

## 2.4 Java 其他面向对象语法

---

- 抽象类
- this 与 super
- 引用
- 枚举
- 内部类



# 目录

1

抽象类

2

this 与 super

3

引用

4

枚举

5

内部类

# 抽象类

- Java 中的**抽象类**[抽象クラス]是一种特殊的类：和接口类似，它可以包含一些未被实现的方法（**抽象方法**），并且无法被实例化（创建对象）；但抽象类也可以包含成员变量和已被实现的方法。
- 要定义抽象类，使用 **abstract** 修饰符：

```
public abstract class Animal { }
```

- 对于成员变量和已实现的方法，用和普通类一样的方式定义。对于未实现的方法，同样使用 **abstract** 修饰符：

```
public abstract void eat(String food);
```

# 抽象类的使用

- 和接口一样，抽象类本身无法被实例化。我们需要先编写子类继承抽象类，并实现抽象方法：

```
1 public class Cat extends Animal {  
2     @Override  
3     public void eat(String food) {  
4         System.out.print(getName() + " ate " + food + ", ");  
5         meow();  
6     }  
7 }
```

- 简单来说，对于不会直接创建实例的类，都可以使用抽象类来编写，防止意外被实例化。





# 接口和抽象类的继承

- 接口可以继承任意多个接口：

```
public interface InterfaceA extends InterfaceB, InterfaceC { }
```

实装 InterfaceA 的类也须实装 InterfaceB 和 InterfaceC 里的方法。

- 如果一个抽象类实装了某些接口或 / 和继承了某个抽象类，它可以不实现接口和父类里未实现的方法，交给子类解决：

```
public abstract class Animal implements Runnable { } // 在 Animal 里不会报错
```

- 和接口不同，抽象类基本还是一个类，所以不能被多重继承。

Q & A

*Question and answer*

# 目录

1

抽象类

2

this 与 super

3

引用

4

枚举

5

内部类



# this 关键字

- Java 中的 **this** 关键字有两种含义：指代当前**对象**；或在构造方法中指代本类的**构造方法**。
- 之前说过，如果直接使用本类的成员变量或方法，实际使用的就是当前对象的变量或方法。那么 this 关键字有什么用呢？
- 可以使用 this 关键字消除**成员变量**和**局部变量**的歧义：

```
1 private void setName(String name) {  
2     this.name = name;  
3 }
```

- 想一想，如果不加 this 会怎样？



# this 指代构造方法

- **this** 也可用于指代本类的构造方法：

```
public Animal(String name) {  
    this.name = name;  
}  
  
public Animal() {  
    this("Unnamed Cat");  
}
```

- 这样就可以和普通方法一样方便的设定默认参数。

# super 关键字

- **super** 关键字同样有两种含义：指代父类的**对象**；指代父类的**构造方法**。
- 第一种含义主要用于调用父类中的同名方法，比如重写的方法：

```
1 public void eat(String food) {  
2     super.eat(food);  
3     meow( );  
4 }
```

- 第二种用法我们之前 (👉 § 2.2.2) 已经学习过了。
- 利用 **this** 和 **super**，你将怎么改进 **animals** 包中的代码？



Q & A

*Question and answer*

# 目录

1 抽象类

2 this 与 super


3 引用

4 枚举

5 内部类



# 回顾：基本类型和引用类型

- 我们说过（ §1.2.1），Java 中的类型分为**原始类型（基本类型）**和**非原始类型（引用类型）**。我们来复习一下它们的区别：
  1. 基本类型存储一些简单数据；引用类型存储一些复杂数据。
  2. 基本类型的数据不是对象，因此不能使用方法；引用类型的数据是对象，可以使用它定义的方法。
  3. 基本类型没有构造方法。
  4. 基本类型不能被继承。
- 今天我们介绍两者之间的一个重要区别：储存形式的不同。

# 基本类型的赋值

## Example ✓

你认为以下代码会输出什么结果？  
试着编写并运行一下，与你的预期相符吗？

```
1 int a = 1;  
2 int b = 2;  
3 a = b;  
4 a = 3;  
5 System.out.println("a = " + a);  
6 System.out.println("b = " + b);
```



# 引用类型的赋值

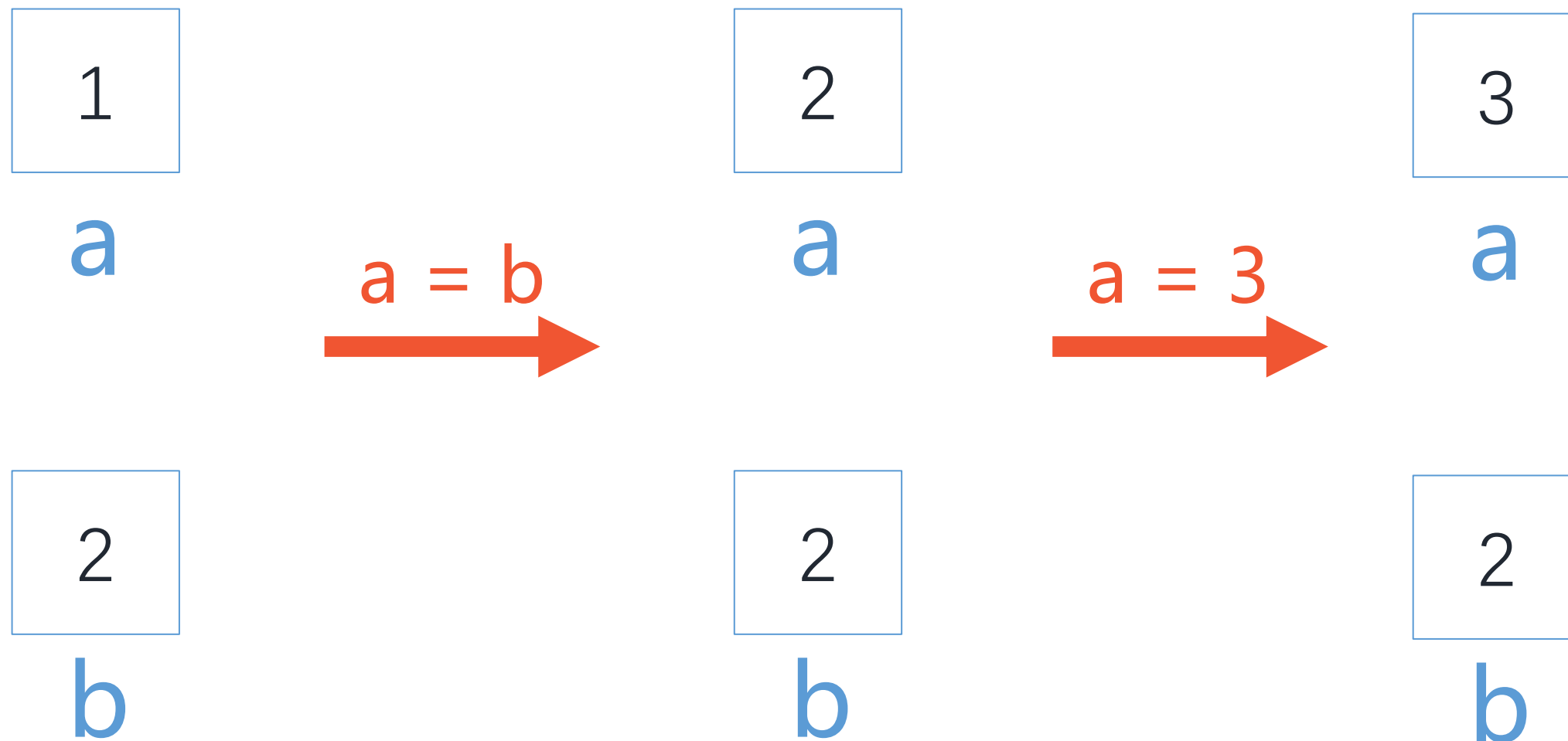
## Example ✓

你认为以下代码会输出什么结果？  
试着编写并运行一下，与你的预期相符吗？

```
1 int[] a = {1, 1};  
2 int[] b = {2, 2};  
3 a = b;  
4 a[0] = 3;  
5 System.out.println("a = " + a[0] + " " + a[1]);  
6 System.out.println("b = " + b[0] + " " + b[1]);
```

# 基本类型的存储方式

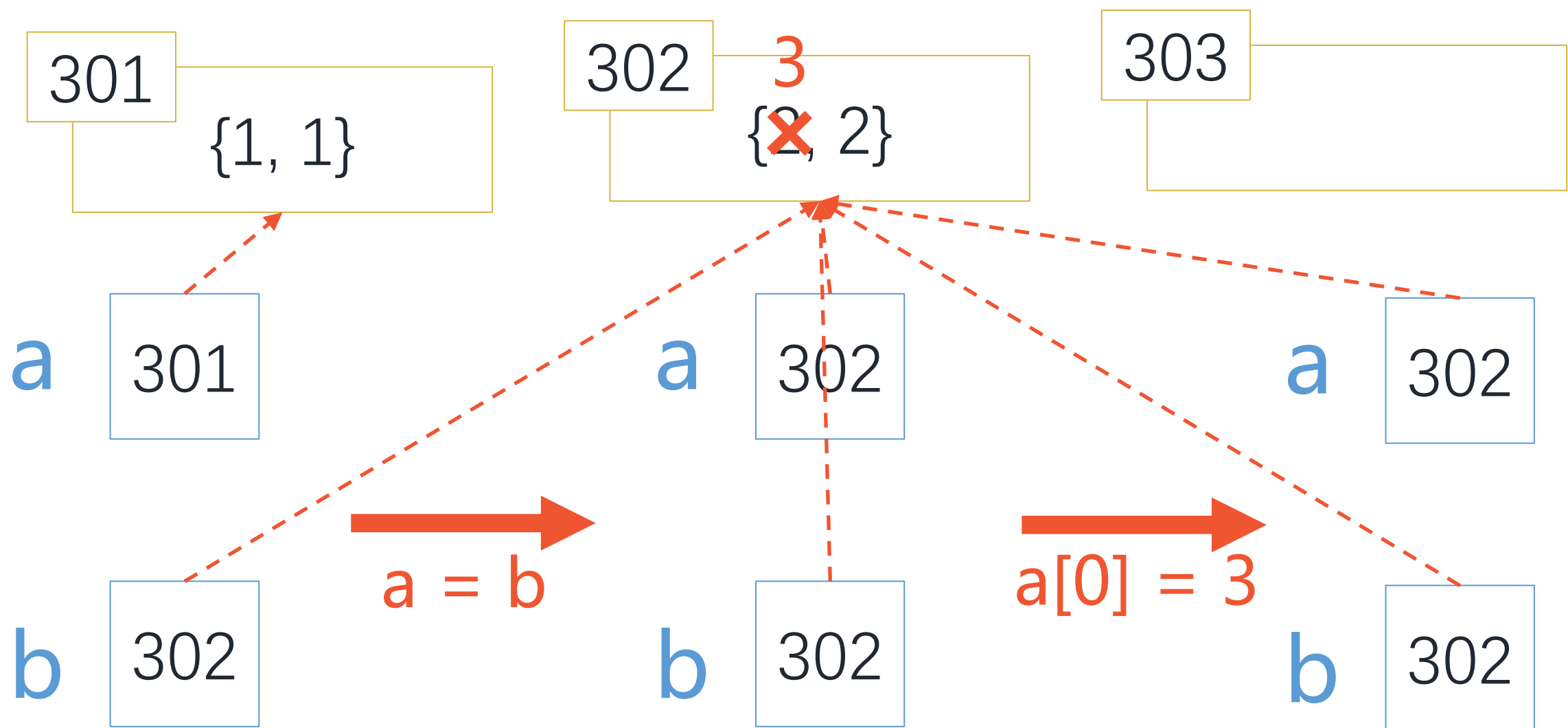
- 在 Java 中，基本类型的变量直接存储某一个基本类型的值。因此，赋值语句只是复制了变量中的值，并没有使两个变量本身产生联系：





# 引用类型的存储方式

- 然而，引用类型变量存储的则是对象的*引用*。简单来说，它存储了某个对象的“门牌号”：



# 自定义的引用类型

- 别的引用类型，比如自己写的类，保存在变量中的也是对象的引用。

## Example ✓

```
1 Cat a = new Cat("Alice");
2 Cat b = new Cat("Bob");
3 a = b;
4 a.setName("Charlie");
5 System.out.println("a = " + a.getName()); // => a = Charlie
6 System.out.println("b = " + b.getName()); // => b = Charlie
```

# 引用类型变量的使用

- 总结来说，引用类型变量间的赋值会导致变量间产生“联系”，它们会保存同一个对象。这种操作被称为“浅拷贝”（Shallow Copy）。
- 因此，在使用引用类型变量时应该谨慎使用赋值语句（“=”）。比如，如果想要复制一个数组，则不应该使用赋值语句。



# 数组的复制

- 正确的做法是新建一个数组，把之前数组的所有元素复制进新的数组（**深拷贝**，Deep Copy）：

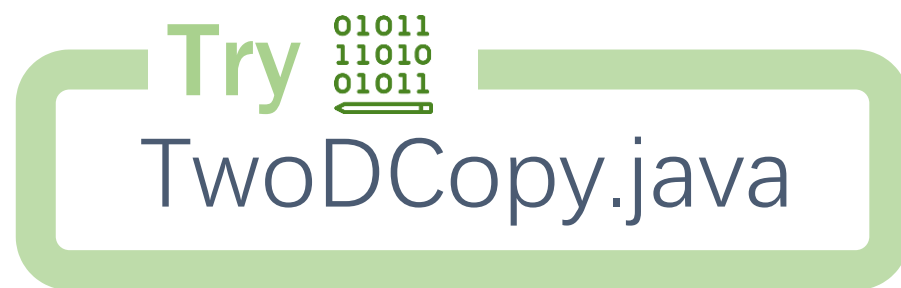
```
1 int a = {1, 1};
2 int b = {2, 2};
3 a = new int[2];
4 for (int i = 0; i < 2; i++) {
5     a[i] = b[i];
6 }
7 a[0] = 3;
8 System.out.println("a = " + a[0] + " " + a[1]); // => a = 3 2
9 System.out.println("b = " + b[0] + " " + b[1]); // => b = 2 2
```

- Java 也在 Array 类中直接提供了复制方法 **clone**:

```
a = b.clone();
```

## 2 维数组的复制

- 想一想，使用 clone 方法能正确复制一个 2 维数组吗？
- 如果不可以，为什么？你能写出正确复制 2 维数组的代码吗？



- 如果暂时还不理解，就请记住在复制时使用此代码。

# 相等运算符 “==”

- 我们知道，Java 中的相等运算符 “==” 可以判断两个变量是否相等。但当我们对引用类型使用相等运算时，须格外小心：我们比较的其实是对象的引用：

```
1 int[] a = {1, 2};  
2 int[] b = {1, 2};  
3 System.out.println(a == b); // => false
```

这个例子里，a 和 b 是两个不同的数组，只是正好装着同样的值而已，因此 “a == b” 是不正确的。



# 字符串的比较

- 非常容易出现的一个错误是在字符串比较时:

```
1 String a = "abc";  
2 String b = "ab";  
3 b += "c";  
4 System.out.println(a == b); // => false
```

- 好在 Java 给我们提供了 `equals` 方法进行判断:

```
1 String a = "abc";  
2 String b = "ab";  
3 b += "c";  
4 System.out.println(a.equals(b)); // => true
```

Note 

始终使用 `equals`  
方法比较字符串!

Q & A

*Question and answer*

# 目录

1

抽象类

2

this 与 super

3

引用

4

枚举

5

内部类



# 枚举

- 有时，我们需要用到某个只有有限个可能的取值的类型，比如状态（开机、关机或待机）、强度（弱、中或强）等。
- 我们当然可以使用一个整数变量来表达（比如开机为 0、关机为 1、休眠为 2），但这样不仅代码可读性不高，变量还有可能被设为一个范围外的值（比如被误设为 3）。
- 在 Java 中，我们可以定义**枚举**[列举]这种特殊的**类**。枚举类型只能取到我们提前定义好的几种值。

# 枚举的定义和使用

- 要定义枚举，使用 **enum** 关键字：

```
1 public enum Status {  
2     RUNNING, POWERED_DOWN, SLEEPING  
3 }
```

- 其中大括号 “{}” 里需要写上所有可能的取值，用逗号隔开。
- 接下来我们就可以像使用其他类一样使用枚举。注意 Status 型变量只能取上面写出的 3 个值：

```
Status pc1Status = Status.RUNNING;  
Status pc2Status = Status.POWERED_DOWN;
```

- 可以注意到，枚举型的值都是以使用 **静态变量** 的形式使用的。

**Note**



枚举的取值应为  
大写**蛇形**命名。

# 枚举和 switch 语句

- 枚举还有一个方便的用法是在 **switch** 语句 (← § 1.3.1) 中:

```
1 switch (pc1Status) {  
2     case RUNNING:  
3         System.out.println("PC1 is running.");  
4         break;  
5     case POWERED_DOWN:  
6         System.out.println("PC1 is not running.");  
7         break;  
8     case SLEEPING:  
9         System.out.println("PC1 is sleeping.");  
10        break;  
11 }
```



# 枚举的成员变量和方法

- 枚举也是一个类，因此他也可以有成员变量和方法（包括构造方法）。如果枚举有构造方法，你需要在定义每一个值时调用：

```
1 public enum Status {  
2     RUNNING(1.0f), POWERED_DOWN(0.0f), SLEEPING(0.2f); // 注意结尾处的分号  
3  
4     private float power;  
5  
6     Status(float power) {  
7         this.power = power;  
8     }  
9  
10    public float getPower() {  
11        return power;  
12    }  
13 }
```

- 枚举的构造方法应为 **private** 的（可省略）。

Try   
enums 包

# 目录

1 抽象类

2 this 与 super

3 引用

4 枚举

5 内部类

# 内部类

- 在 Java 中，你可以在一个类内部定义其他类。这被称为类的嵌套[入れ子]或内部类[内部クラス]：

```
1 public class Outer {  
2     class Inner { }  
3 }
```

- 一个常见的用途是一些小的工具类整理到使用它的类里。比如如果只有汽车类使用到了轮胎类，你可以把轮胎类写成汽车类的内部类。这种情况下，你甚至可以把轮胎类设为 private 的：

```
1 public class Car {  
2     private class Wheel { }  
3 }
```



# 静态和非静态内部类

- 和成员变量及方法一样，内部类也可以是**静态或非静态**的。静态内部类可以直接被实例化；而非静态的内部类必须由一个外部类的对象实例化——实例化的对象将**依存于**这个外部类对象。
- 通常，静态内部类可以用来实现一些**简单的**或 / 和**通用的**工具类；非静态的内部类用于实现一些**依存于**外部类的类型。
- 一个很常见的用例是把类中用到的**枚举**写成内部类。



# 静态内部类

- 静态内部类在功能上和一般的类没有什么不同，使用它的主要目的就是要把不同类整理到一起。使用 **static** 关键字定义：

```
1 public class Car {  
2     Wheel[] wheels;  
3  
4     private static class Wheel { }  
5 }
```

- 外部类以外的其他类使用时，需要使用外部类名 + 内部类名，以“.”隔开：

```
Car.Wheel wheel = new Car.Wheel();
```

# 非静态内部类

- 非静态内部类一般用于定义明确**依存于**外部类的类，比如司机永远能启动自己的车：

```
1 public class Car {  
2     private Driver Driver;  
3  
4     private class Driver { }  
5 }
```

- 如果其他类要实例化这样一个内部类，需要使用一个外部类对象 + **new**，以“.” 隔开：

```
1 Car car = new Car();  
2 Car.Driver driver = car.new Driver();
```

# 非静态内部类

- 由于非静态内部类一定**依存于**某个外部类对象， 它可以直接使用该对象的成员变量或 / 和方法：

```
1 public class Car {  
2     private Driver driver;  
3     private String brand;  
4  
5     private class Driver {  
6         public String getCarBrand() {  
7             return brand;  
8         }  
9     }  
10 }
```



# 内部类：总结

## Sum Up

内部类的语法和使用方式都很复杂，但现阶段只需要记住两件事：

1. 内部类名称的表达方法，即 Outer.Inner 的格式。因为之后可能会使用到别人写好的内部类。
2. 非静态内部类可以访问外部类的成员变量或方法。因为之后我们会用到特殊的非静态内部类（匿名内部类，[↪ § 3.2.2](#)）。



Q & A

*Question and answer*

# 总结

## Sum Up

1. 抽象类的定义和使用。
2. this 和 super 关键字：指代**对象**和**构造器**的用法。
3. 引用：
  - ① 数组的复制：浅拷贝与深拷贝。
  - ② 字符串的 **等于**比较。
4. 枚举的定义和使用方法。
5. 内部类的基本使用方法。



**THANK YOU!**