# Finalize、final、finally三者的区别

## finalize

<http://blog.csdn.net/aimeimeits/article/details/78252881>

GC 判断一个对象的引用不再被其他对象持有（不再有任何未死亡的线程通过任何方法调用来访问该对象（或访问该对象的成员））时，finalize方法将被调用。

finalize中可以执行任何操作（包括使该对象再次可用），但**finalize中应该始终执行与资源回收，清理相关的操作**。

finalize被调用后对象所占的空间不会立即被回收，而是在GC**下一次进行回收操作时回收**。

finalize抛出异常时GC将取消对该对象后续的回收动作。

## final

<http://blog.csdn.net/aimeimeits/article/details/76473101>

final可以修饰类、方法、变量和方法参数，指“这是无法改变的”。

1. 修饰**基本数据类型**（编译期常量）

在定义时就赋值，编译器可以在编译时将常量代入任何用到它的计算式中，可以在编译期执行计算，减轻运行时负担。

1. 修饰**引用数据类型**

被修饰变量的引用不能改变，但**变量本身的属性可以改变**。（同时被final和static修饰时指该变量初始化之后只会占据着一份内存空间）

1. **空白final**：声明为final但在声明时不赋值的成员变量

静态空白final只能在其中一个静态代码块中赋值，非静态空白final只能在其中一个构造代码块中赋值或在**每一个构造器**中赋值。

1. final修饰**方法参数**

final修饰方法参数意味着**无法在方法体中修改参数的值**（基本数据类型）或引用（引用数据类型），这一特性的重要用途之一是使参数直接被方法体中的匿名内部类（或Lambda表达式）使用。（方法中的匿名内部类编译为class文件时当独存放，为保证闭包内数据一致性将数据声明为final，并通过匿名内部类的构造器（编译时自动生成）传递给匿名内部类）

1. final修饰**方法**

把方法锁定，以**防止任何导出类重新定义方法（覆写）**，过去还由于性能因素使用final修饰方法（使之成为内嵌调用），现在不再需要了。只有在想明确禁止覆盖时，才应该使用final修饰方法。

1. final**类**

将类修饰为final，表明**类为最终类，无法被继承**。final类的域仍然可以自己决定是否成为final，但所有的方法都隐式为final。

应该谨慎考虑是否将类定义为final的，因为要预见类时如何被复用一般是很困难的，特别是对于一个通用的类。一旦将类定义为final，即表明类的功能仅限于此，也就没有了扩展性。

final和private：**所有的private方法都隐式指定为final。**

## finally

<https://www.ibm.com/developerworks/cn/java/j-lo-finally/>

finally语句块得到执行的条件：

1. 相应的try块的到执行；
2. 虚拟机关闭时（System.exit(state)）时无法执行；
3. 所在线程退出时无法执行。

当try、catch、finally都有return时，finally语句块会在控制转移语句（return、throw、break、continue（前两者将控制权转交给调用者））之前执行，返回值（有的话）会被缓存（新的return值会覆盖旧的，return的值会被保存到本地变量中（注意是值，只有return关键字才会促使新值覆盖旧值）），finally结束时才会跳转。

# 匿名内部类

***能且仅能继承一个类或实现一个接口***

<http://blog.csdn.net/chenssy/article/details/13170015>

匿名内部类必须（也只能）继承一个父类（父类为public类时若只有一个私有的构造器则不能,父类为final类时不能导出）或实现一个接口，**匿名内部类相当于导出一个新类并立即创建一个它的对象并使用**。

**缺点**：只能使用一次，使用外部类的局部变量时局部变量必须为final的（**外部类的局部变量以参数形式传递给内部类，外部类的成员变量通过this关键字传递**）

匿名内部类不能定义构造函数，构造函数的功能可使用构造代码块替代。

匿名内部类继承抽象类或实现接口时必须实现抽象类或接口的所有抽象方法。

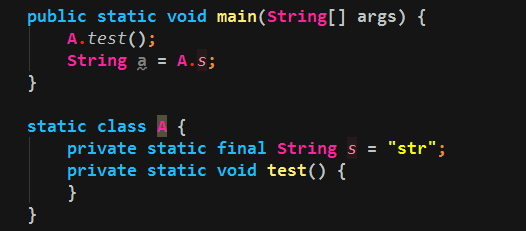
局部变量为何为final：拷贝引用，为了避免引用值发生变化，导致内外不一致（因为在内部类中使用局部变量时是直接使用的，它们任何时候“看起来”都应该是“同一对象”，对它的修改应该内外保持一致）。

# 静态内部类和普通内部类（非静态内部类）

<http://www.cnblogs.com/chenssy/p/3388487.html>

**普通内部类可以直接访问外部类的属性，方法，即使是private修饰的内部类也可以。**

静态内部类不依赖外部类，静态内部类给外部类提供的语法糖之一为，外部类可以直接访问静态内部类的静态成员，priva/protected的都可以：javac会生成对应的“access method”（access$XXX）方法（类似getter、setter）。



静态内部类不会持有外部类的引用。

普通内部类不能声明静态成员。

在非外部类中创建普通内部类的对象需要先创建外部类的对象：

有类ClassB和ClassBInner，在ClassA中创建ClassBInner对象：

ClassB cb = new ClassB();

ClassBInner cbinn = cb.new ClassBInner(); // CLassBInner不能为private（符合一般可见性规则）

创建可访问的静态内部类：new OutClass.InnerClass();

# & 与 &&，| 与 ||

双与双或具有短路效果；单与单或没有，作为算数运算符时为二进制按位与和按位或。

单与单或作为算数运算符。

1011 1011

& 0101 | 0101

----------- ---------

0001 1111

# HashMap 和 HashTable 的区别

<http://www.importnew.com/7010.html>

HashTable继承于Dictionary，**实现Map接口**；

HashMap继承于AbstractMap（实现Map），同时自己也实现了Map接口；

两者主要不同体现在：**线程安全性，同步以及速度**。

HashMap可以接受一个null键和多个null值，HashTable不能（值和键都不能）。

HashMap是非synchronized的，而HashTable是同步的。这意味着多个线程可以共享同一个HashTable，而如果没有正确同步的话，多个线程不能使用同一个HashMap。（java 5提供了ConcurrentHashMap，它是HashTable的替代，它解决了当HashTable变得很大时性能极具下降的问题（速度变慢）<http://www.importnew.com/7166.html> ）。

HashMap的迭代器（HashMap.entrySet.isterator）在迭代过程中如果有其它线程改变数据（增加或者移除），将会抛出ConcurrentModificationException（但迭代器本身的remove方法不会抛）。

单线程时无需进行同步，所以在单线程环境下HashTable要比HashMap慢。

使用Collections.synchronizeMap(HashMap)可将HashMap包装为同步数据结构。

Hashtable和HashMap有几个主要的不同：线程安全以及速度。仅在你需要完全的线程安全的时候使用Hashtable，而如果你使用Java 5或以上的话，请使用ConcurrentHashMap。

# String 是基本数据类型吗？基本类型有哪些？

String不是基本数据类型

char-1 byte-1 short-2 int-4 float-4 long-8 double-8 boolean-?

《java虚拟机规范》：boolean占4个字节（编译之后int代替），boolean数组占1个字节（编译后为byte数组）

# String为什么被设计为final

<http://www.jianshu.com/p/2e513e872cb4>

String类中任何一个试图改变String对象的方法其实都生成了一个新的String对象。

1. java 方法区中的字符串常量池的需要
2. 用作缓存的hashcode，HashMap，HashSet中用String作为键时无需担心String变化（因为String不可变，其hashcode也不可变），更加高效。
3. 便利其它对象，安全问题，打开文件、网络连接、网络传输、反射。
4. 天生的线程安全。

# 字符串“+”与StringBuilder

String中使用的“+”，编译器会用StringBuilder替换，用于String的“+”和“+=”是java中仅有的两个重载过的操作符，而java不允许程序员重载其它操作符。

**在循环中应避免使用String的“+”号**，会new出很多的StringBuilder对象。

不要在StringBuilder.append中使用“+”

StringBuffer线程安全（开销大），StringBuilder适用于单线程字符串拼接。

覆写toString方法时不能使用this关键字，会产生死递归调用（StackOverFlow）。

# OOP 三大基本特性，六大原则。

## 三大特性



### 封装：隐藏了类的内部实现

把客观事物封装成抽象的类，类可以隐藏自己的属性，只让自己的部分属性和方法对外可见；简单的说，**一个类就是一个封装了数据以及操作这些数据的代码的逻辑实体**，在一个对象内部，一些方法和属性可以是私有的，不能被外界访问，通过这种方法，对象对内部的数据提供了不同级别的保护，以防止程序中无关的部分意外的改变或错误使用对象的私有部分。

### 继承：为了重用父类的代码

可以**让某个类型的对象获得另一个类型的属性和方法**，继承的过程就是从一般到特殊的过程，继承有两种途径：实现继承和接口继承。

### 多态：动态绑定

<http://blog.csdn.net/huangrunqing/article/details/51996424>

**一个实例的相同方法在不同情形下有不同的表现形式**（提供不同的输入会有不同的输出），虽然针对不同对象的具体操作不同，但通过一个公共的类，那些操作可以通过相同的方式予以调用。

## 六大原则：SOLID+迪米特原则

<http://www.jianshu.com/p/e697df67df38>

### S：Single Responsibility Principle（单一职责）

就一个类而言，应该仅有一个引起它变化的原因，一个类中应该是一组相关性很高的函数。

### O：Open Close Principle（开闭）

软件中的对象（类，模块，函数）**对扩展来说是开发的，对修改是封闭的**，在软件的生命周期中，因为变化、升级和维护等原因对软件原有代码进行修改时，可能会引入新的错误，破坏系统。当软件有需求变化时，应该尽量通过扩展的方式实现，而不是修改已有代码。

### L：Liskov Substitution Principle（里式替换）

**所有引用基类的地方必须能透明的使用其子类对象**。只要父类能出现的地方子类就能出现，而且替换成子类也不会产生任何错误，里式替换原则依赖于抽象。

### I：Interface Segreation Principle（接口隔离）

客户端不应该依赖于它不需要的接口，一个类对另一个类的依赖应该建立在最小的接口上。

### D：Dependence Inversion Principle（依赖倒置）

**高层模块不应该依赖于底层模块，两者都应该依赖于抽象，抽象不应该依赖于细节，细节应该依赖于抽象。**模块间的依赖关系通过抽象发生，实现类之间不应该发生直接的依赖关系。（最少知道）

### 迪米特原则

**一个对象应该对其它对象有最小的了解**，或者一个类应该对自己需要耦合的类知道的最少，类的内部实现与调用者无关，调用者只需知道调用接口就可以了。

# 重写与重载的区别

## 重载（Overload）

每个重载方法都**必须有一个独一无二的参数列表（个数，顺序，类型）**，不能根据返回值来区分重载方法，因为有时方法调用并不是为了得到返回结果，而是为了其“副作用”。

## 重写（Override，覆写）

继承时非private的方法才能被覆盖，**重写方法和父方法有相同的方法名和参数列表。**

**返回值类型可为父方法返回类型的导出类（协变性）**，**可见性不小于父方法，可抛出父方法异常的子异常，个数无限制。**

# List，Set和Map的区别

<http://blog.csdn.net/vstar283551454/article/details/8682655>

\* java集合框架：<http://www.importnew.com/16658.html>

