



面向对象的软件构造导论

第三章：类和对象



第二章（课程回顾）

- 标识符、关键字、分隔符、注释
- Java基本数据类型、常量
- 控制流程
- 输入与输出
- 数组
- 异常机制
- Java虚拟机与垃圾回收



标识符

□ 标识符: 用来标识类名、变量名、方法名、对象名、数组名、文件名的有效字符系列。

□ 标识符的规定:

标识符可由英文字母、数字、下划线、美元符号\$组合而成，长度不受限制。

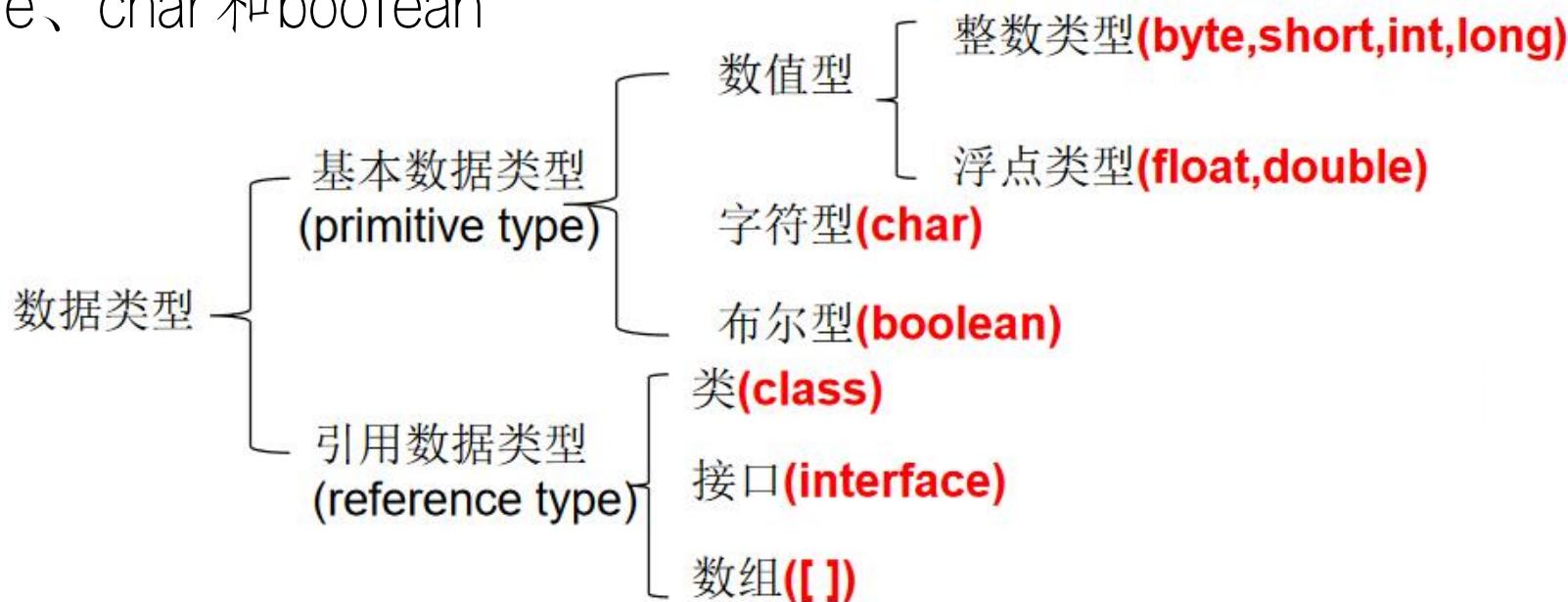
标识符必须以英文字母、下划线、美元符号\$开头，不能以数字开头。

Java标识符区分字母的大小写。



Java基本数据类型

- Java按照数据类型可以将变量分为两类：基本数据类型和引用数据类型
- 其中基本数据类型一共有八种：byte、short、int、long、float、double、char和boolean



Question:什么是强制类型转换，可能会带来什么问题？



Java控制流程

- 条件语句
 - If条件语句
 - If...else语句
 - Switch多分支语句
- 循环语句
 - while循环语句
 - do...while循环语句
 - for循环语句
- 跳转语句
 - Break语句
 - Continue语句
 - Return语句



Java数组

□ 声明数组

有两种声明数组的方式，第一种是Java语言规范提倡的方式，第二种是C语言风格声明方式。

```
int[] a;  
int b[];
```

□ 创建数组

有两种创建数组的方式，第一种指定数组的**长度**，第二种直接给数组赋值。数组一经创建，其长度就无法再更改。

```
int[] c = new int[2];  
int[] d = new int[]{0,1};
```



Java数组

□ 访问数组

数组的下标从0开始一直到长度-1。数组是一种随机访问的数据结构。

```
int[] a = new int[2];
for(int i = 0; i < 2; i++){
    a[i] = i;
    System.out.println(a[i]);
}
```

□ 数组长度

length属性用于访问一个数组的长度

```
int[] a = new int[2];
System.out.println(a.length);
```

□ 数组越界

数组访问下标范围是0到长度-1，一旦超过这个范围,就会产生数组下标越界异常

```
int[] a = new int[2];
System.out.println(a[2]);
```



Java.Lang.ArrayIndexOutOfBoundsException: 2



课程导航

- 对象与类
- 类的声明与构造
- 类的访问域
- static修饰符
- 数组

面向过程与面向对象

□ 四大发明之印刷术

- 雕版印刷术 VS. 活字印刷术
- 操作：改字，加字，多页

思想的突破：低耦合，可拓展，可复用



雕版印刷术



活字印刷术



面向过程与面向对象

□ 面向过程（Procedure Oriented）

- 功能模块化，代码流程化

□ 面向对象（Object Oriented）

- 按人们认识客观世界的系统思维方式
- 以类与对象为中心



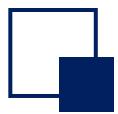
面向过程与面向对象

□ 面向过程（Procedure Oriented）

- 李雷同学：入学登记->计算机学院报到->选课->上课
- 韩梅梅同学：入学登记->理学院报到->选课->上课

□ 面向对象（Object Oriented）

- 类：学生，学院，课程
- 对象：李雷，韩梅梅；计算机学院，理学院



面向过程与面向对象

□ 面向对象三大特性

- **封装(Encapsulation)**

- 隐藏对象的属性和实现细节，仅对外公开访问方法；
- 增强安全性和简化编程

- **继承(Inheritance)**

- 子类继承父类的特征和行为
- 实现代码的复用

- **多态(Polymorphism)**

- 同一个行为具有多个不同表现形态的能力（“一个接口，多个方法”）
- 提高了程序的扩展性和可维护性

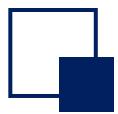
学生

- 私有信息和操作
- 公开操作

• 学生--计算机学院学生

• 上课：

- 计算机学院-编译原理
- 理学院-物理学



面向过程与面向对象

□ 面向过程（Procedure Oriented）

- 优点：简单逻辑下快速开发、计算效率高
- 缺点：灵活性差、无法适用复杂情况

简单场景
高性能计算

□ 面向对象（Object Oriented）

- 优点：低耦合、易扩展、易复用
- 缺点：性能相对低

复杂大型的软件



对象

□ 对象(**Object**)：客观存在的**具体实体**，具有明确定义的**状态**和**行为**

□ 特性：**标识符**（区别其他对象）、**属性**（状态）和**操作**（行为）

- **属性**：与对象关联的**变量**，描述对象**静态**特性；

- **操作**：与对象关联的**函数**，描述对象**动态**特性。

□ **示例**

- **学生**：李雷

- **属性**：姓名，性别，专业；专业 = “计算机”

- **操作**：入学，选课

对象是具体、有意义的个体



类

□ **类(Class)**: 对现实生活中一类具有**共同属性**和**共同操作**的对象的**抽象**

□ 举例

- **类名**: 学生
 - 有**属性**: 姓名, 性别, 专业;
 - 有**操作**: 被录取, 选课

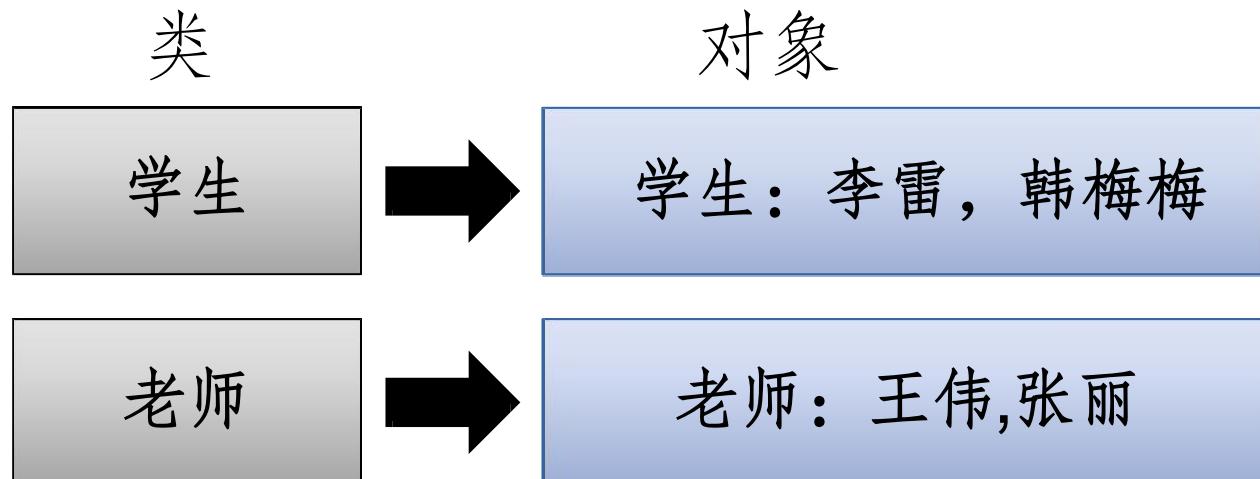
Student	—	类名 (必须)
name gender major	—	属性 (可选)
enrol() take_course()	—	操作方法 (可选)

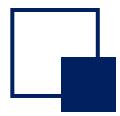


类与对象的对比

□ 类与对象之间的关系

- 类是对象的抽象，是创建对象的模板（代表了同一批对象的共性与特征）
- 对象是类的具体实例（不同对象之间还存在着差异）
- 同一个类可以定义多个对象（一对多关系）





类与对象的对比

□ 类与对象的比较

- 类是静态的，类的存在、语义和关系在程序设计时（执行前）就已经定义好了。
- 对象是动态的，对象在程序执行时可以被创建，修改，删除。

□ 在面向对象的系统分析和设计中，并不需要逐个对对象进行说明，而是着重描述代表一批对象共性的类

现实问题空间

- 物质
- 认识

面向对象空间

- 对象（客观存在，具体）
- 类（抽象概念）



课程导航

- 对象与类
- 类的声明与构造
- 类的访问域
- static修饰符
- 数组

类的声明

□ 格式

```
[类修饰符] class 类名  
{  
    成员变量的声明; // 描述属性  
    成员方法的声明; // 描述功能  
}
```

类修饰符
• public: 公共类
• abstract: 抽象类(继承)
• final: 最终类(非继承)

□ 举例

```
public class Person {  
    String name;  
    int age;  
  
    setName(String name) {...}  
    getName() {...}  
}
```



类的声明

□ 成员变量

- 作用：表示类和对象的属性、状态，在整个类中有效；
- 类型：可以是基本类型和引用类型。

□ 成员方法

- 实质：是实现某一功能的程序段，可以改变成员变量的属性和状态。

```
访问控制符 返回值类型 方法名([参数类型 参数,...]) {  
    // 方法体  
}
```

访问控制符

- **public**: 可被公开访问
- **private**: 私有
- **protected**: 受保护
- **default**: 默认



类的声明

```
public class Person {  
    private String name;  
    private int age;  
  
    public String getName() {  
        return name;  
    }  
}
```

面向对象特性-封装

- 隐藏 name, age
- 仅公开方法 getName()



封装性

- 封装性属于面向对象的第一大特性
- 类内部定义的属性和方法，类的外部不能调用。
 - 如果希望属性或方法不希望被外部所访问的话，则可以使用**private**关键字声明。

```
public class Person {    // 定义类
    String name;    // 不使用封装
    int age;        // 不使用封装
    public void tell() { // 表示一个功能
        System.out.println("姓名：" + name +
            ", 年龄：" + age);
    }
}
```

```
Person per = new Person();
per.name = "张三";
per.age = -30;
per.tell();
```

运行结果：

```
姓名：张三,年龄：-30
```



封装性

```
public class Person {  
    private String name;  
    private int age;  
    public void tell() {  
        System.out.println("姓名：" +  
name + "，年龄：" + age);  
    }  
}
```

```
Person per = new Person();  
per.name = "张三" ;  
per.age = -30;  
per.tell();
```

编译器报错：

```
name has private access in Person  
per.name = " 张三 " ;  
^
```

```
age has private access in Person  
per.age = -30 ;  
^
```

- **name**和**age**两个属性在**Person**中属于私有的访问，所以外部无法直接调用。



封装性

□ 通过getter/setter对属性进行访问

- **private**类型的属性或者方法只能在**本类**中使用
- 需要给被封装的属性一个**设置值**和**取得值**的方法
- 在**Java**开发的标准规定中，只要是属性封装，设置和取得就要依靠**setter**和**getter**方法完成操作

□ 示例

- **Setter**: `public void setName(String n) {}` 设置的时候可以进行检查
- **Getter**: `public String getName() {}` 取得时候只是进行简单的返回



封装性

```
public class Person {  
    private String name;  
    private int age;  
    public void setName(String n) {  
        name = n;  
    }  
    public void setAge(int a) {  
        // 合法年齡  
        if (a >= 0 && a <= 150) {  
            age = a;  
        }  
    }  
}
```

```
public String getName() {  
    return name;  
}  
public int getAge() {  
    return age;  
}  
public void tell() {  
    System.out.println("姓名：" + name  
+ "，年齡：" + age);  
}
```



封装性

编写测试类

```
public static void main(String args[]) {  
    Person per = new Person();  
    per.setName("张三");  
    per.setAge(30);  
    per.tell();  
}
```

运行结果：

```
姓名：张三,年龄：30
```

- 这种写法变成了Java中的一个标准：只要是属性就必须进行封装，封装之后的属性必须通过**setter**和**getter**设置和取得。



封装性

□ 封装的优点

- 安全性：数据和数据相关的操作被包装成对象，可以控制数据的访问权限。
- 高内聚：一种对象只做好一件（或者一类相关的）事情，对象内部的细节外部不关心也看不到。便于修改内部代码，提高可维护性。
- 低耦合：不同种类的对象间相互的依赖尽可能地降低。简化外部调用，便于调用者使用，便于扩展和协作。
- 可复用性：面向对象编程的主要目的是方便程序员组织和管理代码，快速梳理编程思路，带来编程思想上的革新。



类的构造方法

□ 构造方法(**constructor**)是一种特殊的方法

- 用来初始化(**new**)该类的一个新的**对象**
- 构造方法和**类名同名**，而且不写返回数据类型。

```
public class Person {  
    private String name;  
    private int age;  
    //...  
    Person( String n, int a ) {  
        name = n;  
        age = a;  
    }  
}  
  
Person Li = new Person("Li Lei",19);  
Person Han= new Person("Han Meimei"); // error  
Person Han= new Person("Han Meimei", 18);
```



类的构造方法

```
public class Person {  
    private String name;  
    private int age;  
    public Person(String n, int a) {  
        name = n;  
        age = a;  
    }  
}
```

- 从String n, int a定义不能体现n和a的含义
- 根据Java的编程规范，变量的名字应该采用**有意义的单词**



类的构造方法

- 为了让变量名体现出功能，修改Person类的构造函数如下：

```
public Person(String name, int age) {  
    name = name;  
    age = age;  
}
```

- 构造函数的两个参数能够表达出含义来了。但是输入参数和Person类定义的私有变量名称一致，导致name=name，age=age。无法区分哪个是私有变量，哪个是输入变量。



类的构造方法

□ this关键字

```
public class Person {  
    private String name;  
    private int age;  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
    // 编写setter、getter  
    ...  
}
```

- `this.name = name`, 该name是类中`private String name`语句定义的私有变量
- `this.name = name`, 该name是构造方法中的参数
- 通过构造方法中`this.name = name`语句对私有变量name进行了赋值

类的构造方法

□ 构造方法

- 类都有一个至多个构造方法
- 如果没有定义任何构造方法，系统会自动产生一个构造方法，称为默认构造方法

```
// 默认构造函数
class Person {
    public Person() {
    }
}
```

```
class Person {
    private String name;
    private int age;
    // ...
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

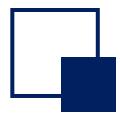
    Person(String name) {
        this.name = name;
        this.age = 18; // 默认入学年龄为18
    }
}

Person Li = new Person("Li Lei",19);
Person Han= new Person("Han Meimei");
```



子类的构造方法

- 继承(**inheritance**)是面向对象的程序设计中最为重要的特征之一
- 继承的好处
 - 提高程序的抽象程度
 - 实现代码重用，提高开发效率和可维护性
- 子类(**subclass**)，父类或超类(**superclass**)
 - 子类继承父类的状态和行为
 - 可以[添加新的状态和行为](#)



子类的构造方法

- Java使用关键字来实现继承
- 子类不能直接**extends**继承父类的构造方法
- 子类使用**super**语句来继承父类的构造方法

```
class Person {  
    String name;  
    int age;  
    String getname( ... ) { ... }  
    public Person(String name, int age) {...}  
}  
  
class Student extends Person {  
    super(name,age);  
    String school;  
    String getschool( ... ){ ... }  
}
```

以下哪项不是封装的优点？

- A 数据的安全性得以提高
- B 类内部的结构可以自由修改
- C 代码的可维护性和复用性得以提高
- D 模块的耦合性变高

提交

以下哪项不是封装的优点？

- A 数据的安全性得以提高
- B 类内部的结构可以自由修改
- C 代码的可维护性和复用性得以提高
- D 模块的耦合性变高

提交



课程导航

- 对象与类
- 类的声明与构造
- 类的访问域
- static修饰符
- 数组



类的访问域

□ 成员的访问权限控制

- 定义：本类及本类内部的成员（成员变量、成员方法、内部类）对其他类的可见性，即这些内容是否允许其他类访问
- 类型： private、 default、 protected、 public

```
public class Person {  
}
```



类的访问域

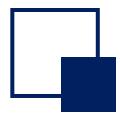
□ 类的访问权限控制

- **public**: 该类可以被其他类所访问
- ***default***: 该类只能被同一个包中的类访问
- **private**: 无法被其他类所访问
- **protected**: 可以被子类访问，以及子类的子类（作用于继承关系）

	private	<i>default</i>	protected	public
同一个类中	✓	✓	✓	✓
同一个包中		✓	✓	✓
子类中			✓	✓
全局范围内				✓

作用：
实现**封装**特性

包(package): 我们在java编程中经常把功能相似或者相关的类放在一个包里



类的访问域

成员的访问权限控制？

```
package p;  
public class Demo{  
    private int var1=1;  
    int var2 = 2;  
    protected int var3 = 3;  
    public int var4 = 4;  
    ...  
}
```

	var1	var2	var3	var4
package p1				
package p				
class newDemo1 extends Demo in p				
class newDemo2 extends Demo in p1				
class Demo





课程导航

- 对象与类
- 类的声明与构造
- 类的访问域
- **static**修饰符
- 数组



静态成员

□ 含义

- 表明该属性、该方法是属于类的，称为静态属性或静态方法
(无static修饰，则是实例属性或实例方法)

```
static 成员属性; // 静态属性
```

```
static 成员方法; // 静态方法
```

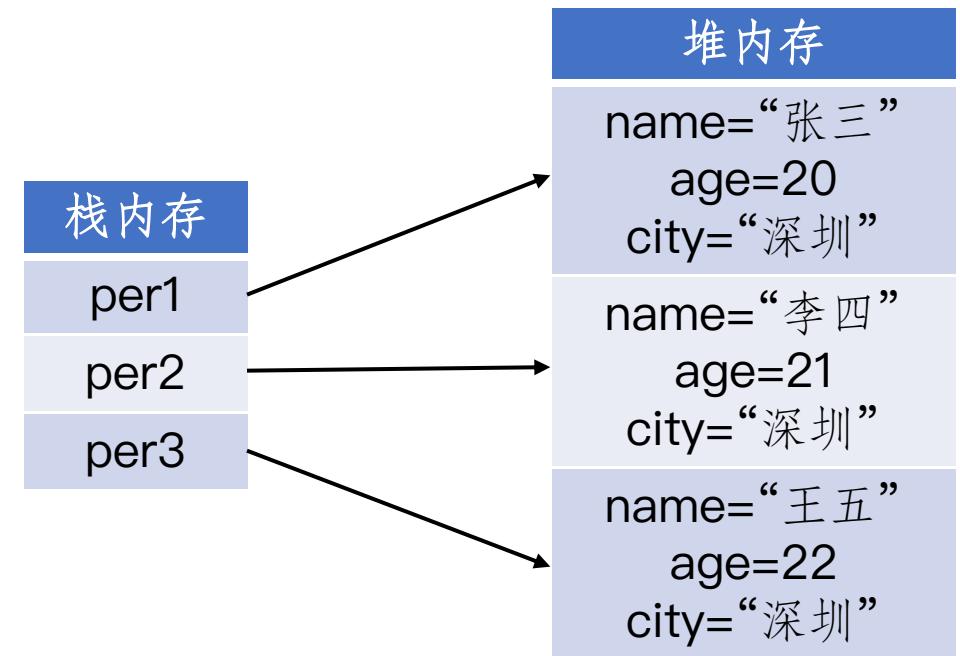
□ 说明

- 静态成员属于类所有，不属于某一具体对象私有；
- 静态成员随类加载时被静态地分配内存空间、方法的入口地址

静态成员

```
public class Person {  
    private String name;  
    private int age;  
    String city = "深圳";  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```

```
Person per1 = new Person("张三", 20);  
Person per2 = new Person("李四", 21);  
Person per3 = new Person("王五", 22);
```

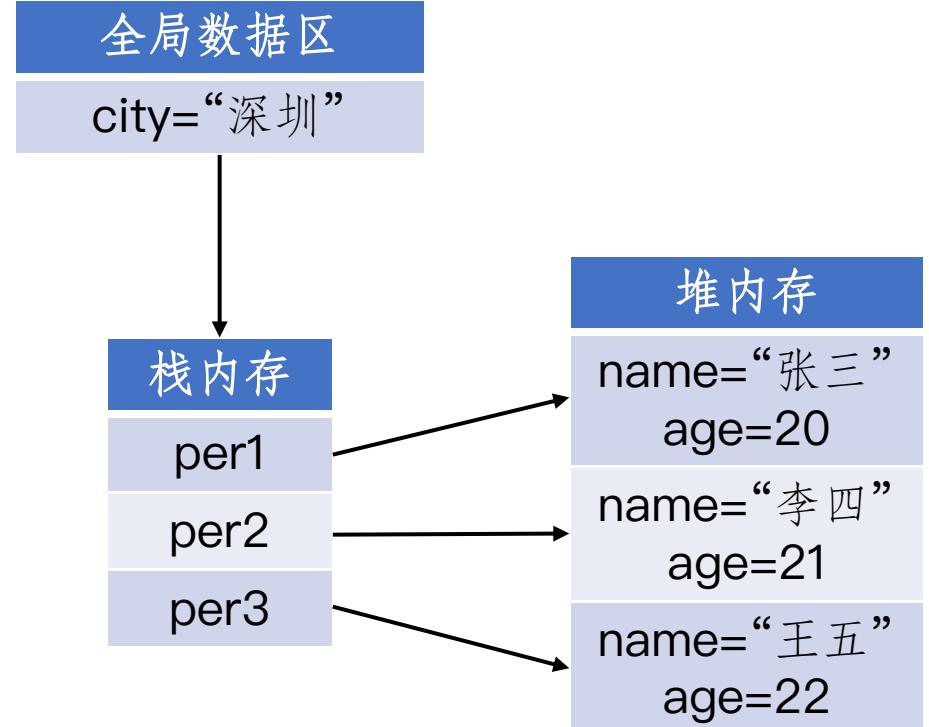


- 每个对象占用自己的city 属性，会造成内存空间的浪费
- 可以用 **static** 将city属性设置成一个公共属性。

静态成员

□ 用 static 声明属性

```
public class Person {  
    private String name;  
    private int age;  
    static String city = "深圳";  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
    // 调用  
    Person.city; //类名.属性
```



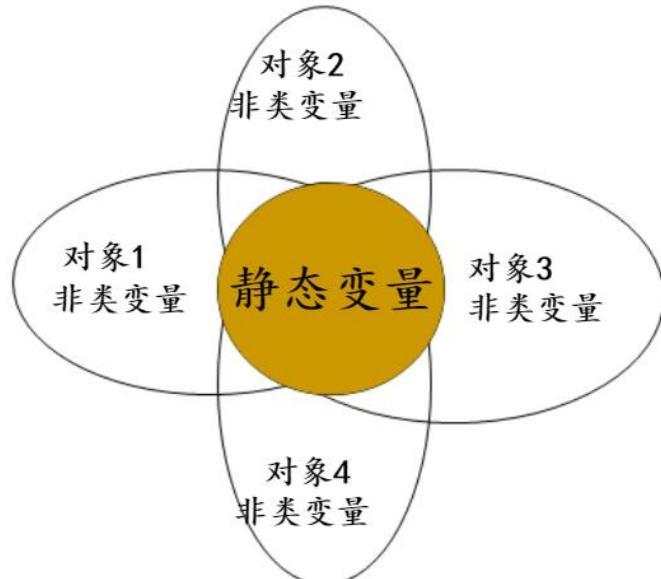
- 由于全局属性拥有可以通过类名称直接访问的特点，所以这种属性又称为类属性。



静态成员

□ 静态属性

- 是类的字段，不属于任何一个对象实例
- 静态属性被所有的对象所**共享**，在内存中**只有一个副本**，它当且仅当在**类初次加载时会被初始化**
- 非静态属性是对象所拥有的，在创建对象的时候被初始化，**存在多个副本**，各个对象拥有的副本互不影响



```
class Person {  
    static long totalNum; // 代表总人数，它与具体对象无关
```

```
    int age;  
    String Name;
```

```
}
```

```
Person Li= new Person("Li Lei", 19)
```

```
Person Han= new Person("Han Meimei", 18)
```

```
访问: Person.totalNum, Li.totalNum, Han.totalNum
```



静态成员

□ 用static声明方法

```
class Person {  
    private String name;  
    private int age;  
    private static String city = “深圳”;  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
    public static void setCity(String c) {  
        city = c;  
    }  
}  
  
// 调用  
Person.setCity(“哈尔滨”); //类名.方法
```



静态成员

□ 注意

- 使用**static**声明的方法，不能访问非**static**的操作（属性或方法）
- 非**static**声明的方法，可以访问**static**声明的属性或方法

□ 原因

- 如果一个类中的属性和方法都是非**static**类型的，一定要有实例化对象才可以调用
- **Static**声明的属性或方法可以通过类名访问，可以在没有实例化对象的情况下调用

静态成员

□ 静态方法

- 在非静态成员方法中是可以访问静态成员方法/属性的
- 在静态方法中不能访问类的非静态成员属性和成员方法

```
public class Demo
{
    private static String str1 = "hello";
    private String str2 = "world";
    public void print1( ) { }
    public static void print2( ) { }
}
```

```
public void print1( )
{
    System.out.println(str1);
    System.out.println(str2);
    print2();
}
```



```
public static void print2( )
{
    System.out.println(str1);
    System.out.println(str2);
    print1();
}
```





静态成员

□ 静态块

- 可以置于类中的任何地方，类中可以有多个**static**块
- 在类被加载的时候**执行且仅会被执行一次**,按照**static**块的**顺序**来执行每个**static**块
- 一般用来**初始化静态属性**和**调用静态方法**

```
class Person
{
    static string city;
    String name;
    public Person(String name) { ... }
    static //静态块
    {
        city="Shenzhen"
        System.out.println("run
                           static code block!");
    }
}
```

```
Person p1=new Person("aa" );
Person p2=new Person("bb" );
Person p3=new Person("cc" );
// 输出多少次 “run static code block!” ?
```

不管构建多少次**Person**类的对象实例，**static{}**都会被**执行且只执行一次**。

所以，只会输出一次**run static code block!**



静态成员

□ 思考： static是否破坏面向对象的特性？

- 属于类，而非具体对象
- 初始化加载到内存，被所有对象所共享（一定程度上的全局属性）
- 保持类的封装性



课程导航

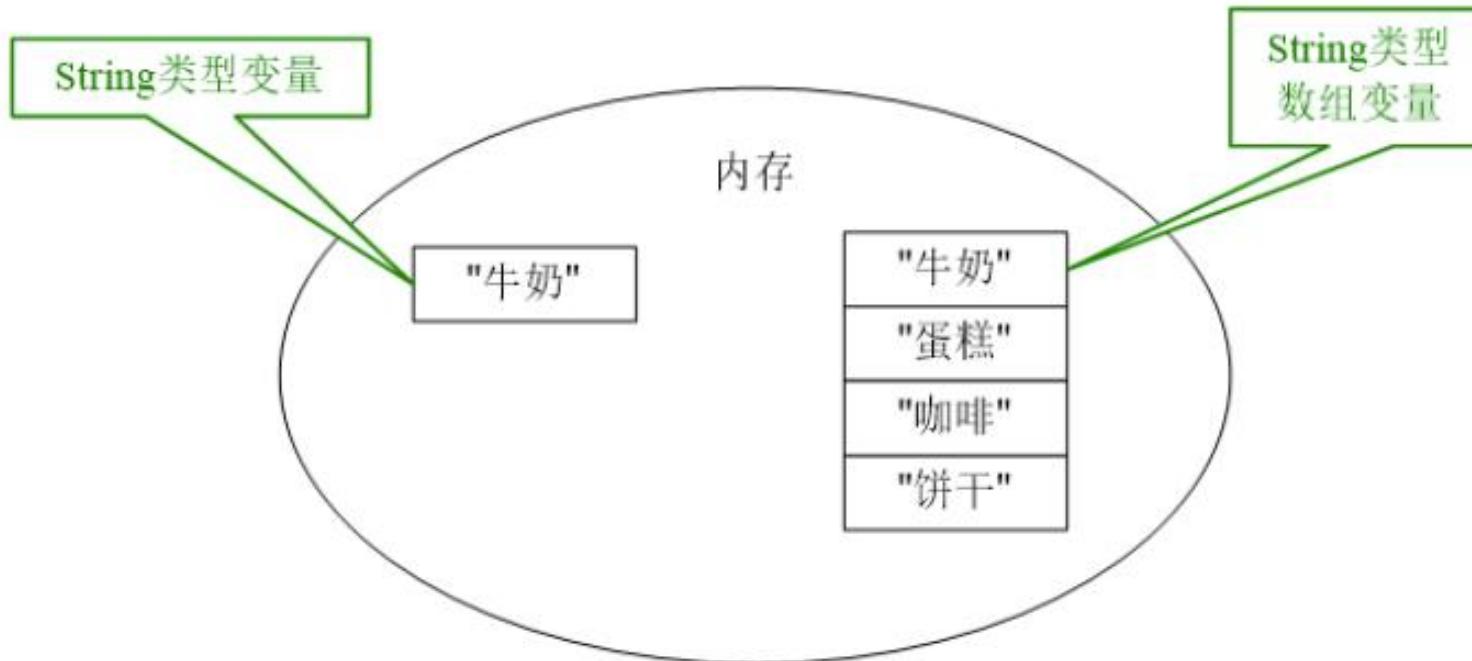
- 对象与类
- 类的声明与构造
- 类的访问域
- static修饰符
- 数组



数组

□ 定义

- 数组是一个变量，存储相同数据类型的一组数据

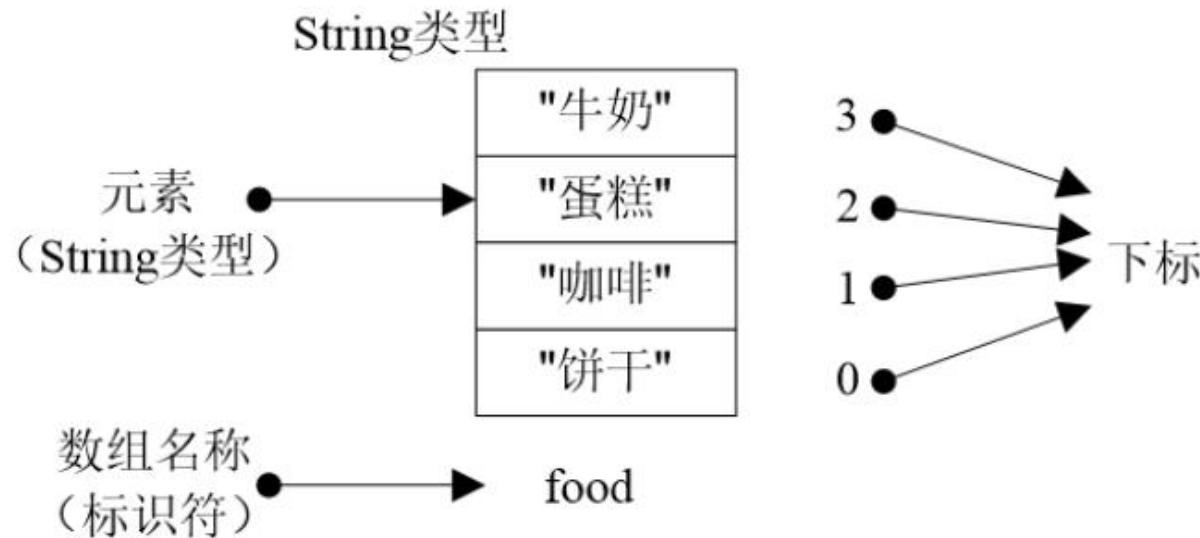


- 声明一个变量就是在内存空间划出一块合适的空间
- 声明一个数组就是在内存空间划出一串连续的空间



数组

- 数组只有一个名称，即**标识符**
- 元素**下标**标明了元素在数组中的位置，**从0开始**
- 数组中的每个元素都可以**通过下标来访问**
- 数组**长度固定不变**，避免数组越界





数组

□ 数组的使用：初始化和赋值

- 初始化：告诉计算机分配几个连续的空间
 - 动态初始化：数组声明且为数组元素分配空间与赋值的操作分开进行
 - 静态初始化：在定义数组的同时就为数组元素分配空间并赋值

```
int[] arr = new int[3];  
arr[0] = 3;  
arr[1] = 9;  
arr[2] = 8;
```

```
int[] arr = new int[] {3, 9, 8};  
int[] arr = {3, 9, 8};
```

- 赋值：向分配的格子里放数据



数组

□ 数组错误使用

```
Public class ErrorDemo1{  
    public static void main(String[] args){  
        int[] score = new int[];  
        score[0] = 89;  
        score[1] = 63;  
        System.out.println(score[0]);  
    }  
}
```

编译出错，没
有写明数组的
大小



数组

□ 数组错误使用

```
Public class ErrorDemo2{  
    public static void main(String[] args){  
        int[] score = new int[2];  
        score[0] = 90;  
        score[1] = 85;  
        score[2] = 65; // Red box highlights this line  
        System.out.println(score[2]);  
    }  
}
```

编译出错，数
组越界

□ 数组错误使用

```
Public class ErrorDemo3{  
    public static void main(String[] args){  
        int[] score1 = new int[5];  
        score1 = {60, 80, 90, 70, 85};  
        int[] score2;  
        score2 = {60, 80, 90, 70, 85};  
    }  
}
```

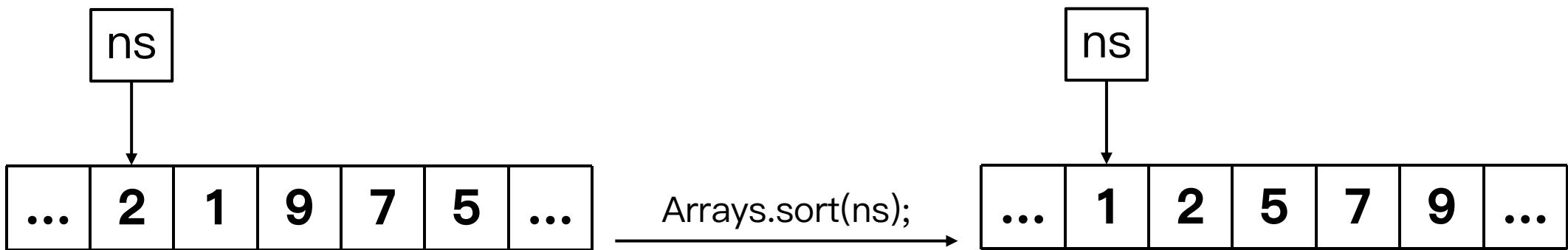
编译出错，创建
数组并赋值的方
式必须在一条
语句中完成

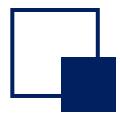


数组

□ 排序

```
int[] ns = { 2, 1, 9, 7, 5};  
Arrays.sort(ns);  
System.out.println(Arrays.toString(ns));
```





□ 多维数组的使用

- 一维数组是几何中的线性图形，二维数组是表格
- 含义：一维数组又作为另一个一维数组的元素而存在
- 从数组底层的运行机制来看，没有多维数组



数组

□ 多维数组的使用：初始化

- 动态初始化：`int[][] arr = new int[m][n];`
 - 二维数组中有m个一维数组，每个一维数组中有n个元素
- 动态初始化：`int[][] arr = new int[m][];`
 - 二维数组中有m个一维数组，每个一维数组都是默认初始化值null
 - 可以对这个三个一维数组分别进行初始化
 - `arr[0] = new int[3]; arr[1]= new int[1]; arr[2] = new int[2];`
 - `int[][]arr = new int[][3]` // 非法！



数组

□ 多维数组的使用：初始化

- 静态初始化：

```
int[][] arr = new int[][]{{1,2,3},{2,7},{4,5,6,7}};
```

➤ 定义一个名称为arr的二维数组，二维数组中有三个一维数组

➤ arr[0] = {1,2,3}; arr[1] = {2,7}; arr[2] = {4,5,6,7};

➤ 特殊写法情况：int[] x, y[]; x是一维数组，y是二维数组

➤ Java中多维数组不必都是规则矩阵形式



数组

□ 对象数组

- 数组中的元素是对象，数组中的每一个元素都是对一个对象的引用

```
Person[] students;  
Person students[];//不推荐
```

```
class Person {  
    private String name;  
    private int age;  
    // ...  
    Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
    // ...  
}
```



数组

□ 对象数组静态初始化

- 在定义数组的同时对数组元素进行初始化

```
Person[] students = {  
    new Person("张三", 21),  
    new Person("李四", 19)  
}
```



数组

□ 对象数组动态初始化

- 使用运算符new为数组分配空间

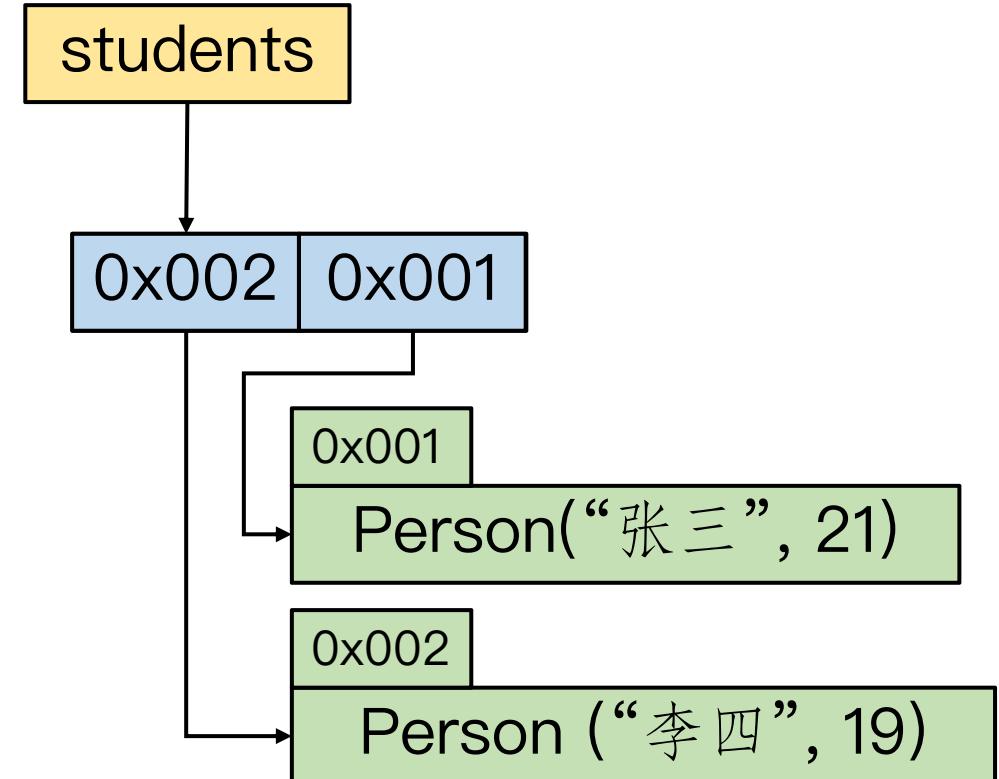
```
Person[] students = new Person[2];
Person per1 = new Person("张三", 21);
Person per2 = new Person("李四", 19);
students[0] = per1;
students[1] = per2;
```



数组

□ 对象数组动态初始化

```
Person[] students = new Person[2];  
Person per1 = new Person("张三", 21);  
Person per2 = new Person("李四", 19);  
students[0] = per2;  
students[1] = per1;
```





课程导航

- 对象与类
- 类的声明与构造
- 类的访问域
- static修饰符
- 数组