

初始化与清理

0. 规范

1 | 1. final和static同时修饰的基本数据类型，变量名全部大写，用下划线分隔各个单词。

1. 静态初始化

1 | 1. 静态初始化只有在必要时刻才会进行，只有在第一次访问静态数据（或者第一个对象被创建（因为构造函数也是静态方法））的时候，静态变量才会被初始化。
2 | 2. 初始化的顺序是：先静态对象（注意第3点），后非静态对象，然后才调用构造函数
3 | 3. 构造器也是static方法，尽管static关键字没有显式地写出来。因此更准确地讲，类是在其任何static成员被访问时加载的。

2. 静态块（静态子句）

1 | 1. 只有在必要时刻才会进行，只有在第一次访问静态数据（或者第一个对象被创建）的时候，静态变量才会被初始化。
2 | 2. 格式：
3 | static {
4 |
5 | }

3. 非静态实例初始化

1 | 1. 语法格式 {}
2 | 2. 语句块，在构造函数之前初始化。

4. 数组初始化

1 | 1. int[] a1 = {1, 2, 3, 4, 5};
2 | 等价于使用new来分配空间
3 | a1 是数组的引用
4 |

5. this关键字

1 | 1. 在构造器中调用构造器
2 | public class f{
3 | f(int a){...}
4 | f(int a, int b){
5 | this(a);
6 |
7 | }
8 | }
9 | this调用别的构造器，必须置于起始处，否则编译报错（也就是说只能调用一个构造器）
10 | 2. static方法是没有this的方法，在static方法内部不能调用非静态方法（不是完全不可能），反之可以。

6. 清理：终结处理和垃圾回收 (finalize () 方法)

```
1 补充1：垃圾回收器准备释放对象占用的存储空间时，将首先调用其finalize()方法，并且在下一次垃圾回收动作发生  
2 时，才会真正回收对象占用的内存。  
3 补充2：无论是“垃圾回收”还是“终结”，都不保证一定会发生。如果jvm并未面临内存耗尽的情形，他是不会浪  
4 费时间去执行垃圾回收以恢复内存。  
5  
6 1. 三点知识  
7 ① 对象可能不被垃圾回收  
8 ② 垃圾回收不等于析构  
9 ③ 垃圾回收只与内存有关  
10 2. 垃圾回收器只释放由new分配的内存，所以如果有对象获得了一块‘特殊’的内存区域，需要定义finalize()方法  
11 (不等同于析构函数)。  
12 3. finalize () 主要作用于使用“本地方法”情况，调用非java代码，如c语言的malloc函数申请内存，要自己释  
13 放，垃圾回收机制不知道如何释放该内存，要在finalize函数中调用free () 函数释放内存。
```

7. 可变参数列表

```
1 class aaa{}  
2 public class array {  
3     static void printArr(Object[] args){  
4         for(Object obj : args){  
5             System.out.print(obj.getClass().getName() + " ");  
6         }  
7         System.out.println();  
8     }  
9     public static void main(String[] args) {  
10         printArr(new Object[] {new aaa(), 1, 1.0, 3L});  
11     }  
12 }  
13 //java se5之后可以用下面这种方法  
14 2. static void fun(Object... args){  
15     for(Object obj : args)System.out.print(obj + " ");  
16 }  
17 //另外可以采用下面这种方法调用  
18 fun(aaa(), 1, 1.0, 3L);  
19 //第一种定义，这样调用会报错。  
20 /*错误:(19, 9) java: 无法将类 mycode.array中的方法      printArr应用到给定类型;  
21 需要: java.lang.Object[]  
22 找到: mycode.aaa,int,double,long  
23 原因: 实际参数列表和形式参数列表长度不同*/  
24 3. 使用2时，可变参数列表可以和自动包装机制共存，自动包装机制将有选择的将int参数提升为Integer.  
25 4. 当不使用参数调用可变参数列表时，编译器会不知道调用哪一个方法，可以通过在某个方法中增加一个非可变参  
数来解决。  
26 5. 注意：你应该总是只在重载方法的一个版本上使用可变参数列表，或者压根就不是用它。
```

8. enum类型

```
1 1. 定义 // enumWeek.java  
2     public enum enumWeek{  
3         monday, tuesday, wednesday, thursday, friday,           saturday, sunday //常量一般大写  
4     }  
5 2. 使用 // enumInstance.java  
6     public class enumInstance {  
7         public static void main(String[] args) {  
8             enumWeek week = enumWeek.friday;//定义引用使用它  
9             for(enumWeek e : enumWeek.values())           System.out.println(e + ", ordinal = "+e.ordinal());  
10        }  
11    }  
12    /*  
13     * 运行结果  
14     * monday, ordinal = 0  
15     * tuesday, ordinal = 1
```

```
16 * wednesday, ordinal = 2
17 * thursday, ordinal = 3
18 * friday, ordinal = 4
19 * saturday, ordinal = 5
20 * sunday, ordinal = 6
21 */
22 3. enum会创建toString()方法，这正是能直接sout的原因。
23 4. enum会创建ordinal()方法，表示某个常量的声明顺序。
24 5. enum会创建static values()方法，按照常量的声明顺序，产生这些常量值构成的数组，可以用values()[1]，表示
25 第二个枚举tuesday。
26 6. enum看起来类似于一个数据类型，但是enum是一个类，并且具有自己的方法。可以吧enum当做任何类一样来
处理。
27 7. 与switch配合使用，是个绝佳组合。
```

第六章 访问权限控制

✓ 1. 组合与继承

```
1 在面向对象编程中，可以运用组合技术使用现有的类来开发新的类，而继承技术是不太常用的。在OOP时，应慎用继
2 承。继承的使用场合仅限于你确信使用该技术确实有效的情况。
3 到底该用组合还是继承，一个最清晰的判断方法就是问问自己是否需要从新类向基类进行向上转型。如果必须向上转
型，则继承是必要的；如果不需要，则应当好好考虑自己是否需要继承。
```

✓ 2. final关键字

```
1 1. 对于基本数据类型，final使数值恒定不变；对于对象引用，final使引用恒定不变（一旦引用被初始化指向一个对
象，就无法再把它改为指向另一个对象。然而，对象其自身却是可以被修改的，java未提供是任何对象恒定不变的
途径，但是可以自己编写类实现此效果），这一限制同样适用于数组，它也是对象。
2
3 2. 我们不能因为某数据是final，就认为在编译时就可以知道它的值。例如
4 private final int i4 = rand.nextInt(20);
5 static final int INT_5 = rand.nextInt(20);
6
7 3. java允许生成“空白final (blank final)”（指被声明为final但又未给定初值的域）。无论什么情况，编译器都
8 确保空白final在使用前被初始化。如何做到这样？就是在域的定义处或者每个构造器中用表达式对final进行赋
9 值。
10
11 4. java允许在参数列表中以声明的方式将参数指明为final。这意味着在方法中无法更改参数引用所指向的对象（同
1）。
12 5. private方法属于final方法
```

第七章 多态

✓ 1. 实现多态时，要覆盖父类的方法，而不是重载

```
1 //file1.java
2 public enum Note{
3     MIDDLE_C, C_SHARP, B_FLAT;
4 }
5 //file2.java
6 class Instrument{
```

```

7  public void play(Note n){
8      System.out.println("Instrument.play()");
9  }
10 }
11 //file3.java
12 public class Wind extends Instrument{
13     //注意play的参数列表，和父类一样，谓之方法覆盖
14     public void play(Note n){
15         System.out.println("Wind.play() " + n);
16     }
17 }
18 //file4.java
19 public class Music{
20     public static void main(String [] args){
21         Wind flute = new Wind();
22         tune(flute); //Upcasting
23     }
24 }
25 /*-----Output-----
26 Wind.play() MIDDLE_C
27 *-----End-----/

```

✓ 2. 方法调用绑定（将一个方法调用同一个方法主体关联起来被称为绑定）

1 前期绑定：在程序执行之前进行绑定（如果有的话，由编译器和连接程序实现）
 2 后期绑定（动态绑定或运行时绑定）：在运行时根据对象的类型进行绑定。

✓ 3. 注意点（缺陷：“覆盖私有方法”）

结论：只有非private方法才可以被覆盖

例子：

```

1  public class PrivateOverride{
2      private void f(){
3          System.out.println("private f()");
4      }
5      public static void main(String[] args){
6          PrivateOverried po = new Derived();
7          po.f();
8      }
9  }
10 class Derived extends PrivateOverride{
11     public void f(){
12         System.out.println("public f()");
13     }
14 }
15 /*-----Output-----
16 private f()
17 *-----End-----/
18 /*
19 我们期望的输出是public f(), 但是由于private方法被自动认为是final方法，而且对导出类时屏蔽的。因此，在这种情况下，Derived类中的f()方法就是一个全新的方法；既然基类f()方法在子类Derived中不可见，因此甚至也不能被重载。
20 */

```

✓ 4. 注意点（缺陷：域与静态方法）

只有普通的方法调用可以是多态的，静态方法的行为也不具有多态性。例如，如果你直接访问某个域，这个访问就将在编译期进行解析，如下例子：

```
1 class Super{
2     public int field = 0;
3     public int getField(){return field;}
4 }
5 class Sub extends Super{
6     public int field = 1;
7     public int getField(){return field;}//覆盖
8     public int getSuperField(){return super.field;}
9 }
10 public class FieldAccess{
11     public static void main(String[] args){
12         Super sup = new Sub(); //Upcast
13         System.out.println(sup.field + " " + sup.getField());
14     }
15 }
16 /*****Output*****
17 0 1
18 ****End****/
19 /*
20 当Sub对象转型为Super引用时，任何域访问操作都将由编译器解析，因此不是多态的。在本例中，为Super.field和
21 Sub.field分配了不同的存储空间。
22 */
```

✓ 5. 构造器和多态

1. 基类的构造器总是在导出类的构造过程中被调用，而且按照继承层次逐渐向上链接，以使每个基类的构造器都能得到调用。
2. 组合、继承以及多态在构建顺序上的作用举例：

```
1 class Meal{
2     Meal(){print("Meal");}
3 }
4 class Bread{
5     Bread(){print("Bread");}
6 }
7 class Cheese{
8     Cheese(){print("Cheese");}
9 }
10 class Lettuce{
11     Lettuce(){print("Lettuce");}
12 }
13 class Lunch extends Meal{
14     Lunch(){print("Lunch");}
15 }
16 class PortableLunch extends Lunch{
17     PortableLunch(){print("PortableLunch");}
18 }
19 public class Sandwich extends PortableLunch{
20     private Bread b = new Bread();
21     private Cheese c = new Cheese();
22     private Lettuce l = new Lettuce();
23     public Sandwich(){print("Sandwich");}
```

```

24 public static void main(String[] args){
25     new Sandwich();
26 }
27 }
28 /******Output*****
29 Meal()
30 Lunch()
31 PortableLunch()
32 Bread()
33 Cheese()
34 Lettuce()
35 Sandwich()
36 *****End*****/

```

由上可见复杂对象调用构造器要遵照下面的顺序：

- 1 调用基类构造器。这个步骤会不断反复递归下去，首先是构造这种层次结构的根，然后是下一层导出类，等等，直到最低层的导出类（本类的直接父类）。
- 2 按声明顺序调用成员的初始化方法。
- 3 调用导出类构造器的主体。

3. 构造器内部的多态方法的行为

```

1 class Glyph{
2     void draw(){print("Glyph.draw()");}
3     Glyph(){
4         print("Glyph() before draw()");
5         draw();
6         print("Glyph() after draw()");
7     }
8 }
9 class RoundGlyph extends Glyph{
10    private int radius = 1;
11    RoundGlyph(int r){
12        radius = r;
13        print("RoundGlyph.RoundGlyph(), radius = " + radius);
14    }
15    void draw(){
16        print("RoundGlyph.draw(), radius = " + radius);
17    }
18 }
19 public class aaa{
20     public static void main(String[] args){
21         new RoundGlyph(5);
22     }
23 }
24 /******Output*****
25 Glyph() before draw()
26 RoundGlyph.draw(), radius = 0
27 Glyph() after draw()
28 RoundGlyph.RoundGlyph(), radius = 5
29 *****End*****/
30 /*
31 Glyph.draw()方法设计为将要被覆盖，这种覆盖是在RoundGlyph中发生的，但是Glyph构造器会调用这个
方法，结果导致了RoundGlyph.draw()的调用。当Glyph的构造器调用draw（）方法时，radius不是默认初
始值1，而是0。所以这样的方式，会使结果难以预料。从而得出以下结论（初始化的实际过程）
32 */

```

初始化的实际过程：

```
1 | 1. 在其他任何事物发生之前，将分配给对象的存储空间初始化成二进制的零。
2 | 2. 如前所述那样调用基类构造器。此时调用被覆盖后的draw()方法（要在调用RoundGlyph构造器之前调用），由于步骤1的缘故，我们此时会发现radius的值为0。
3 | 3. 按照声明的顺序调用成员的初始化方法。
4 | 4. 调用导出类的构造器主体。
5 |
6 | 这样做的优点：所有东西都至少初始化为0
```

✓ 6. 协变返回类型（导出类（子类）覆盖（即重写）基类（父类）方法时，返回的类型可以是基类方法返回类型的子类）

```
1 | class Grain{
2 |     public String toString(){return "Grain";}
3 | }
4 | class Wheat extends Grain{
5 |     public String toString(){return "Wheat";}
6 | }
7 | class Mill{
8 |     Grain process(){return new Grain();}
9 | }
10 | class WheatMill extends Mill{
11 |     //方法覆盖：
12 |     Wheat process(){return new Wheat();}
13 | }
14 | /*
15 | java SE5与java较早版本之间的主要差异就是，较早版本将强制process()的覆盖版本必须返回Grain，而不能返回
16 | Wheat，尽管Wheat是从Grain导出的。协变返回类型允许返回更具体的Wheat类型。
*/
```

7. 向下转型与运行时类型识别

```
1 | java语言中，所有的转型都会得到检查，即使是由括号形式的类型转换，在运行期时仍然会对其进行检查，以便保证它的确是我们希望的那种类型。如果不是，就会返回一个ClassCastException（类转型异常）。这种在运行期间对类型进行检查的行为称作“运行时类型识别”（RTTI）。
2 | 例如父类引用指向子类对象，如果想要访问子类扩展的方法，就可以尝试进行向下转型。
```

✓ 第九章 接口

✓ 1. 抽象类和抽象方法

1. 抽象类：包含抽象方法的类
2. 如果一个类包含一个或多个抽象方法，该类必须被限定为抽象的，否则编译器报错。
3. 如果一个抽象类不完整，那么当创建该类的对象时，编译器会报错。
4. 如果从一个抽象类继承，并想创建该新类的对象，那么就必须为基类的所有抽象方法提供方法定义。如果不这样做（可以选择不做），那么导出类便也是抽象类，且编译器将会强制我们用abstract关键字来限定这个类。
5. 我们可能会创建一个没有抽象方法的抽象类。（如果有一个类，让其包含任何abstract方法都显得没有实际意义，而且我们也想要阻止产生这个类的任何对象，那么这样做就有意义了）

✓ 2. 接口

1. 可以在interface前面添加public关键字（但仅限于该接口在与其同名的文件中被定义）。如果不添加public，则它只具有包访问权限，这样它就只能在同一个包内可用。
2. 接口也可以包含域，但是这些域隐式地是final和static的。
3. 接口中被定义的方法必须是public的，可以选择在接口中显式地将方法声明为public，但即使不这么做，它们默认也是public的。
4. 使用接口的原因：

```
1 | reason1 (核心原因) : 为了能够向上转型为多个基类类型 (以及由此而带来的灵活性) (一个类继承了多个  
2 | 接口, 这个类能够向上转型为任意一个继承的接口类型)。  
3 | reason2 (与使用抽象基类相同) : 防止客户端程序员创建该类的对象, 并确保这仅仅是建立一个接口 (引发  
出下一问题, 该用接口还是抽象类?)。  
4 | reason3: 允许同一个接口具有多个不同的具体实现。
```

5. 使用接口创建常量组(java SE5之后, 使用更强大灵活的enum来定义, 使用接口的方式意义不大)

```
1 | public interface Months{  
2 |     int  
3 |         JANUARY = 1, FEBRUARY = 2, MARCH = 3, ...., NOVEMBER = 11, DECEMBER = 12;  
4 | }
```

6. 初始化接口中的域

```
1 | 1. 在接口中定义的域 (一定是final static) 不能是“空final”，但是可以被非常量表达式初始化。他们在类  
第一次被加载时初始化 (这发生在任何域首次被访问时)。
```

7. 在打算组合的不同接口中使用相同的方法名通常会造成代码可读性的混乱，请尽量避免这种情况。

8. 该使用接口还是抽象类？

```
1 | 如果要创建不带任何方法定义和成员变量的基类, 那么就应该选择接口而不是抽象类。  
2 | 事实上, 如果知道某事物应该成为一个基类, 那么第一选择应该是使它成为一个接口。
```

9. 完全解耦 (待补充, 没看太明白)

1. 策略设计模式：创建一个能够根据所传递的参数对象的不同而具有不同行为的方法。

```
1 | 这类方法包含所要执行的算法中固定不变的部分，而“策略”包含变化的部分。  
2 | 策略就是传递进去的参数对象，它包含要执行的代码。
```

2. 适配器设计模式：将一个类的接口转成客户期望的另外一个接口。适配器模式使得原本由于接口不匹配而不能一起工作的那些类可以一起工作。
3. 工厂方法设计模式：与直接调用构造器不同，我们在工厂对象上调用的是创建方法，而该工厂对象将生成接口的某个实现的对象。（理论上，我们的代码将完全与接口的实现分离，这就使我们可以透明的将某个实现替换为另一个实现）。
4. 迭代器设计模式。

10. java中的多重继承（接口是实现多重继承的途径）

1. 将一个具体类和多个接口组合到一起时（用一个类继承这个类和这些接口），这个具体类必须放在前面，后面跟着的才是接口，否则编译器报错。例如：

```
1 | class Hero extends Action implements CanFight, CanSwim, CanFly{}  
2 | //class Hero implements CanFight, CanSwim, CanFly extends Action{}  
3 | //上面这样写编译器报错。
```

11. 嵌套接口（内部接口）：接口可以嵌套在类或其他接口中。

```
1 | 1. 像非嵌套接口一样，嵌套接口可以拥有public和“包访问”两种可视性。同时作为一种新添加的方式，接口也可以被实现为private的（接口嵌套于类中）。  
2 | 2. 接口嵌套于接口中，必须为public。  
3 | 3. 内部接口只能是static（可以省略不写，默认也是静态的），因为接口不能被实例化，只有静态的才有意义。  
4 | 例子：  
5 | //Map.java  
6 |  
7 | public interface Map {  
8 |     interface Entry{  
9 |         int getKey();  
10 |    }  
11 |  
12 |    void clear();  
13 | }  
14 |  
15 | //MapImpl.java  
16 |  
17 | public class MapImpl implements Map {  
18 |  
19 | //如果Entry不是static的，使用Entry接口还需要构造Map对象。会出现什么问题，所以编译器不允许非静态。  
20 |     class ImplEntry implements Map.Entry{  
21 |         public int getKey() {  
22 |             return 0;  
23 |         }  
24 |     }  
25 |  
26 |     @Override  
27 |     public void clear() {  
28 |         //clear  
29 |     }  
30 | }
```

✓ 第十章 内部类

可以将一个类的定义放在另一个类的定义内部，这就是内部类。

✓ 1. 如果想从外部类的非静态方法之外的任意位置创建某个内部类的对象，那么必须以“外部类名.内部类名”的方式，具体指明这个对象的类型。

✓ 2. 内部类的作用：

- 1 1. 是一种名字隐藏和组织代码的模式。
- 2 2 (最引人注目的) . 当生成一个内部类的对象时，此对象与制造他的外围对象之间就有了一种联系，所以他能访问其外围对象的所有成员，而不需要任何特殊条件。此外，内部类还拥有其外围类的所有元素的访问权。

✓ 3. 内部类自动拥有对其外围类所有成员的访问权的实现原理：

- 1 当某个对象创建了一个内部类对象时，此内部类对象必定会秘密地捕获一个指向那个外围类对象的引用。然后，在你访问此外围类的成员时，就是用那个引用来选择外围类的成员（编译器会帮助处理所有的细节）。
- 2
- 3 内部类的对象只能在与其外围类的对象相关联的情况下才能被创建（就想你应该看到的，在内部类是static类时）。构建内部类对象时，需要一个指向其外围类对象的引用，如果编译器访问不到这个引用就会报错。

✓ 4. 使用 .this 与 .new

1. 如果需要生成对外部类对象的引用，可以使用“外部类的名字.this”。这样产生的引用自动地具有正确的类型，这一点在编译期就被知晓并受到检查，因此没有任何运行时开销。例如：

```
1 public class DotThis{  
2     void f(){  
3         print("DotThis.f()");  
4     }  
5     public class Inner{  
6         public DotThis outer(){  
7             return DotThis.this;  
8         }  
9     }  
10    public Inner inner(){  
11        return new Inner();  
12    }  
13    public static void main(String[] args){  
14        DotThis dt = new DotThis();  
15        DotThis.Inner dti = dt.inner();  
16        dti.outer().f();  
17    }  
18    //output  
19    //DotThis.f()  
20 }
```

2. 有时你可能想要告知某些其他对象，去创建其某个内部类的对象。要实现此目的，必须在new表达式中提供对其他外部类对象的引用，这就需要.new语法，例如：

```
1 public class DotNew{  
2     public class Inner{}  
3     public static void main(String[] args){  
4         DotNew dn = new DotNew();  
5         //要想创建内部类对象，必须通过外部类的对象  
6         DotNew.Inner dni = dn.new Inner();  
7     }  
8 }
```

3. 在拥有外部类对象之前是不可能创建内部类对象的。也就是，要想直接创建内部类的对象，必须使用外部类的对象来创建，就像上面的程序那样。这就解决了内部类名字作用域的问题，因此不必声明（实际上也不能声明）dn.new DotNew.Inner()。

✓ 5. 在方法和作用域内的内部类

1. 可以在一个方法里面或者在任意的作用域内定义内部类，这么做的理由：

```
1 1. 如果实现了某类型的接口，于是可以创建并返回对其的引用。  
2 2. 要解决一个复杂的问题，想创建一个类来辅助你的解决方案，但是又不希望这个类是公共可用的。
```

2. 在方法的作用于内（而不是在其他类的作用于内）创建一个完整的类，这被称作局部内部类：

```
1 public class Parcel5{  
2     public Destination destination(String s){  
3         class PDestination implements Destination{  
4             private String label;  
5             PDestination(String w){label = w;}  
6             public String readLabel(){return label;}  
7         }  
8         return new PDestination(s);  
9     }  
10    public static void main(String[] args){  
11        Parcel5 p = new Parcel5();  
12        Destination d = p.destination("Tasmania");  
13    }  
14 }  
15 /*  
16 PDestination类时destination () 方法的一部分，而不是Parcel5的一部分，所以，在destination () 之外  
17 不能访问PDestination.  
*/
```

✓ 6. 匿名内部类

```

1 public class Parcel7{
2     public Contents contents(){
3         return new Contents(){
4             private int i = 11;
5             public int value(){return i;}
6         }; //注意分号标记的是表达式的结束，和别的分号作用一样
7     }
8     public static void main(String[] args){
9         Parcel7 p = new Parcel7();
10        Contents c = p.contents();
11    }
12}

```

contents()方法将返回值的生成与表示这个返回值的类的定义结合在一起。另外，这个类是匿名的，它没有名字。这种语法指的是：创建一个继承自Contents的匿名类的对象。通过new表达式返回的引用被自动向上转型为对Contents的引用。

1. 在匿名类中定义字段时，还能够对其进行初始化操作

```

1 public class Parcel9{
2     public Destination destination(final String dest){
3         return new Destination(){
4             private String label = dest;
5             public String readLabel(){
6                 return label;
7             }
8         };
9     }
10    public static void main(String[] args){
11        Parcel9 p = new Parcel9();
12        Destination d = p.destination("Tasmania");
13    }
14}
15/*
16如果定义一个匿名内部类，并希望它使用一个在其外部定义的对象，那么编译器会要求其参数引用时final的，否则报错。
17*/

```

✓ 7. 嵌套类：

1. 如果不需要内部类对象与其外围类对象之间有联系，那么可以将内部类声明为static，这通常称为嵌套类。

嵌套类意味着：

1. 要创建嵌套类的对象，并不需要其外围类的对象。
2. 不能从嵌套类的对象中访问非静态的外围类对象。

嵌套类与普通的内部类还有一个区别：普通内部类的字段与方法，只能放在类的外部层次上，所以普通的内部类不能有static数据和static字段，也不能包含嵌套类。但是嵌套类可以包含所有这些东西。

✓ 8. 为什么需要内部类?

1. 每个内部类都能独立地继承自一个（接口的）实现，所以无论外围类是否已经继承了某个（接口的）实现，对于内部类都没有影响。
2. 内部类允许继承多个非接口类型（译注：类或抽象类）

✓ 9. 内部类知识补充

1. 网址：<https://blog.csdn.net/vcliy/article/details/85235363>
- 2.

Java中静态内部类和非静态内部类有什么区别？

转载 vcliy 最后发布于2018-12-24 16:33:34 阅读数 8644 收藏

Java中的内部类是在Jdk1.1版本之后增加的，内部类是Java语言中一个比较重要的概念，如果能把内部类运用好，那么会明显增强Java程序的灵活性。

要想清楚static内部类和非static内部类的区别，首先要了解内部类的概念及特点，然后再进行一个全面的对比。

什么是内部类呢？简单的说就是在一个类的内部又定义了一个类，这个类就称之为内部类（Inner Class）。看一个简单的例子：

```
1 public class TestInner {  
2     public class Inner{  
3     }  
4     }  
5     //内部类的范围限定可以根据需要任意设定  
6     private class A_Inner{  
7     }  
8     }  
9 }  
10
```

悟空问答

内部类有以下几个主要的特点：

第一，内部类可以访问其所在类的属性（包括所在类的私有属性），内部类创建自身对象需要先创建其所在类的对象，看一个例子：

```
1 public class TestInner {  
2     private int number = 100;  
3     public class Inner{  
4         private int number = 200;  
5         public void paint(){  
6             int number = 500;  
7             //局部覆盖原则  
8             System.out.println(number);  
9             //通过this引用内部类的成员属性  
10            System.out.println(this.number);  
11            //通过外部类名加this的方式访问外部类成员属性  
12            System.out.println(TestInner.this.number);  
13        }  
14    }  
15    public static void main(String args[]){  
16        //注意创建内部类对象分为两个步骤  
17        TestInner inner = new TestInner();  
18        TestInner.Inner in = inner.new Inner();  
19        in.paint();  
20    }  
21 }
```

悟空问答

第二，可以定义内部接口，且可以定义另外一个内部类实现这个内部接口，看一个例子：

```
1 public class TestInner {  
2     //内部接口  
3     public interface Fly{  
4         void doFly();  
5     }  
6     public class Inner implements Fly{  
7         @Override  
8         public void doFly() {  
9             System.out.println("i'm SuperMan");  
10        }  
11    }
```

```
11
12 ►     public static void main(String args[]){
13         //注意创建内部类对象分为两个步骤
14         TestInner inner = new TestInner();
15         TestInner.Fly in = inner.new Inner();
16         in.doFly();
17
18     }
19 }
```

悟空问答

第三，可以在方法体内定义一个内部类，方法体内的内部类可以完成一个基于虚方法形式的回调操作，看一个例子：

```
1 ► public class TestInner {
2     public class Inner{
3         public M_Car getCar(){
4             //方法体内的内部类
5             class BMW extends M_Car{
6                 @Override
7                 public void paint() {
8                     System.out.println("BMW");
9                 }
10            }
11            return new BMW();
12        }
13    }
14 ►     public static void main(String args[]){
15         //注意创建内部类对象分为两个步骤
16         TestInner inner = new TestInner();
17         TestInner.Inner in = inner.new Inner();
18         in.getCar().paint();
19     }
20 }
21 class M_Car{
22     public void paint(){
23         System.out.println("car");
24     }
25 }
```

悟空问答

第四，内部类不能定义static元素，看一个例子：

```
1 ► public class TestInner {
2     public class Inner{
3         public static void paint(){
4             //Inner classes cannot have static declarations
5         }
6     }
7 }
8 ►     public static void main(String args[]){
9         //注意创建内部类对象分为两个步骤
10        TestInner inner = new TestInner();
11        TestInner.Inner in = inner.new Inner();
12    }
13 }
```

悟空问答

第五，内部类可以多嵌套，看一个例子：

```
1 ► public class TestInner {
2     public class Inner{
3         //又定义了一个内部类
4         public class AInner{
5
6         }
7     }
8 ►     public static void main(String args[]){
9         //注意创建内部类对象的多个步骤
10        TestInner inner = new TestInner();
11        TestInner.Inner in = inner.new Inner();
12        //按层次创建
13        TestInner.Inner.AInner ain = in.new AInner();
```

```
14 }  
15 }
```

悟空问答

static内部类是内部类中一个比较特殊的情况，Java文档中是这样描述static内部类的：一旦内部类使用static修饰，那么此时这个内部类就升级为顶级类。

也就是说，除了写在一个类的内部以外，static内部类具备所有外部类的特性，看一个例子：

```
1 public class TestInner {  
2     //定义static内部类  
3     public static class Inner{  
4         //static内部类可以定义static元素  
5         public static void paint(){  
6             System.out.println("inner?");  
7         }  
8     }  
9     public static void main(String args[]){  
10        //注意创建static内部类对象的步骤  
11        TestInner.Inner inner = new TestInner.Inner();  
12        inner.paint();  
13    }  
14 }
```

悟空问答

通过这个例子我们发现，static内部类不仅可以在内部定义static元素，而且在构建对象的时候也可以一次完成。从某种意义上说，static内部类已经不算是严格意义上的内部类了。

与static内部类不同，内部接口自动具备静态属性，也就是说，普通类是可以直接实现内部接口的，看一个例子：

```
1 public class TestInner {  
2     public interface Fly{  
3         void doFly();  
4     }  
5 }  
6 class SuperM implements TestInner.Fly{  
7     @Override  
8     public void doFly() {  
9     }  
10 }  
11 }
```

悟空问答

第十二章 通过异常处理错误

✓ 1. 能够抛出任意类型的Throwable对象，它是异常类型的根类。

✓ 2. 捕获异常

1. try块与异常处理程序

```
1 | try{  
2 |     //code  
3 | }catch (Type1 a){  
4 |     //code  
5 | }catch (Type2 a){  
6 |     //code  
7 | }
```

2. 异常与记录日志

java.util.logging

3. 异常说明

```
1 | 0. 关键字throws  
2 | 1. 异常说明属于方法声明的一部分，后面接一个所有潜在异常类型的列表，多个异常类型用逗号分隔。例如  
3 | void f() throws E1, E2, E3{//...}  
4 | 2. 从RuntimeException继承的异常，可以在没有说明的情况下被抛出。  
5 | 3. 如果一个函数声明部分有异常说明，那么在调用该函数时，必须放在try块里面，并catch异常类型。  
6 | 4. 方式：可以声明方法将抛出异常，实际上却不抛出。编译器相信了这个声明，并强制此方法的用户像真的  
7 |     抛出异常那样使用这个方法。  
8 |     好处：这样做的好处是，为异常先占个位子，以后就可以抛出这种异常而不用修改已有的代码。  
     用途：定义抽象基类和接口时，很重要，这样派生类或接口实现就能抛出这些预先声明的异常。
```

4. 使用finally进行清理

```
1 | 格式：  
2 | try{...}  
3 | catch(...){...}  
4 | catch(...){...}  
5 | finally{...}  
6 | 特点：1. finally里的代码总能运行，无论异常是否被抛出。  
7 |     2. 当涉及break和continue语句的时候，finally子句也会得到执行。  
8 |     3. 在函数return代码之后的finally也仍会执行。  
9 | 用途：当要把除内存之外的资源恢复到他们的初识状态时，就要用到finally子句。这种需要清理的资源包括：  
     已经打开的文件或网络连接，在屏幕上画的图形，甚至可以是外部世界的某个开关。
```

✓ 3. 异常的限制

```
1 | package ThinkInJava.MyException;  
2 |  
3 | class BaseBallException extends Exception{}  
4 | class Foul extends BaseBallException{}
```

```
5 class Strike extends BaseBallException{}
6
7 abstract class Inning{
8     public Inning() throws BaseBallException{}
9     public void event() throws BaseBallException{}
10    public abstract void atBat() throws Strike, Foul;
11    public void walk(){}
12 }
13
14 class StormException extends Exception{}
15 class RainedOut extends StormException{}
16 class PopFoul extends Foul{}
17
18 interface Storm{
19     public void event() throws RainedOut;
20     public void rainHard() throws RainedOut;
21 }
22
23 public class StormyInning extends Inning implements Storm{
24     //异常限制对构造器不起作用，此构造器可以抛出任何异常，而不用理会基类所抛出的异常。
25     //然而，因为基类构造器必须以这样或那样的方式被调用，所以，派生类构造器的异常说明必须包含基类构造器的异常说明。
26     //派生类构造器不能捕获基类构造器抛出的异常
27     public StormyInning() throws RainedOut, BaseBallException{}
28     public StormyInning(String s) throws Foul, BaseBallException{}
29     /* 因为多态的存在，调用Inning.walk()的时候不用做异常处理，而当把StormyInning强制转换为Inning对象时
30     * 可能会抛出异常，于是程序就失灵了。所以，通过强制派生类遵守基类方法的异常说明，对象的可替换性得到了
31     * 保证。
32     */
33     //void walk() throws PopFoul{} //编译错误，因为方法覆盖时，基类的walk方法不会抛出异常
34     //接口不能向基类中的现有方法添加异常
35     //public void event() throws RainedOut{}
36     //如果event这个方法不是覆盖的基类的方法，上面的代码是对的。
37     /* 如果一个类（StormyInning）在扩展基类（Inning）的同时又实现了一个接口（Storm），那么Storm里的
38     event（）
39     * 方法就不能改变在Inning中的event（）方法的异常接口。否则的话，在使用基类的时候就不能判断是否捕获
40     * 了正确的
41     * 异常，所以这很合理。当然，如果接口里定义的方法不是来自于基类，比如rainHard，那么此方法抛出什么样的
42     * 异常都
43     * 没有问题。
44     */
45     public void rainHard() throws RainedOut{} //不能抛出实现的接口所规定的异常之外的异常。
46     //public void rainHard() throws Foul{} //错误
47     //重写方法只能抛出继承的异常，但也可以不抛出异常，即使它是基类所定义的异常，因为这样不会破坏已有程
48     //序。
49     public void event(){ //只能抛出基类和接口的共同的异常类型。因为没有共同的（当然，肯定有
50     RuntimeException异常），所以次函数不能抛出异常
51     public void atBat() throws PopFoul{} //正确，基类会抛出Strike, Foul的异常，而PopFoul继承自Foul，基类
52     肯定会捕获到这个异常。
53     //知识点：异常说明本身不属于方法类型的一部分，方法类型是由方法的名字与参数的类型组成的。因此，不能
54     基于异常说明来重载方法。
55
56     public static void main(String[] args) {
57         try{
58             StormyInning si = new StormyInning();
59             si.atBat();
60         }catch (PopFoul e){
61             System.out.println("pop foul");
62         }catch(RainedOut e){
63             System.out.println("rained out");
64         }catch(BaseBallException e){
65             System.out.println("Generic baseball exception");
66         }
67     }
68 }
```

```
61 }catch (Strike e){  
62     System.out.println("Strike");  
63 }catch (Foul e){  
64     System.out.println("Foul");  
65 }catch (RainedOut e){  
66     System.out.println("Rained out");  
67 }catch (BaseBallException e){  
68     System.out.println("Generic baseball exception");  
69 }  
70 }  
71 }  
72 }
```

✓ 4. 异常匹配

1. 抛出异常的时候，异常处理系统就会按照代码的书写顺序找出“最近”的处理程序。找到匹配的处理程序之后，它就认为异常将得到处理，然后就不再继续查找。
2. 查找的时候不要求抛出的异常同处理程序所声明的异常完全匹配。派生类的对象也可以匹配其基类的处理程序。
3. 如果把捕获基类异常的catch子句，放在捕获派生类异常的catch子句之前，以此想把派生类的异常完全给“屏蔽”掉，这样编译器会发现E2的catch子句永远也得不到执行，因此报错。例如：

```
1 class E1 extends Exception{}  
2 class E2 extends E1{}  
3 public class ExceptionMatch {  
4     public static void main(String[] args) {  
5         try{  
6             throw new E2();  
7         }catch (E1 e){//此代码也会捕获到E2类的异常，所以下面这句永远都不会执行。编译错误。  
8         catch (E2 e){  
9         }  
10    }
```

✓ 5. 异常使用指南

1. 在恰当的级别处理问题。（在知道该如何处理的情况下才捕获异常。）
2. 解决问题并且重新调用产生异常的方法。
3. 进行少许修补，然后绕过异常发生的地方继续执行。
4. 用别的数据进行计算，以代替方法预计会返回的值。
5. 把当前运行环境下能做的事情尽量做完，然后把相同的异常重抛到更高层。
6. 把当前运行环境下能做的事情尽量做完，然后把不同的异常抛到更高层。
7. 终止程序。
8. 进行简化。（如果你的异常模式使问题变得太复杂，那用起来会非常痛苦也很烦人。）
9. 让类库和程序更安全。（这既是在为调试做短期投资，也是在为程序的健壮性做长期投资。）

第十三章 字符串

✓ 1. 不可变String

1. String对象时不可变的。String类中每一个看起来会修改String值的方法，实际上都是创建了一个全新的String对象，以包含修改后的字符串内容。而最初的String对象则丝毫未动。

```
1 package ThinkingInJava.MyString;
2
3 public class Immutable {
4
5     public static void main(String[] args) {
6         String s = new String("aaa");
7         System.out.println(System.identityHashCode(s));
8         s = new String("bbb");
9         System.out.println(System.identityHashCode(s));
10        s = new String("aaa");
11        System.out.println(System.identityHashCode(s));
12        s = "aaa";
13        System.out.println(System.identityHashCode(s));
14        s = "bbb";
15        System.out.println(System.identityHashCode(s));
16        s = "aaa";
17        System.out.println(System.identityHashCode(s));
18    }
19 }
20 ****Output*****
21 284720968
22 189568618
23 793589513
24 1313922862
25 495053715
26 1313922862
27 *****End*****
```

✓ 2. StringBuilder

1. 编译器自动引入java.lang.StringBuilder类。对于

```
1 String mango = "mango";
2 String s = "abc" + mango + "def" + 47;
3 //编译器创建了一个StringBuilder对象，用以构造最终的String，并为每个字符串调用一次StringBuilder的
  append () 方法，总计四次。最后调用toString () 方法生成结果，并存为s。
```

编译器会自动使用StringBuilder类，因为它更高效。

2. 显式创建StringBuilder还允许预先指定其大小。可以避免因空间不够而多次重新分配缓冲。

- 当为一个类编写`toString()`方法时，如果字符串操作比较简单，那就可以信赖编译器，他会为你合理地构造最终的字符串结果。但是，如果要在`toString()`方法中使用循环，那么最好自己创建一个`StringBuilder`对象，用它来构造最终的结果。
- 如果想走捷径，例如`append("aaa" + "bbb" + str)`，那么编译器就会调入陷阱，从而为你另外创建一个`StringBuilder`对象处理括号内的字符串操作。

✓ 3. String成员方法

- `charAt()`方法：传递 int 索引，返回该索引位置的char
- `getChars(),getBytes()`方法：传递要复制部分的起点和终点的索引，复制到的目标数组，目标数组的起始索引；

```
1 | String s1 = "abcdefg";
2 | char[] arrChar = new char[10];
3 | s1.getChars(0, 4, arrChar, 0);
4 | //此时arrChar = ['a' 'b' 'c' 'd']
5 | //复制的元素不包含终点索引位置的那个元素
```

- `toCharArray()`方法：无参函数，返回一个`char[]`，包含String的所有字符。

- `equals(), equalsIgnoreCase()`方法。

- `compareTo ()` 方法。

- `contains ()` 方法：

- 注意点：当需要改变字符串的内容时，`String`类的方法都会返回一个新的`String`对象。同时，如果内容没有发生改变，`String`的方法只是返回指向原对象的引用而已。这可以节约存储空间以及避免额外的开销。

✓ 4. 格式化输出

- `System.out.printf()` 和 `System.out.format()`这两个等价，并和c语言的`printf`用法一样。
- `Formatter`类：java中，所有新的格式化功能都由`java.util.Formatter`类处理。
- `String.format()`

```
1 | 1. java SE5参考了c中的sprintf () 方法，以生成格式化的String对象。它接受与Formatter.format()方法一样的参数。
2 | 2. 当你只需使用format () 方法一次的时候，String.format () 用起来很方便。
3 | 3. 其实，String.format()内部，它也是创建一个Formatter对象，然后将传入的参数转给该Formatter。不过，预期自己做这些事情，不如使用便捷的String.format()方法，何况这样的代码更清晰易读。
```

✓ 5. 正则表达式

第十四章 类型信息

第二十二章 图形化用户界面

附一 GOF23设计模式

创建型模式：单例模式、工厂模式、抽象工厂模式、建造者模式、原型模式。

结构型模式：适配器模式、桥接模式、装饰模式、组合模式、外观模式、享元模式、代理模式。

行为型模式：模板方法模式、命令模式、迭代器模式、观察者模式、中介者模式、备忘录模式、解释器模式、状态模式、策略模式、职责链模式、访问者模式。

✓ 1. 单例设计模式

优点：①由于单例模式只生成一个实例，减少了系统性能开销，当一个对象产生需要比较多的资源时，如读取配置、产生其他依赖对象时，则可以通过在应用启动时直接产生一个单例对象，然后永久驻留内存的方式来解决。

②单例模式可以在系统设置全局的访问点，优化共享资源访问，例如可以设计一个单例类，负责所有数据表的映射处理。

1. 饿汉式（线程安全，调用效率高。但是，不能延迟加载）

```
1 public class SingletonDemo2{  
2     private static (final) SingletonDemo2 instance = new SinletonDemo2(); //类初始化时，立即加载这  
3     //个对象  
4     private SingletonDemo2(){} //私有化构造器  
5     public static SingletonDemo2 getInstance(){ //方法没有同步，调用效率高  
6         return instance;  
7     }  
8 }  
9 /**  
10  * ① 饿汉式单例模式代码中，static变量会在类装载时初始化，此时不会涉及多个线程对象访问该对象的问  
11  * 题。虚拟机保证只会装载一次该类，肯定不会发生并发访问的问题。因此getInstance方法可以省略  
* synchronized关键字，提高效率。  
12  * ② 问题：如果只是加载本类，而不是要调用getInstance ()，设置永远没有调用，则会造成资源浪费。  
13 */
```

2. 懒汉式（线程安全，调用效率不高【因为使用了互斥锁】。但是，可以延迟加载）

```
1 public class SingletonDemo1{
2     private static (final) SingletonDemo2 instance;//类初始化时，不加载这个对象，等到使用时，再去
3     加载。
4     private SingletonDemo1(){}//私有化构造器
5     public static synchronized SingletonDemo1 getInstance(){}//方法同步，线程安全，但调用效率低
6         if (null == instance)
7             instance = new SingletonDemo1();
8         return instance;
9     }
10    /**
11     ① lazy load, 延时加载，懒加载，真正用的时候才加载，资源利用率高。
12     ② 问题：每次调用getInstance()方法都要同步，并发效率较低。
13 */

```

3. 双重检测锁实现（由于JVM底层内部模型原因【指令重排】，偶尔会出问题。不建议使用）

```
1 public class SingletonDemo3{
2     private static (final) SingletonDemo3 instance = null;
3     private SingletonDemo3(){}//私有化构造器
4     public static SingletonDemo3 getInstance(){
5         if (null == instance){
6             SingletonDemo3 s;
7             synchronized (SingletonDemo3.class){
8                 s = instance;
9                 if (null == s){
10                     synchronized (SingletonDemo3.class){
11                         if (null == s)
12                             s = new SingletonDemo3();
13                     }
14                     instance = s;
15                 }
16             }
17         }
18         return instance;
19     }
20 }
21 /**
22 ① 双重检测锁将同步内容放到if内部，提高了执行效率，不必每次获取对象都进行同步，只有第一次才同
步。
23 ② 问题：由于编译器优化原因和JVM底层内部模型原因，偶尔会出现问题（指令重排）。
24 */

```

4. 静态内部类实现方式（线程安全，调用效率高，可以延迟加载）

```
1 public class SingletonDemo4{
2     private static class SingletonClassInstance{
3         private static (final) SingletonDemo4 instance = new SingletonDemo4();
4     }
5     private SingletonDemo4(){}//私有化构造器
6     public static SingletonDemo4 getInstance(){}//方法没有同步，调用效率高
7         return instance;

```

```

8    }
9 }
10 /**
11 ① 外部类没有static属性，则不会像饿汉式那样立即加载对象
12 ② 只有真正调用getInstance方法时，才会加载静态内部类。加载类时是线程安全的。instance是static
13 final类型，保证了内存中只有这样一个实例存在，而且只能被赋值一次，从而保证了线程安全性。
14 */

```

5. 枚举式（不需延迟加载时，强烈推荐）（线程安全，调用效率高，但是，不能延迟加载）

```

1 public enum SingletonDemo5{
2     // 定义了一个枚举元素，它就代表了Singleton的一个实例
3     INSTANCE;
4     // 单例可以有自己的操作
5     public void singletonOperation(){}
6 }
7 /**
8 ① 实现简单
9 ② 枚举本身就是单例模式。由JVM从根本上提供保障，避免通过反射和反序列化的漏洞
10 ③ 问题：无延迟加载
11 */

```

1) 如何选用哪种单例模式？？？

1. 单例对象占用资源少，不需要延时加载：

枚举式 好于 饿汉式

2. 单例对象占用资源大，需要延时加载：

静态内部类式 好于 懒汉式

2) 问题

1. 反射可以破解上面几种（除了枚举式）实现方式。

解决：在构造方法中手动抛出异常控制

2. 反序列化可以破解上面几种（除了枚举式）实现方式。

解决：可以通过定义readResolve()方法防止获得不同对象。

```

1 // 反序列化时，如果对象所在的类定义了readResolve()，（实际上是一种回调），定义返回哪个对
象。（貌似不是重写方法，但格式必须这样）
2 public class SingletonDemo1 implements Serializable{
3     private static SingletonDemo1 s;
4     private SingletonDemo1() throws Exception{
5         if (null != s){
6             throw new Exception("只能创建一个对象");
7             //避免通过反射创建多个对象。
}

```

```
8    }
9 }
10 public static synchronized SingletonDemo1 getInstance() throws Exception{
11     if (null == s){
12         s = new SingletonDemo1();
13     }
14     return s;
15 }
16 //反序列化时，如果对象所在类定义了readResolve()。 (实际是一种回溯)。 定义返回哪个对
象。
17 private Object readResolve() throws ObjectStreamException{
18     return s;
19 }
20 }
```

3) 效率

1 | 1. 懒汉式效率最低，大约慢25倍

2. 双重检查锁式其次，大约慢2倍
3. 其余差不多