内存空间的使用权

构造方法

- 构造方法与类同名且没有返回值
- 构造方法的语句格式
- 只能在对象实例化的时候调用
- 当没有指定构造方法时，系统会自动添加无参的构造方法
- 当有指定构造方法，无论是有参、无参的构造方法，都不会自动添加无参的构造方法
- 一个类中可以有多个构造方法



this关键字

- `this`: 当前对象的默认引用
- `this`的使用
 - 调用成员变量，解决成员属性和局部变量同名冲突

```

    // 成员属性：昵称、年龄、体重、品种
    String name;// 昵称
    int month;// 年龄
    public Cat(String name) {
        this.name=name;
        System.out.println("我是单参构造");
    }

```

◦ 调用成员方法

```

    // 成员方法：跑动、吃东西
    // 跑动的方法
    public void run() {
        this.eat();
        System.out.println("小猫快跑");
    }

```

◦ 调用成员方法

```

    // 吃东西的方法
    public void eat() {
        System.out.println("小猫吃鱼");
    }

```

◦ 调用重载的构造方法

```

    public Cat(){
        System.out.println("我是无参构造");
    }

    public Cat(String name) {
        this();
        this.name=name;
        System.out.println("我是单参构造");
    }

```

##

第2节 Java封装

Java中一个包里不能
存在同名类

· 域名倒序+模块+功能

域名全部小写

必须放在Java源文件
中的第一行

建议每个包内存储
信息功能单一

封装

- 将类的某些信息隐藏在类内部，不允许外部程序直接访问
- 通过该类提供的方法来实现对隐藏信息的操作和访问
- 隐藏对象的信息
- 留出访问的接口
- 特点：
 1. 只能通过规定的方法访问数据
 2. 隐藏类的实例细节，方便修改和实现
- 步骤

封装



- 常见问题

1-5 关于封装应用中的常见问题

1、如果不使用封装，在要调用的普通成员方法中编写相关限制代码，实现避免在主方法中所调用属性及方法的值被非法篡改，这样不可以吗，为什么一定要用封装？

在面向对象的设计思想中，封装可以理解为是一种利用抽象的函数接口实现细节信息的包装隐藏的方式。我们可以把封装认为是一个保护屏障，防止该类的私密代码和数据被外部类定义的代码随机访问和修改。简单来说，就是“按我的规则，才能玩我的游戏”。而在隐藏信息的同时，我们还要注意“职责单一”原则的应用，也就是“各司其职”。

如果只是从功能实现的角度来说，当然可以将限制代码写在任意的功能实现方法中，但是试想，如果一个类中，有10个功能性方法中需要针对某一属性进行相同的设定，我们是设置一次方便，还是设置十次更方便安全呢？

因此，适当的封装可以让代码更容易理解与维护，也加强了安全性。调用者不能随意通过“变量名.属性名”的方式来修改类的私密数据信息；同时，在使用的时候，也只需直接调用封装后的方法即可，无需再操心细节处理。

2、get/set 用两个方法实现取值、赋值，放在一个方法里不是更简单？是否可以改成别的名字呢？

如果仅仅是为了实现功能，那么，无论是写在一个方法里，还是用其他名字命名方法都是OK的。但是，在基于面向对象的编程思想中，更推荐大家采用get/set方法分别实现“取值”和“赋值”的功能，让他们“各司其职”，也更加“通俗易懂”，毕竟当业务越来越复杂，团队协作的时候，“约定俗成”会比“各有千秋”更有价值。

3、有了get/set方法，为什么还需要带参数构造方法？或者说，在构造方法中直接写if...else...判断限制输入输出结果不行么，为什么要多写两个方法get/set？

构造方法与get/set方法的作用是不同的，构造方法只能在创建对象时进行调用，如果在对象构建完成后，再想对其某些属性进行赋值和取值，就无法再次应用构造方法啦。因此两者的存在并不冲突，我们可以应用带参数构造在对象初始化时进行某些属性的设置，也可以通过get/set方法，在对象构建完成后进行后续修订。

4、main方法中为什么可以添加return？什么时候能加？有什么作用？是否可以用break进行替换？

关于return和break的应用：

- 当方法中出现return表示方法运行终止；当出现break则表示循环语句或者switch语句运行结束。
- 如果方法设置了返回值，那么必须出现return，应用return带回返回值；如果方法的返回值为void，也可以出现return，但是后面什么也不能加，直接加分号结束。

因此，主方法中的return语句不能与break随意替换。

5、private修饰的成员怎么用？

- 在同一个类中，可以直接访问。譬如：

```
public class Cat {  
    //私有成员属性  
    private String name;  
  
    //私有成员方法  
    private void call() {  
        //同一个类中，可以直接访问类中私有成员  
        System.out.println(name+"在说悄悄话");  
    }  
  
    //公有成员方法  
    public void eat() {  
        //同一个类中，可以直接访问类中私有成员  
        call();  
        System.out.println("聊天结束，开吃~");  
    }  
}
```

- 在不同类中，需要通过对应的非私有方法访问。譬如：

```
public class Cat {  
    //私有成员属性  
    private String name;  
  
    //公有成员方法  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return this.name;  
    }  
}
```

```
public class CatTest {  
    public static void main(String[] args) {  
        Cat one=new Cat();  
        one.setName("花花");  
        System.out.print(one.getName());  
    }  
}
```

包

- 作用：

- 1. 管理Java文件
- 2. 解决同名文件冲突

- 定义包

语法：

package 包名;

例：package com.imooc.animal;

- 注意：

- 1、必须放在Java源文件中的第一行
- 2、一个Java源文件中只能有一个package语句
- 3、包名全部英文小写
- 4、命名方式：域名倒序+模块+功能

- 导入包

语法：

import 包名.类名;

例：

- 导入包中全部类：

import com.imooc.*;

导入包中指定类：

import com.imooc.animal.Cat;

- 常用系统包

tips :

常用系统包

-

java.lang	包含Java语言基础的类，该包系统加载时默认导入 如：System、String、Math
java.util	包含Java语言中常用工具 如：Scanner、Random
java.io	包含输入、输出相关功能的类 如：File、InputStream

static

1. static+属性---静态属性

2-2 static关键字的应用---静态属性

static是Java中常用的关键字，代表‘全局’或者‘静态’的意思。关于static的特征，可以理解为：方便在没有创建对象的情况下进行某些操作。通常可用于修饰成员变量和方法，也可以形成静态代码块。

实际应用中，可将需频繁操作、通用型信息设置、公共组件封装等操作设置为‘静态’。在本节中，我们将针对“static + 成员变量”的应用进行相关总结。

应用一：static + 成员变量 vs 成员变量

概念：

静态成员：用static修饰的成员变量，通常也称为静态成员、静态属性、类成员、全局属性等。

非静态成员：没有被static修饰的成员变量，也称为叫做非静态成员、实例变量、实例成员、对象成员、对象属性等。

特征：

静态成员：

- 静态成员是属于整个类的，由类所进行维护，仅在类首次加载时会被初始化，在类销毁时回收。
- 通过该类实例化的所有对象都共享类中静态资源，任一对象中信息的修订都将影响所有对象。
- 由于静态成员在类加载期间就已经完成初始化，存储在Java Heap（JDK7.0之前存储在方法区）中静态存储区，因此优先于对象而存在，可以通过类名和对象名两种方式访问。

非静态成员：

- 非静态成员属于对象独有，每个对象进行实例化时产生各自的成员，随着对象的回收而释放。
- 对象对各自成员信息的修订不影响其他对象。
- 只能通过对象名访问。

应用：

可以将频繁调用的公共信息、期望加快运行效率的成员设置为静态。但需注意，由于其生命周期长，即资源占用周期长，要慎用。

示例：

定义国产车类CarDemo，分别包含静态属性firm，对象属性color、price

定义测试类CarTest，针对CarDemo进行测试

备注：此处由于展示篇幅有限，暂不考虑属性的访问权限

```
public class CarDemo {  
    static String firm; // 厂商  
    String color; // 颜色  
    int price; // 价格  
  
    public CarDemo(String color, int price) {  
        this.color = color;  
        this.price = price;  
    }  
  
    // 信息展示方法  
    public void display() {  
        // 类内静态成员访问方式1：类名.成员名  
        System.out.println("本款汽车信息展示：厂商--" + CarDemo.firm + ", 颜色--" + this.color + "价格--" + this.price);  
  
        // 类内静态成员访问方式2：this.成员名  
        System.out.println("本款汽车信息展示：厂商--" + this.firm + ", 颜色--" + this.color + "价格--" + this.price);  
  
        // 类外静态成员访问方式3：成员名  
        System.out.println("本款汽车信息展示：厂商--" + firm + ", 颜色--" + this.color + "价格--" + this.price);  
    }  
  
}  
  
public class CarTest {  
  
    public static void main(String[] args) {  
        // 类外静态成员访问方式1：类名.成员名  
        CarDemo.firm = "一汽集团";  
  
        CarDemo one = new CarDemo("蓝色", 200000);  
        one.display();  
        CarDemo two = new CarDemo("红色", 1500000);  
        two.display();  
  
        // 类外静态成员访问方式2：对象名.成员名  
        two.firm = "上汽集团";  
  
        one.display();  
    }  
}
```

此时输出结果为：

```
[terminated> C:\Users\user\Desktop\Java Application] C:\Program Files\Java\jre1.8.0_121\bin\java.exe  
本款汽车信息展示：厂商--一汽集团，颜色--蓝色价格--200000  
本款汽车信息展示：厂商--一汽集团，颜色--红色价格--1500000  
本款汽车信息展示：厂商--上汽集团，颜色--蓝色价格--200000
```

总结：

- 类外可应用“类名.成员名”或“对象名.成员名”的方式访问非私有静态成员，更推荐用“类名.成员名”体现其特性。应用“对象名.成员名”时会出现警告，但不影响程序运行。
- 类内可应用“类名.成员名”或“this.成员名”或“成员”的方式访问，应用“this.成员名”时会出现警告，但不影响程序运行。

下一节

2. static+方法---静态方法

2-5 static关键字的应用---静态方法

在本节中，我们将针对“static + 成员方法”的应用进行相关内容总结。

应用二：static + 成员方法 vs 成员方法

概念：

静态方法：用static修饰的成员方法，通常也称为静态方法、类方法、全局方法等。

非静态方法：没有被static修饰的成员方法，也称为叫做非静态方法、实例方法、对象方法等。

特征：

与静态成员相似，静态方法属于整个类的，由类所进行维护，优先于对象而存在，因此可以通过类名和对象名两种方式访问，也因此在静态方法中无法直接访问同类中的非静态成员。

示例：

定义国产车类CarDemo，分别包含静态属性firm，对象属性color、price，静态方法display。

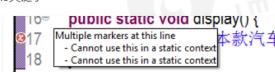
定义测试类CarTest，针对CarDemo进行测试。

备注：此处由于展示篇幅有限，暂不考虑属性的访问权限

· 当display方法中分别访问类内静态及非静态成员时：

```
6 public class CarDemo {  
7     static String firm; // 厂商  
8     String color; // 颜色  
9     int price; // 价格  
10    public CarDemo(String color, int price) {  
11        this.color = color;  
12        this.price = price;  
13    }  
14  
15    // 信息展示方法  
16    public static void display() {  
17        System.out.println("本款汽车信息展示：厂商--" + CarDemo.firm + ", 颜色--" + this.color + "价格--" + this.price);  
18    }  
19 }
```

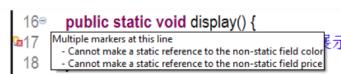
错误提示：不能在静态方法中应出现this关键字



· 当修改方法内部，去掉this

```
14  
15    // 信息展示方法  
16    public static void display() {  
17        System.out.println("本款汽车信息展示：厂商--" + CarDemo.firm + ", 颜色--" + color + "价格--" + price);  
18    }  
19 }
```

错误提示：不能在静态方法中应出现非静态成员



· 修订display方法体，成功访问类内静态及非静态方法

```
public static void display() {  
    // 类内静态成员访问方式1：类名.成员名  
    System.out.println("本款汽车信息展示：厂商--" + CarDemo.firm);  
    // 类内静态成员访问方式2：成员名  
    System.out.println("本款汽车信息展示：厂商--" + firm);  
  
    // 类内非静态成员访问方式：生成对象，通过对象访问  
    CarDemo demo = new CarDemo("蓝色", 400000);  
    System.out.println("本款汽车信息展示：颜色--" + demo.color + "价格--" + demo.price); // 同理可demo.run()  
}
```

总结：

1、静态方法中可以通过“类名.成员”或“成员”的方式访问类内静态成员/静态方法

2、不允许直接访问本类中的非静态成员/非静态方法

3、可以通过实例化产生本类对象，通过“对象.成员”的方式访问类内非静态成员/非静态方法。

· 当display方法中，仅保留静态成员firm

```
public class CarDemo {  
    static String firm; // 厂商  
    String color; // 颜色  
    int price; // 价格  
    public CarDemo(String color, int price) {  
        this.color = color;  
        this.price = price;  
    }  
  
    // 信息展示方法  
    public static void display() {  
        System.out.println("本款汽车信息展示：厂商--" + CarDemo.firm);  
    }  
}
```

测试类中

```
public class CarTest {  
    public static void main(String[] args) {  
  
        CarDemo.firm = "一汽集团";  
        // 类外静态方法访问方式1：类名.成员方法  
        CarDemo.display();  
  
        CarDemo one = new CarDemo("蓝色", 200000);  
  
        // 类外静态方法访问方式2：对象名.成员方法  
        one.display();  
    }  
}
```

此时输出结果为：

```
<terminated> CarTest [Java Application] C:\Program Files\Java\jre1.8.0_121\bin  
本款汽车信息展示：厂商--一汽集团  
本款汽车信息展示：厂商--一汽集团
```

```

总结:
1、类外可应用“类名.成员方法”或“对象名.成员方法”的方式访问非私有静态方法
2、应用“对象名.成员方法”时会出现警告，但不影响程序运行。
- 类中添加非静态方法run，并在其中访问类内静态成员

public void run() {
    // 类内静态成员访问方式1：类名.成员名
    System.out.println("本款汽车信息展示：厂商--" + CarDemo.firm); // 同理可CarDemo.display()
    // 类内静态成员访问方式2：成员名
    System.out.println("本款汽车信息展示：厂商--" + firm); // 同理可display()
    // 类内静态成员访问方式3：this.成员名
    System.out.println("本款汽车信息展示：厂商--" + this.firm); // 同理可this.display()
}

```

总结：
1、非静态方法可以通过“类名.成员法”或“成员”或“this.成员”的方式访问类内静态成员/静态方法
2、应用“this.静态成员/静态方法”时会出现警告，但不影响程序运行。

- 分别在run和display方法中添加静态局部变量test

```

16~ public void run() {
17     static int test=10;
18     // 类内静态成员访问方式1：类名.成员名
19     System.out.println("本款汽车信息展示：厂商--" + CarDemo.firm); // 同理可CarDemo.display()
20 }
21 // 信息展示方法
22~ public static void display() {
23     static int test=10;
24     // 类内静态成员访问方式1：类名.成员名
25     System.out.println("本款汽车信息展示：厂商--" + CarDemo.firm);
26
27     // 类内非静态成员访问方式：生成对象，通过对象访问
28     CarDemo demo = new CarDemo("蓝色", 400000);
29     System.out.println("本款汽车信息展示：颜色--" + demo.color + "价格--" + demo.price); // 同理可demo.run()
30 }
31 }

```

错误提示：test前的static修饰符无效，即不允许在方法内定义静态变量。

```

16~ public void run() {
17     static int test=10; // Illegal modifier for parameter test; only final is permitted
18     // 类内静态成员访问方式1：类名.成员名
19
22~ public static void display() {
23     static int test=10; // Illegal modifier for parameter test; only final is permitted
24     // 类内静态成员访问方式1：类名.成员名

```

当去掉局部变量前面的static，正常显示

```

public void run() {
    int test=10;
    // 类内静态成员访问方式1：类名.成员名
    System.out.println("本款汽车信息展示：厂商--" + CarDemo.firm); // 同理可CarDemo.display()
}
// 信息展示方法
public static void display() {
    int test=10;
    // 类内静态成员访问方式1：类名.成员名
    System.out.println("本款汽车信息展示：厂商--" + CarDemo.firm);

    // 类内非静态成员访问方式：生成对象，通过对象访问
    CarDemo demo = new CarDemo("蓝色", 400000);
    System.out.println("本款汽车信息展示：颜色--" + demo.color + "价格--" + demo.price); // 同理可demo.run()
}

```

总结：
不允许在方法内部定义静态局部变量。

下一节

1. 可以直接调用同类中的静态成员
2. 不可以直接调用同类中的非静态成员
3. 可以通过实例化对象后，对象调用的方式完成非静态成员调用

```

4. 1 // 吃东西的方法
2 // static+方法-->类方法、静态方法
3 public static void eat() {
4     // 静态方法中不能直接访问同一个类中的非静态成员，只能直接调用同一个类中的静态成员
5     // 只能通过对象实例化后，对象.成员方法的方式访问非静态成员
6     Cat temp = new Cat();
7     temp.run();
8     temp.name = "小胖"; // 静态方法中不能使用this
9     //      temp.name="小胖";
10    price = 1500;
11    System.out.println("小猫吃鱼");
12 }

```

3. static+类---不存在
4. static+方法内局部变量---不存在
5. 工具类，不需要实例化也行

代码块

1. 通过{}可以形成代码块
2. 方法内的代码块称为：普通代码块
3. 类内的代码块称为：构造代码块
4. 构造代码块前+static：静态代码块

2-11 static关键字的应用---静态代码块（上）

在本节中，我们将针对“static + 代码块”的应用进行相关内容，分别从概念、特征、应用进行总结。

应用三：static + 代码块 vs 代码块

概念：

静态代码块：被static修饰的，定义在类内部，用{}括起的代码段。

构造代码块：没有被static修饰的，定义在类内部，用{}括起的代码段。

普通代码块：定义在方法内部，用{}括起的代码段。

特征：

静态代码块：

- 只能出现在类内，不允许出现在方法内。
- 可以出现多次，按顺序在类加载时执行。
- 无论该类实例化多少对象，只执行一次。

构造代码块：

- 可以在类内出现多次，按顺序在每个对象实例化时执行。
- 执行优先级：晚于静态代码块，高于构造方法。
- 每次执行对象实例化时，均会执行一次。

普通代码块：

- 可以在方法内出现多次，按顺序在方法调用时执行。

应用：

静态代码块：基于性能优化的考量，多适用于需要在项目启动时执行一次的场景，譬如项目资源整体加载等。

构造代码块：多适用于类中每个对象产生时都需要执行的功能封装。与构造方法的区别在于，构造方法是在new执行时有选择性的调用带参或者无参构造，而构造代码块则是，在每个对象实例化时都一定会执行。

普通代码块：适用于在方法内进行代码功能拆分。

[下一节](#)

2-12 static关键字的应用---静态代码块（下）

在本节中，我们将针对“static + 代码块”的应用结合示例进行总结。

示例：

定义国产车类CarDemo，包含静态成员firm，非静态方法run；定义测试类CarTest，针对CarDemo进行测试

备注：此处由于展示篇幅有限，暂不考虑属性的访问权限

· 当CarDemo类在静态代码块和构造代码块中分别对静态成员赋值，并添加构造方法

```
public class CarDemo {
    static String firm; //厂商

    static {
        firm = "一汽";
        System.out.println("我是静态代码块");
    }

    {
        firm = "东风";
        System.out.println("我是构造代码块");
    }

    public CarDemo() {
        System.out.println("我是构造方法");
    }

    public void run() {
        System.out.println("本款汽车信息展示：厂商--" + CarDemo.firm);
    }
}
```

测试类CarTest中实例化CarDemo，并调用run方法

```
public class CarTest {

    public static void main(String[] args) {
        CarDemo one = new CarDemo();
        one.run();
        System.out.println("*****");
        CarDemo two = new CarDemo();
        two.run();
    }
}
```

```

    }
}

运行结果:
<terminated> CarTest [Java Application] C:\Program Files\Java\jre1.8.0_
我是静态代码块
我是构造代码块
我是构造方法
本款汽车信息展示: 厂商--东风
*****
我是构造代码块
我是构造方法
本款汽车信息展示: 厂商--东风

总结:
1、执行优先级: 静态代码块>构造代码块>构造方法
2、执行次数: 静态代码块只执行1次; 构造代码块、构造方法随对象实例化个数而定
- CarDemo类中添加成员属性color，并在静态代码块和构造代码块中分别对color进行赋值时
  6 public class CarDemo {
  7     static String firm; // 厂商
  8     String color;//颜色
  9
 10    static {
 11        firm = "一汽";
 12        color="蓝色";
 13        System.out.println("我是静态代码块");
 14    }
 15
 16    {
 17        firm="东风";
 18        color="蓝色";
 19        System.out.println("我是构造代码块");
 20    }
 21

错误提示: 不能在静态块中操作非静态成员
  10  static {
 11      firm = "一汽";
 12      Cannot make a static reference to the non-static field color
 13      System.out.println("我是静态代码块");

总结:
1、不能在静态代码块中直接对非静态成员赋值。
2、可以在构造代码块中直接操作静态和非静态成员。
- 当将属性定义移入静态代码块时
  6 public class CarDemo {
  7
  8     static {
  9         static String firm="一汽"; // 厂商
 10        String color="蓝色";
 11        System.out.println("我是静态代码块");
 12    }
 13
 14    {
 15        firm="东风";
 16        color="红色";
 17        System.out.println("我是构造代码块");
 18    }
 19

错误提示分别为: 不允许在静态代码块中声明静态成员; 无法在构造代码块中访问到相关成员,
  8  static {
 9      illegal modifier for the variable firm: only final is permitted
 10     String color="蓝色";
 11     System.out.println("我是静态代码块");
 12
 13
 14    {
 15        firm cannot be resolved to a variable
 16        color="红色";
 17        System.out.println("我是构造代码块");
 18    }
 19

总结:
1、不能在静态代码块中声明静态成员, 可以声明非静态成员。
2、静态代码块中声明的成员, 在外部无法进行访问。
- 当在run方法中分别添加2个代码块时
  public class CarDemo {
  public void run() {
  {
  int sum=10;
  System.out.println("我是普通代码块1: "+sum);
  }
  int sum=20;
  System.out.println("我是普通代码块2: "+sum);
  }
}

public class CarTest {
  public static void main(String[] args) {
  CarDemo one=new CarDemo();
  one.run();
  }
}

运行结果:
<terminated> CarTest [Java Application] C:\Program I
我是普通代码块1: 10
我是普通代码块2: 20

```

- 当在两个代码块上部添加同名局部变量声明时:

The screenshot shows a Java code editor with the following code:

```
6 public class CarDemo {  
7     public void run() {  
8         int sum=50;  
9         {  
10            int sum=10;  
11            System.out.println("我是普通代码块1: "+sum);  
12        }  
13        {  
14            int sum=20;  
15            System.out.println("我是普通代码块2: "+sum);  
16        }  
17    }  
18 }
```

A tooltip "Duplicate local variable sum" appears over the second declaration of the variable "sum" at line 10. Below the code, a message says "错误提示: 不允许定义重名变量".

总结:

- 1、普通代码块在方法内顺序执行，各自作用范围独立。
- 2、方法内定义的局部变量，作用范围为：自定义位置起，至方法结束。在此期间，不允许方法中普通代码块内存在局部变量的声明。

下一节

- 代码块执行顺序

- 无论实例产生多少对象，静态代码块只执行一次
- 构造代码块随实例化过程调用
- 普通代码块随方法调用执行