# 【OOP：面向对象编程\_第四天】

## 主要内容

1. final
2. 接口
3. 内部类
4. 虚拟机和垃圾回收

## 学习目标

|  |  |
| --- | --- |
| 知识点 | 要求 |
| final | 掌握 |
| 接口 | 掌握 |
| JDK1.8接口新特征 | 掌握 |
| 接口应用：内部比较器Comparable | 掌握 |
| 成员内部类 | 了解 |
| 静态内部类 | 了解 |
| 方法内部类 | 了解 |
| 匿名内部类 | 了解 |
| 虚拟机 | 理解 |
| 垃圾回收 | 理解 |

## 第一节 final

### 1.1 final

**final关键字的作用：**

* **修饰变量:** 被他修饰的变量不可改变。一旦赋了初值，就不能被重新赋值。

final int MAX\_SPEED = 120;

* **修饰方法：**该方法不可被子类重写。但是可以被重载！

final void study(){}

* **修饰类:** 修饰的类不能被继承。比如：Math、String、System等。

final class A {}

#### 【示例1】模拟实现Math类

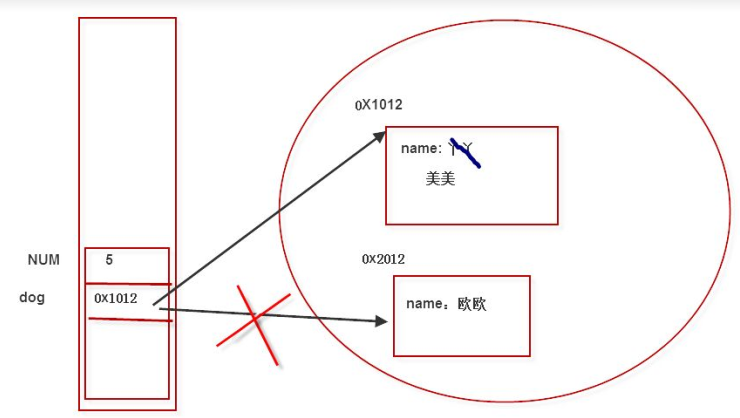
|  |
| --- |
| **package** com.bjsxt.finalDemo;  **public class** Test1 {  **public static void** main(String[] args) {  */\*  final修饰局部变量,常量,只能赋值一次  \*/* **final int** a=10;  System.***out***.println(MyMath.***PI***);  }  }  */\* \* final 修饰的类叫做最终类,不能被继承 \* \*/* **final class** MyMath{  */\*  final修饰成员变量  final和static修饰的成员变量叫做静态常量  一般静态常量属性名所有的字母都大写  \*/* **public final static double *PI***=3.1415926;   */\*  \* final修饰 的方法最终方法,不能被子类重写的方法  \* \*/* **public final int** getMax(**int** a,**int** b){  **return** a>b?a:b;  }  } */\*class MyMath2 extends MyMath{  public final int getMax(int a,int b){  return a>b?a:b;  } }\*/* |

注意：

* 注意：final不能修饰构造方法
* final修饰基本数据类型,值只能赋值一次，后续不能再赋值
* final修饰引用数据类型，final Dog dog = new Dog("亚亚");，不能变化的引用变量的值，可以变化的是对象的属性

#### 【示例2】final关键字修饰引用变量

|  |
| --- |
|  |



### 本节作业

1. final关键字的作用
2. 抽象类的作用
3. 抽象类和抽象方法的关键技能点

## 第二节 接口

接口(interface)就是规范，定义的是一组规则，体现了现实世界中“如果你是…则必须能…”的思想。如果你是天使，则必须能飞。如果你是汽车，则必须能跑。接口的本质是契约，就像我们人间的法律一样。制定好后大家都遵守。

### 2.1 接口

**声明格式：**

|  |
| --- |
| [访问修饰符] **interface** 接口名 [extends 父接口1，父接口2…] {  常量定义；  方法定义；  } |

|  |
| --- |
| */\* \* 1接口中的方法都是公有的抽象方法 \* 2接口中的成员变量都用公有的静态常量 \* 3接口不能被实例化 \* 4接口不能被普通的JAVA类继承,只能被普通的JAVA类实现 \* 5一个JAVA类只能有一个直接的父类 单继承 一个JAVA类可以同时实现多个接口(解决单继承) \* 6可以实例化接口的实现类声明成接口 \* 7接口作为方法参数,其实现类对象就可以作为实参传入 \* 8接口作为返回值,其实现类对象就可以作为结果返回 \* 9只有接口才能继承接口,一个接口还可以同时继承多个接口 \* 10实现类要实现所有层级接口中定义的抽象方法*  *11*  *必须先extends 再implements \* \*/* **public interface** MyInter1 {   **public final static int *A***=10;   **public abstract void** methodA();  } **interface** MyInter2 {   **public final static int *B***=10;   **public abstract void** methodB();  }  **interface** MyInter3 **extends** MyInter2,MyInter1{  **public abstract void** methodC(); }  **class** A{   } **class** B{   }  **class** MyImpl2 **implements** MyInter3 {   @Override  **public void** methodA() {    }   @Override  **public void** methodB() {   }   @Override  **public void** methodC() {   } }  **class** MyImpl **extends** A **implements** MyInter1,MyInter2{   @Override  **public void** methodA() {  System.***out***.println(**"MyImpl 实现了 methodA方法"**);  }   @Override  **public void** methodB() {  System.***out***.println(**"MyImpl 实现了 methodB方法"**);  } }  测试代码  **public class** Test1 {  **public static void** main(String[] args) {  System.***out***.println(MyInter1.***A***);  MyInter1 mi1=**new** MyImpl();  mi1.methodA();  MyInter2 mi2=**new** MyImpl();  mi2.methodB();  MyImpl mi =**new** MyImpl();  *testa*(mi);  *testb*(mi);   }   **public static void** testa(MyInter1 myInter1){   }   **public static void** testb(MyInter2 myInter2){   } } |

**定义接口的详细说明：**

* **访问修饰符：**只能是public或默认。
* **接口名：**和类名采用相同命名机制。
* **extends：**接口可以多继承。
* **常量：**接口中的属性只能是常量，总是：public static final 修饰。不写也是。
* **方法：**接口中的方法只能是：public abstract。 省略的话，也是public abstract。

|  |
| --- |
| **要点**   * 子类通过**implements**来实现接口中的规范。 * 接口不能创建实例，但是可用于声明引用变量类型。 * 一个类实现了接口，必须实现接口中所有的方法，并且这些方法只能是public的。 * JDK1.8（不含8）之前，接口中只能包含静态常量、抽象方法，不能有普通属性、构造方法、普通方法。 * **JDK1.8（含8）后，接口中包含普通的静态方法、默认方法。** |

**需求：张三去试驾 开奔驰 奥迪 宝马。**

|  |
| --- |
|  |

**总结1：接口的组成**

* 接口和数组、类、抽象类是同一个层次的概念
* 成员变量:接口中所有的变量都使用public static final修饰，都是全局静态常量
* 成员方法： 接口中所有的方法都使用public abstract修饰，都是全局抽象方法
* 构造方法：接口不能new，也没有构造方法
* 接口做方法的形参，实参可以该接口的所有实现类

**总结2：接口和多继承**

* C++ 多继承

好处 ：可以从多个父类继承更多的功能  
 缺点：不安全 有两个父类Father1,Father2,都有一个方法giveMoney(),子类如果重写了，没有问题，如果子类没有重写，调用giveMoney()是谁的

* Java 单继承

好处：安全 缺点：功能受限  
 解决方案：既安全，功能又强大，采用接口。接口变相的使Java实现了C++的多继承，又没有C++多继承的不安全性  
 public class Bird extends Animal implements Flyable,Sleepable  
 必须先extends 再implements

**对比接口和抽象类的异同**

**相同:**

1 都可以有抽象方法,都要求子类/实现类重写,都是对子类/实现类的要求

2 都不可以被实例化

**不同:**

1 抽象类是被普通类继承,接口是被普通类实现,只有接口能继承接口

2 一个子类只能继承一个抽象类,一个实现类可以同时实现多个接口

3 抽象类中可以定义普通的成员变量,接口中只能定义共有的静态常量

4 抽象类中可以有非抽象方法,接口中只能是抽象方法(JDK1.7)

5 抽象类中可以有构造方法,接口中没有构造方法的

### 2.2接口新特征

**JDK7及其之前**

* 接口的变量都是public final static 全局静态常量，无变化
* 接口中都是抽象abstract方法，不能有static方法（因为abstract和static、final、private不能共存）

**JDK8及其之后**

* 接口中可以添加非抽象方法（static），实现类不能重写，只能通过接口名调用。
* 如果子类中定义了相同名字的静态方法，那就是完全不同的方法了，直接从属于子类。可以通过子类名直接调用
* 接口中可以添加非抽象方法（default），实现类可以重写，只能通过对象名调用
* 实现类可以直接使用default方法，可以重写default方法，但是必须去掉default。（default只能接口中使用）
* 上级接口中default方法的调用：MyInterface.super.method2()

**提供非抽象方法的目的**

* 为了解决实现该接口的实现类代码重复的问题
* 为了既有的成千上万的Java类库的类增加新功能，且不必对这些类重新进行设计。比如只需在Collection接口中增加default Stream<E> stream()，相应的Set和List接口以及它们的子类都包含此的方法，不必为每个子类都重新copy这个方法。

**JDK9及其之后**

**接口中可以定义private的非抽象方法**，便于将多个方法中的冗余代码进行提取，并且不对外公开。减少冗余、也实现了代码隐藏。

#### 【示例10】JDK8的接口新特征

|  |
| --- |
| **public interface** MyInterface {  **public static final double *PI*** = 3.14;  **public abstract void** method1();  **public static void** method2(){  System.***out***.println(**"JDK1.8中的非抽象方法有两种，一种是static的"**);  }  **public default void** method3(){  System.***out***.println(**"JDK1.8中的非抽象方法有两种，一种是default的"**);;  }  **public static void** main(String[] args) {  MyInterface.*method2*();  } }  **public class** MyClass **implements** MyInterface {  **public void** method1() {  System.***out***.println(**"接口中的抽象方法，子类必须实现"**);  }  **public void** method3(){  MyInterface.*method2*();  MyInterface.**super**.method3();  System.***out***.println(**"重写接口的default方法，须将default去掉"**);  }  **public static void** main(String[] args) {  MyInterface mi = **new** MyClass();  mi.method1();  MyInterface.*method2*();  mi.method3();  } } |

### 2.3 面向接口编程

**面向接口编程是面向对象编程的一部分。**

为什么需要面向接口编程? 软件设计中最难处理的就是需求的复杂变化，需求的变化更多的体现在具体实现上。我们的编程如果围绕具体实现来展开就会陷入”复杂变化”的汪洋大海中，软件也就不能最终实现。我们必须围绕某种稳定的东西开展，才能以静制动，实现规范的高质量的项目。

**接口就是规范，就是项目中最稳定的核心！ 面向接口编程可以让我们把握住真正核心的东西，使实现复杂多变的需求成为可能。**

通过面向接口编程，而不是面向实现类编程，可以大大降低程序模块间的耦合性，提高整个系统的可扩展性和和可维护性。

面向接口编程的概念比接口本身的概念要大得多。设计阶段相对比较困难，在你没有写实现时就要想好接口，接口一变就乱套了，所以设计要比实现难！

|  |
| --- |
| **老鸟建议**  接口语法本身非常简单，但是如何真正使用？这才是大学问。我们需要后面在项目中反复使用，大家才能体会到。 学到此处，能了解基本概念，熟悉基本语法，就是“好学生”了。 请继续努力！再请工作后，闲余时间再看看上面这段话，相信你会有更深的体会。 |

### 2.4 接口应用：内部比较器Comparable

图书类、学生类、新闻类、商品类等是不同的类，可是却都需要有比较的功能，怎么办？共同的父类不可以，可以定义一个比较接口Comparable，其中定义一个实现比较的方法compareTo(Object obj)。让各个类实现该接口即可。Java中就是这么来实现的，下面就来模拟实现一下Comparable接口吧。

#### 【示例7】定义Comparable接口

|  |
| --- |
| **interface** Comparable{  */\*\*  \* 两个对象对比大小的方法  \** ***@param obj*** *另一个对象  \** ***@return*** *当前对象如果比传入的对象大,则返回整数,如果小,返回负数,如果相同,返回0  \*/* **int** compareTo(Object obj); }  参照 java.lang.Comparable接口 |

#### 【示例8】实现Comparable接口

|  |
| --- |
| **public class** Student **implements** Comparable<Student>{   **private** String **name**;   **private int age**;   **public** Student(){}   **public** Student(String name, **int** age) {  **this**.**name** = name;  **this**.**age** = age;  }   **public** String getName() {  **return name**;  }  **public void** setName(String name) {  **this**.**name** = name;  }  **public int** getAge() {  **return age**;  }  **public void** setAge(**int** age) {  **this**.**age** = age;  }   @Override  **public** String toString() {  **return "Student{"** +  **"name='"** + **name** + **'\''** +  **", age="** + **age** +  **'}'**;  }   @Override  **public int** compareTo(Student o) {   **int** a= **this**.**age**;*//比较时候前面对象中age属性* **int** b= o.**age**;*//传递进来的对象中age属性* String aa=**this**.**name**;   String bb=o.**name**;  *//注意 compareTo 不是自己定义的compareTo方法 而是String中compareTo* **return** aa.compareTo(bb);  } } |

#### 【示例9】测试Comparable接口

|  |
| --- |
| **public class** TestA {   **public static void** main(String[] args) {   */\* String st1="bac8weuiwuwiew";   String st2="bac";   int n= st2.compareTo(st1);\*/* Student stu1=**new** Student(**"zs"**,78);  Student stu2=**new** Student(**"zs"**,19);  **int** i = stu1.compareTo(stu2);  System.***out***.println(i);  }  } */\* \* 引用类型的数据也是可以比较大小 \* 如果想要比较大小 对象需要实现Comparable \* 重写里面的compareTo方法 比较两个值做一个减法， \* 我们根据返回的值是正数 负数 还是0 就可以知道那个对象比较大了 \* \*/* |

### 2.5 接口应用：外部比较器Comparator

|  |
| --- |
| **public class** People {   **private** String **name**;   **private int age**;   **public** People(){}   **public** People(String name, **int** age) {  **this**.**name** = name;  **this**.**age** = age;  }  **public** String getName() {  **return name**;  }  **public void** setName(String name) {  **this**.**name** = name;  }   **public int** getAge() {  **return age**;  }   **public void** setAge(**int** age) {  **this**.**age** = age;  }  @Override  **public** String toString() {  **return "People{"** +  **"name='"** + **name** + **'\''** +  **", age="** + **age** +  **'}'**;  } } |

**自定义比较器算法**

|  |
| --- |
| **import** java.util.Comparator;  **public class** BJ **implements** Comparator<People> {  @Override  **public int** compare(People o1, People o2) {  **int** age1 = o1.getAge();  **int** age2 = o2.getAge();  **return** age1-age2;  } } |

总结一下，两种比较器Comparable和Comparator，后者相比前者有如下优点：

1、如果实现类没有实现Comparable接口，又想对两个类进行比较（或者实现类实现了Comparable接口，但是对compareTo方法内的比较算法不满意），那么可以实现Comparator接口，自定义一个比较器，写比较算法

2、实现Comparable接口的方式比实现Comparator接口的耦合性 要强一些，如果要修改比较算法，要修改Comparable接口的实现类，而实现Comparator的类是在外部进行比较的，不需要对实现类有任何修 改。从这个角度说，其实有些不太好，尤其在我们将实现类的.class文件打成一个.jar文件提供给开发者使用的时候。实际上实现Comparator 接口的方式后面会写到就是一种典型的策略模式。

### 本节作业

1. 接口的语法特点
2. 使用接口实现飞行比赛
3. 使用Comparable接口让Book类和Person类具备比较的能力
4. JDK1.8中接口的新变化
5. 定义图形接口interface Graphic,接口中要求有获得周长和面积的抽象方法,要求圆形类,长方形类和三角形类实现该接口并测试

|  |
| --- |
|  |

## 第三节 内部类

内部类是一类特殊的类，指的是定义在一个类的内部的类。实际开发中，为了方便的使用外部类的相关属性和方法，这时候我们通常会定义一个内部类。

一般情况，我们把类定义成独立的单元。有些情况下，我们把一个类放在另一个类的内部定义，称为内部类(innerclasses)。

在Java中内部类主要分为成员非静态成员内部类、静态成员内部类、局部内部类、匿名内部类。

### 3.1 非静态成员内部类

作为类的成员存在，和成员变量、成员方法、构造方法、代码块并列。因为是类的成员，所以非静态成员内部类可以使用public、protected 、默认、private修饰，而外部类只能使用public、默认修饰。

#### 【示例11】非静态成员内部类

|  |
| --- |
| **package** com.bjsxt.innerClassDemo1;  */\* \* 1类中的类就是内部类 \* 普通内部类 \* 静态内部类 \* 局部内部类 \* 匿名内部类 \* \* 普通内部类的特征 \* 1可以使用四个访问修饰符 \* 2内部类中可以直接使用外部类的成员变量 \* 3内部类可以直接调用外部类的成员方法 \* 4在内部类方法中 通过外部类类名.this.成员变量名区分内部类和外部类的同名成员变量 \* 5外部类是不能直接使用内部类的成员变量和成员方法的 \* 6内部类可以实现很好的隐藏,当一个类仅仅为另一个类服务,就可以将类作为内部类 \* 7内部类可以帮我我们区分因继承和实现接口所造成的方法不可区分的问题 \* \*/* **import** javax.print.attribute.standard.MediaSize;  **public class** OuterClass **implements** InterA{   *// 成员变量* **private** String **name** =**"OuterClass name"**;  *// 成员方法* **public void** showName(){  System.***out***.println(**name**);   }  **public void** testA(){  InnerClass ic =**new** InnerClass();  System.***out***.println(ic.**username**);  ic.showUsername();  eat();  ic.eat();  }  *// 构造方法* **public** OuterClass(){   }   @Override  **public void** eat() {   }   *// 成员内部类* **private class** InnerClass **extends** Person{  *//成员变量* String **username**=**"innerclass username"**;  String **name** =**"InnerClass name"**;  *//成员方法* **public void** showUsername(){  System.***out***.println(**username**);  */\* System.out.println(name);  System.out.println(OuterClass.this.name);  showName();\*/* }  *// 构造方法* **public** InnerClass(){  }  } }  **interface** InterA{  **void** eat(); }  **class** Person {  **public void** eat(){  System.***out***.println(**"吃饭"**);  } } |

**总结1：基本特征**

* 内部类可以直接访问外部类的成员
* 外部类不能直接访问内部类的成员，需要先创建对象再通过对象名访问
* 内部类如何访问外部类的同名成员变量：OuterClass.this.num
* 必须先创建外部类的对象，才能创建内部类的对象。非静态成员内部类是属于某个外部类对象的

**总结2：更多特征**

* 非静态内部类不能有静态方法、静态属性和静态初始化块。
* 外部类的静态方法、静态代码块不能访问非静态内部类，包括不能使用非静态内部类定义变量、创建实例。

|  |
| --- |
| **注意**  内部类只是一个编译时概念，一旦我们编译成功，就会成为完全不同的两个类。对于一个名为Outer的外部类和其内部定义的名为Inner的内部类。编译完成后会出现Outer.class和Outer$Inner.class两个类的字节码文件。所以内部类是相对独立的一种存在，其成员变量/方法名可以和外部类的相同。 |

### 3.2 静态成员内部类

#### 【示例12】静态成员内部类

|  |
| --- |
| **package** com.bjsxt.innerClassDemo2;  */\* \* 1类中的类就是内部类 \* 普通内部类 \* 静态内部类 \* 局部内部类 \* 匿名内部类 \* \* 8static修饰内部类 静态内部类 静态内部类只能直接使用外部类的静态成员 \* 9一个类不仅仅可以有内部类.还可以有内部抽象类和内部接口 \* \*/* **public class** OuterClass {   *// 成员变量* **private static** String *name* =**"OuterClass name"**;  *// 成员方法* **public static void** showName(){  System.***out***.println(*name*);   }  *// 构造方法* **public** OuterClass(){   }   **public void** testa() {  InnerClass ic =**new** InnerClass();  System.***out***.println(ic.**username**);  ic.showUsername();  }   *// 成员内部类* **static class** InnerClass {  *//成员变量* String **username**=**"innerclass username"**;   *//成员方法* **public void** showUsername(){  System.***out***.println(**username**);  System.***out***.println(*name*);  *showName*();  }  *// 构造方法* **public** InnerClass(){  }  }    **abstract class** InnterClassA{   }   **interface** InterA{   }   } |

**总结：**

* **静态内部类只能够访问外部类的静态成员**
* 静态内部类如何访问外部类的同名的成员变量：OuterClass.num
* 静态内部类属于整个外部类的。创建静态内部类的对象，不需先创建外部类的对象
* 外部类可以通过类名直接访问内部类的静态成员，访问非静态成员依旧需要先创建内部类对象。

### 3.3 局部内部类

还有一种内部类，它是定义在方法内部的，作用域只限于本方法，称为局部内部类。

局部内部类的的使用主要是用来解决比较复杂的问题，想创建一个类来辅助我们的解决方案，到那时又不希望这个类是公共可用的，所以就产生了局部内部类。局部内部类和成员内部类一样被编译，只是它的作用域发生了改变，它只能在该方法中被使用，出了该方法就会失效。

局部内部类在实际开发中应用很少。

#### 【示例13】局部内部类

|  |
| --- |
| **package** com.bjsxt.innerClassDemo3;  */\* \* 1在方法中声明的类就是局部内部类 \* 2仅仅在当前方法中可用 \* 3局部内部类中 只能直接使用外部类的常量 \* 4JDK1.8开始,局部内部类中使用常量的final关键字可以省略不写 \* \*/* **public class** Test1 {  **public static void** main(String[] args) {  **final int** i=10;     **class** A{  String **name**;  **public void** showName(){  System.***out***.println(i);   }  **public** A(){   }  }   A a =**new** A();  }  **public static void** method(){   }   } |

**注意：局部内部类访问所在方法的局部变量，要求局部变量必须使用final修饰。JDK1.8中final可以省略，但是编译后仍旧会加final。**

### 3.4 匿名内部类

匿名内部类就是内部类的简化写法，是一种特殊的局部内部类。

前提：存在一个类或者接口，这里的类可以是具体类也可以是抽象类。

本质是什么呢?是一个继承了该类或者实现了该接口的子类匿名对象。

适合那种只需要创建一次对象的类。比如：Java GUI编程、Android编程键盘监听操作等等。比如Java开发中的线程任务Runnble、外部比较器Comparator等。

|  |
| --- |
| **package** com.bjsxt.innerClassDemo4;  */\* \* 当一个接口中/抽象类中 抽象方法的数量比较少 应用的位置也比较少  \* 接口的实现类对象 应用位置也比较少 \* 用一个类去实现接口就显得比较麻烦 \* 可以通过局部内部类去简化写法 \* 匿名内部类可以帮助我们快速的为接口或者抽象类产生一个实现对象/子类对象 \* 匿名内部类没有构造方法 \* 匿名内部类自定义的方法和属性的getset方法往往是无法调用的 \* 只能使用代码块对成员变量进行初始化 \* 匿名内部类作为局部内部类的一种,是方法内的内部类 \*  \* \* \*/* **public class** Test1 {  **public static void** main(String[] args) {   **int** value=10;    A a=**new** A(){  **private** String **name**;  {  String[] names={**"Tom"**,**"Jhon"**,**"Mark"**,**"King"**,**"Allen"**,**"Scott"**};  **int** index =(**int**)(Math.*random*()\*names.**length**);  **name**=names[index];  }  **public void** methodA(){   }  **public void** setName(String name){  **this**.**name**=name;  }   @Override  **public int** getNum(**int** i) {  System.***out***.println(**name**);  System.***out***.println(value);    methodA();  **return** (**int**)(Math.*random*()\*i);  }   };    System.***out***.println(a.getNum(10));     B b =**new** B(){  @Override  **public int** getNum(**int** i) {  **return** 10;  }  };  } }   **interface** A{  */\*\*  \* 根据范围返回随机正整数  \** ***@param i*** *范围  \** ***@return*** *随机整数,不包含给定的范围值  \*/* **int** getNum(**int** i); } **abstract class** B{  **public abstract int** getNum(**int** i); } |

**语法：**

|  |
| --- |
| new 父类构造器(实参类表) \实现接口 () {  //匿名内部类类体！  } |

结论：

* 匿名内部类可以实现一个接口，也可以继承一个类（可以是抽象类）。
* 匿名内部类只能实现一个接口，而不是多个
* 必须实现所有的方法，匿名内部类不能是抽象类
* 匿名内部类不可能有构造方法，因为类是匿名的
* 匿名内部类没有访问修饰符
* 如果想实现构造方法的一些初始化功能，可以通过代码块实现
* 如果要访问所在方法的局部变量，该变量需要使用final修饰。（JDK1.8可省略final）

### 3.5 内部类的作用和使用场合

**内部类的作用：**

* 内部类提供了更小的封装。只能让外部类直接访问，不允许同一个包中的其他类直接访问。
* 内部类可以直接访问外部类的私有属性，内部类被当成其外部类的成员。 但外部类不能访问内部类的内部属性。
* 接口只是解决了多重继承的部分问题，而内部类使得多重继承的解决方案变得更加完整。
* 用匿名内部类实现回调功能。我们用通俗讲解就是说在Java中，通常就是编写一个接口，然后你来实现这个接口，然后把这个接口的一个对象作以参数的形式传到另一个程序方法中， 然后通过接口调用你的方法，匿名内部类就可以很好的展现了这一种回调功能

**内部类的使用场合：**

* 由于内部类提供了更好的封装特性，并且可以很方便的访问外部类的属性。所以，在只为外部类提供服务的情况下可以优先考虑使用内部类。
* 使用内部类间接实现多继承：每个内部类都能独立地继承一个类或者实现某些接口，所以无论外部类是否已经继承了某个类或者实现了某些接口，对于内部类没有任何影响。

### 本节作业

1. 练习非静态成员内部类
2. 练习静态成员内部类
3. 练习局部内部类
4. 扩展：为什么局部内部类访问的所在方法的局部变量要求是final的。

参考：<https://blog.csdn.net/dazhaoDai/article/details/83097017>

1. 说明匿名内部类的相关技能点
2. 使用匿名内部类实现Comparator比较器
3. 内部类的作用和使用场合

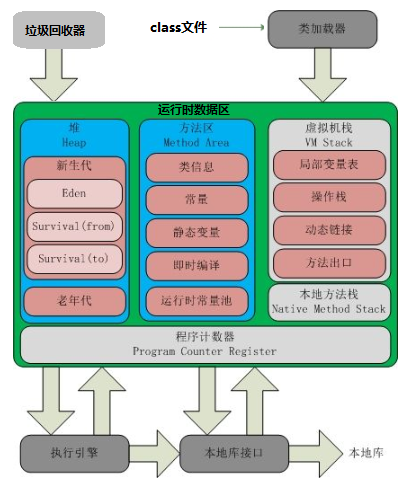
## 第四节 虚拟机和垃圾回收

### 4.1 虚拟机及其构成

虚拟机指通过软件模拟的具有完整硬件系统功能的、运行在一个完全隔离环境中的完整计算机系统。Java虚拟机对字节码进行解释生成对应平台的机器码并执行。Java虚拟机是Java跨平台的重要原因。

Java虚拟机在执行Java程序的过程中会把它所管理的内存划分为若干个不同的数据区域。

主要包括类加载器、运行时数据区、执行引擎、本地方法接口、本地方法库、垃圾回收器。其中重点关注的运行时数据区，又可细分为方法区、堆区、虚拟机栈、本地方法栈、程序计数器，其中方法区和堆区为进程的所有子线程共享，其它的为线程独有。



**类加载器**

负责把class文件装载到内存中，共虚拟机执行。

**执行引擎**

负责执行字节码

**本地方法库：**

本地方法（native方法）就是一个Java调用非Java代码的接口。一个native方法是指该方法的实现由非Java语言实现，比如用C或C++实现。

**本地方法库接口JNI:**

允许Java代码使用以其他语言编写的native本地方法库。

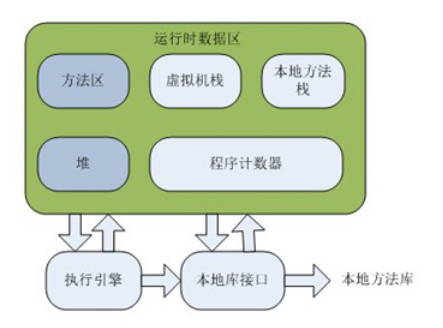
**垃圾回收器**

回收无用对象所占用的内存空间。

**运行时数据区**

JVM的核心区域，程序运行时的工作区域，数据加工处理的场所。下面进行重点说明。

### 4.2 运行时数据区



**程序计数器**

程序计数器是一块较小的内存空间，它的作用可以看作是当前线程所执行的字节码的行号指示器。每个线程都有一个独立的程序计数器，各条线程之间的计数器互不影响，独立存储，是线程私有内存。如果正在执行Native方法，则这个计数器为空。

**Java虚拟机栈(Java Virtal Machine Stack)**

同样也是属于线程私有区域，每个线程在创建的时候都会创建一个虚拟机栈，生命周期与线程一致，线程退出时，线程的虚拟机栈也回收。虚拟机栈内部保持一个个的栈帧，每次方法调用都会进行压栈，JVM对栈帧的操作只有出栈和压栈两种，方法调用结束时会进行出栈操作。该区域存储着局部变量表，编译时期可知的各种基本类型数据、对象引用、方法出口等信息。

**本地方法栈（Native Method Stack）**

与虚拟机栈类似，本地方法栈是在调用本地方法时使用的栈，每个线程都有一个本地方法栈。

**Java堆（heap）**

Java堆是垃圾收集器管理的主要区域，因此很多时候也被称为GC堆，Java堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。此区域的唯一目的就是存放对象实例，几乎所有对象的实例都在这分配内存。

**方法区**

方法区和Java堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息（Class对象，反射机制中重点讲授），常量，静态变量，即时编译器编译后的代码等数据。

方法区是一种Java虚拟机的规范。由于方法区存储的数据和堆中存储的数据一致，实质上也是堆，因此，在不同的JDK版本中方法区的实现方式不一样。

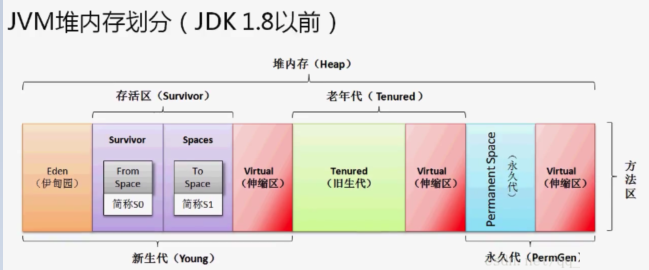
### 4.3堆内存

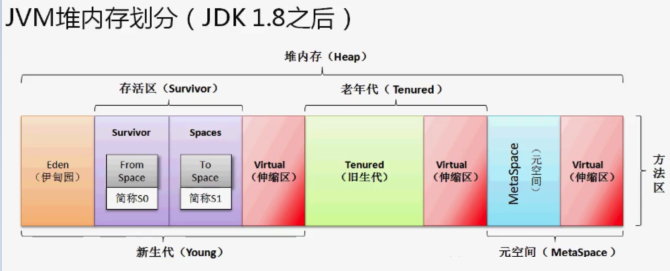
堆内存分为三部分

1.年轻代：Young

2.老年代：Tenured

3.永久代 PermGen JDK8中变成元空间MetaSpace





**年轻代：分为eden区+两个大小相同的存活期s0、s1；**

所有使用关键字new新实例化的对象，一定会在伊甸园区进行保存(除非大对象，伊甸园区容不下)；存活区会分为两个相等大小的存活区，存活区保存的一定是在伊甸园区保存好久，并且经过了好几次的小GC还保存下来的活跃对象，那么这个对象将晋升到存活区中。

存活区一定会有两块大小相等的空间。目的是一块存活区未来晋升，另外一块存活区为了对象回收。这两块内存空间一定有一块是空的。

在年轻代中使用的是MinorGC，这种GC采用的是复制算法

**老年代**

主要接收由年轻代发送过来的对象，一般情况下，经过了数次Minor GC 之后还会保存下来的对象才会进入到老年代。每次进行Minor GC 后存活的对象，年龄都会+1，到了一定年龄后（默认15），进入老年代

如果要保存的对象超过了伊甸园区的大小，此对象也将直接保存在老年代之中；

当老年代内存不足时，将引发 “major GC”，即，“Full GC”。

**永久代**

永久代：是HotSpot的一种具体实现，实际指的就是方法区，JDK1.8 之后将最初的永久代内存空间取消了，代之以元空间（metaspace）。

为什么废弃永久代:除永久代是为融合HotSpot JVM与 JRockit VM而做出的努力，因为JRockit没有永久代，不需要配置永久代。另外由于永久代内存经常不够用或发生内存泄露，爆出异常java.lang.OutOfMemoryError: PermGen。

元空间功能和永久代类似，唯一到的区别是：永久代使用的是JVM的堆内存空间，而元空间使用的是物理内存，直接受到本机的物理内存限制。

**注意1：**

* JDK6 及以前版本，字符串常量池是放在永久代的（方法区）。
* JDK7 的版本中，字符串常量池已经从Perm区移到正常的Java Heap区域了。为什么要移动，Perm 区域太小是一个主要原因
* JDK8中，永久代被元空间替代。

**注意2：**

现在最新的垃圾回收算法中取消了年轻代、老年代的划分。方法区也几经变迁，但是对于我们初始学习时候，仍可按照最初的规范讲解，先熟悉基本的概念，不必过细的深入底层。

|  |
| --- |
| **扩展：内存泄漏和内存溢出**  内存泄漏（Memory Leak）：对象所要做的事情完成了，希望他们会被回收掉。但如果还有对这个对象的其他引用，它是不会被回收的，它还会占用内存。持续累加，内存很快被耗尽。  内存溢出（out of memory） :指程序申请内存时，没有足够的内存供申请者使用。  内存泄漏：租了个带钥匙的柜子，你存完东西之后把柜子锁上之后，把钥匙丢了或者没有将钥匙还回去，那么结果就是这个柜子将无法供给任何人使用。  内存溢出：柜子少人多，柜子都被占用了，没有多余的柜子。但是可能是因为有效占用，也可能是因为不还钥匙的无效占用。 |

### 4.4如何判定哪些对象需要回收？

在对堆进行对象回收前，首先要判断哪些是无效对象。我们知道，一个对象不被任何对象或变量引用，那么就是无效对象，需要被回收。

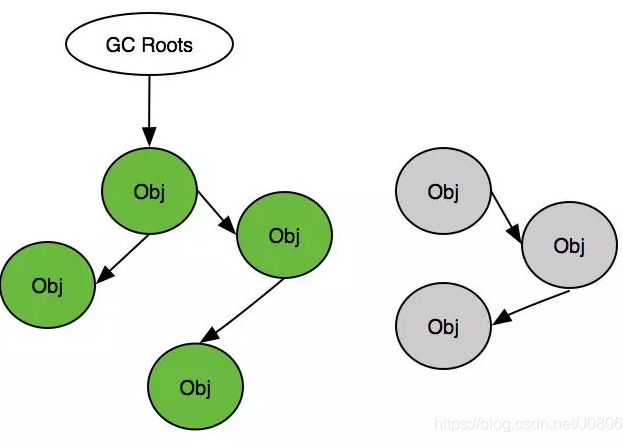
一般有两种判别方式：

[1]引用计数法

每个对象都有一个计数器，当这个对象被一个变量或者另一个对象引用一次，该计数器加一；若该引用失效，则计数器减一。当计数器为0时，就认为该对象是无效对象。

[2]可达性分析法(根搜索算法)

所有和GC Roots直接或间接关联的对象都是有效对象，和GC Roots没有关联的对象就是无效对象。算法的基本思路就是通过一系列的称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为“引用链(Reference Chain)”。当一个对象到GC Roots没有任何引用链相连(用图论的话来说，就是从GC Roots到这个对象不可达)时，则证明此对象是不可用的，可以被回收的。



即使在可达性分析算法中不可达的对象，也并非是“非死不可”的，这时候它们暂时处于“缓刑”阶段，要真正宣告一个对象死亡，至少要经历两次标记过程。

如果对象在进行可达性分析后发现没有与GC Roots相连接的引用链，那它将会被第一次标记并且进行一次筛选，筛选的条件是次对象是否有必要执行finalize()方法。当对象没有覆盖finalize()方法，或者finalize()方法已经被虚拟机调用过时，虚拟机将这两种情况都视为“没有必要执行”。

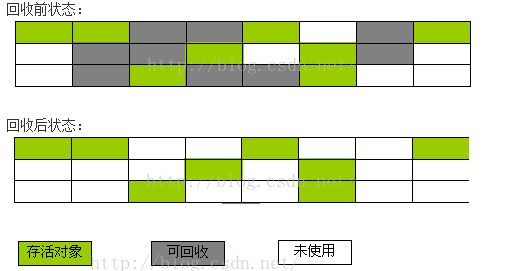
如果这个对象被判定为有必要执行finalize()方法，那么这个对象将会被放置在一个叫做F-Queue的队列之中，并在稍后由一个虚拟机自动建立的、低优先级的Finalizer线程去执行它。

inalize()方法是对象逃脱死亡名义的最后一次机会，稍候GC将对F-Queue中的对象进行第二次小规模标记。如果对象要在finalize()中成功拯救自己–只要重新与引用链上的任何一个对象建立关联即可。譬如把自己(this关键字)赋值给某个类变量或者对象的成员变量，那在第二次标记时它将被移除出“即将回收”的集合。如果对象这时候还没有逃脱，那基本上它就真的被回收了。

### 4.5垃圾回收算法

**2.1 标记-清除**

是最基础的一种收集算法。分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。标记过程就是上面可达性分析算法中所讲的二次标记过程。标记-清除算法的执行过程如下图所示：



缺点：

（1） 效率问题：标记和清除的两个过程效率都不高；

（2） 空间问题：标记清除后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后需要分配较大对象时，无法找到足够的连续内存而不得不提前出发另一次垃圾收集动作；

**2.2 复制算法**

为了解决上面算法的效率问题，复制算法出现。它将可用内存按容量划分为 大小相等的两块，每次只使用其中的一块。当这一块的内存使用完了，就将还存活的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。

复制算法的 优点：

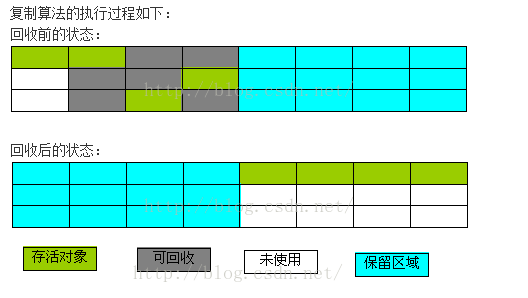
（1）每次都是对 整个半区进行内存回收，实现简单、运行也高效；

（2）在那块使用内存上进行内存分配时，不用考虑内存碎片的问题，只要移动堆顶指针，按顺序分配内存即可；

缺点：

将内存缩小为原来的一半，代价较高。

复制算法的执行过程如下：



按照新生代的特点，新生代中的对象98%是“朝生夕死”的，因此，可以改进上面的复制算法，目前商业虚拟机正是用这种改进的收集算法来 回收新生代。

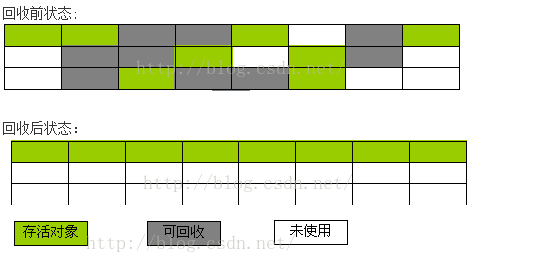
改进的收集算法：

根据新生代的特点，我们并不需要按照1:1的比例来划分内存空间，而是将内存 划分为一块较大的Eden空间和两块较小的Survivor空间， 每次使用Eden和其中一块Survivor。当回收时，将Eden和Survivor中还存活的对象一次性地复制到另外一块Survivor空间，最后清理掉Eden和刚才用过的Survivor空间，清理完成后，刚刚被清理的Eden和另一块在回收时放入存活对象的Survivor空间作为使用内存，刚被清理的Survivor作为保留空间，以便后面用来回收之用。

这种改进的收集算法也有一个问题，就是在回收时，那块空的Survivor空间能否放得下Eden和使用的Survivor空间中还存活的对象，如果Survivor空间不够存放上一次新生代收集下来的存活对象，此时就需要向老年代“借”内存，那些剩余未放下的对象就通过 分配担保机制进入老年代。

**2.3 标记-整理算法**

复制算法如果在对象存活率较高时，就需要进行较多次的复制操作，效率也会变低。而对于老年代中的对象，一般存活率都较高，因此需要选用其他收集算法：标记 - 整理算法。标记过程仍然与“标记-清除”算法中一样，但是在标记完成后并不直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。算法示意图如下：



**2.4 分代收集算法**

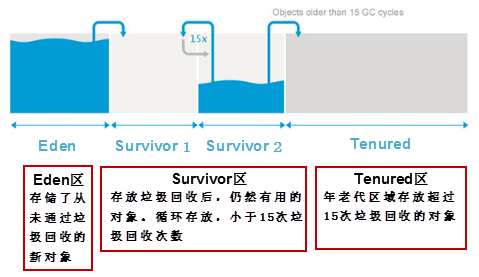
当前商业虚拟机都采用这个“分代收集”算法（Generation Collection），它根据对象存活周期的不同将内存划分为几块，一般是把java堆分为新生代和老年代，根据各个年代的特点选用不同的收集算法。在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，因此可以选用“复制算法”，此时只需要付出少量存活对象的复制成本即可；对于老年代，因为对象存活率较高、也没有额外空间为期分配担保，就必须使用“标记-清除”或“标记-整理”算法来进行回收。

**分代垃圾回收机制**，是基于这样一个事实：不同的对象的生命周期是不一样的。因此，不同生命周期的对象可以采取不同的回收算法，以便提高回收效率。我们将对象分为三种状态：年轻代、年老代、持久代。同时，将处于不同状态的对象放到堆中不同的区域。 JVM将堆内存划分为 Eden、Survivor 和 Tenured/Old 空间。

1. 年轻代：所有新生成的对象首先都是放在Eden区。 年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象，对应的是Minor GC，每次 Minor GC 会清理年轻代的内存，算法采用效率较高的复制算法，频繁的操作，但是会浪费内存空间。当“年轻代”区域存放满对象后，就将对象存放到年老代区域。

2. 年老代：在年轻代中经历了N(默认15)次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。年老代对象越来越多，我们就需要启动Major GC和Full GC(全量回收)，来一次大扫除，全面清理年轻代区域和年老代区域。

3. 永久代：用于存放静态文件，如Java类、方法等。持久代对垃圾回收没有显著影响。JDK7以前就是“方法区”的一种实现。JDK8以后已经没有“永久代”了，使用metaspace元数据空间和堆替代。



**垃圾回收的相关技能点**

* 垃圾回收机制主要是回收JVM堆内存里的对象空间。
* 垃圾回收发生具有不可预知性，程序无法精确控制垃圾回收机制执行。
* STW:stop the world。垃圾回收时会暂停所有程序执行，不同的垃圾算法暂停的时间会差异很大。减少STW时间也是各垃圾回收算法追求的目标。
* 可以将对象的引用变量设置为null，暗示垃圾回收机制可以回收该对象。
* 程序员可以通过System.gc()或者Runtime.getRuntime().gc()来通知系统进行垃圾回收，会有一些效果，但是系统是否进行垃圾回收依然不确定。
* 垃圾回收机制回收任何对象之前，总会先调用它的finalize方法（如果覆盖该方法，让一个新的引用变量重新引用该对象，则会重新激活对象）。
* 永远不要主动调用某个对象的finalize方法，应该交给垃圾回收机制调用。

#### 【示例17】finalize()和gc()

|  |
| --- |
| **public class** Student {  @Override  **protected void** finalize() **throws** Throwable {  System.**out**.println(**"-----gc-----------"**);  }  **public static void** main(String[] args) {  **new** Student();  **new** Student();  **new** Student();  **new** Student();  **new** Student();  //System.gc();  Runtime.getRuntime().gc();  } } |

### 本节作业

1. 虚拟机的内存模型
2. 堆内存在JDK1.7和JDK1.8的组成结构
3. 垃圾回收的主要步骤