# 语法

## final、finally和finalize

### Final修饰符

#### 修饰数据（变量被声明为final）

只能进行一次赋值操作，声明时给定初始值，在以后的引用中只能读取，不可修改。

#### 修饰方法参数

在整个方法中，我们不会（实际上是不能）改变参数的值。

#### 修饰方法

用final关键字修饰方法，它表示该方法不能被覆盖。这种使用方式主要是从设计的角度考虑，即明确告诉其他可能会继承该类的程序员，不希望他们去覆盖这个方法。这种方式我们很容易理解，然而，关于private和final关键字还有一点联系，这就是类中所有的private方法都隐式地指定为是final的，由于无法在类外使用private方法，所以也就无法覆盖它。

#### 类被声明为final

不能再派生出新的子类，不能作为父类被继承。

### Finally

异常处理中提供finally块来执行任何清除操作。

### Finalize

object类中的一个方法，子类可以重写finalize()方法实现对资源的回收。

finalize()方法是在垃圾收集器删除对象之前对这个对象调用的。

## assert

### 两种形式

#### assert 表达式

#### assert 表达式1 ： 表达式2

### 如何在编译时启用断言

Javac -source 1.4 Test.java

### 如何在运行时启用断言

-enableassertions或者-ea

### 如何在运行时禁用断言

-disableassertions或者-da

### 如何在系统类中启用断言

-esa或者-dsa

## abstract class和interface

### Abstract class

#### 特点

不能创建实例。

不能有抽象构造函数。

不能有抽象静态方法。

可实现接口。

可继承实体类，但前提是实体类必须有明确的构造函数。

#### 抽象方法

不能是static。

不能是native。

不能是synchronized。

### Interface

#### 特点

抽象类的变体。

所有方法都是抽象的。

可以定义static final成员变量。

可继承接口。

## Synchronized

### 参考资料

<http://www.importnew.com/21866.html>

### 修饰一个代码块

Public void function() {

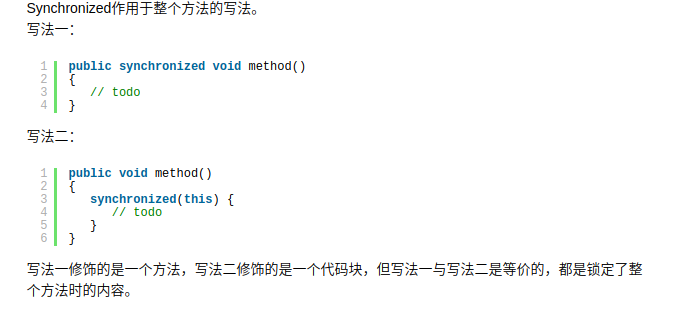
Synchronized (this) {

}

}

当一个线程访问对象的一个synchronized(this)同步代码块时，另一个线程仍然可以访问该对象中的非synchronized(this)同步代码块。

### 修饰一个方法



synchronized关键字不能继承。

### 修饰一个静态的方法

public synchronized static void method() {}

静态方法是属于类的而不属于对象的。同样的，synchronized修饰的静态方法锁定的是这个类的所有对象。

### 修饰一个类

class ClassName {

public void method() {

synchronized(ClassName.class) {

}

}

}

# 运算符

## 参考资料

<http://www.runoob.com/java/java-operators.html>

## 算术运算符



## 关系运算符



## 位运算符



### 原码、反码和补码

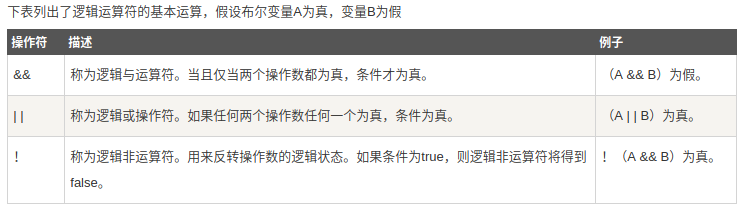
原码：第一位为符号位（符号位表示0为正数，1为负数）

负数反码：符号位不管，原码取反

负数补码：符号位不管，反码加1

正数补码：和原码相同。

## 逻辑运算符



## 赋值运算符



## 其他运算符

### 条件运算符（?:）

### instanceof 运算符

## 运算符优先级



# 类（Class）

## 参考资料

<https://blog.csdn.net/a327369238/article/details/52780442>

## 文件类

### 定义

文件类放在最前面，是因为文件类是与主类（一个文件中public类）关系最不密切的一类。什么是文件类？看代码就知道：

public class Main{}

class Test{}//Test就是文件类

//是的，一个.java文件里面定义在主类外面的就是文件类

//主类、文件类称为顶级类（top level class），Java语言规范中定义：非嵌套类即为顶级类。

【注意】：主类这一定义是我自己按语义称呼的，有的地方称为基本类，但我觉得很不符合语义，Java语言规范我也没找到相关定义。

因为一个.java文件只能有一个主类（public 类），所以文件类默认只能是包访问权限，即：不是同一个包的是无法引入和使用的。

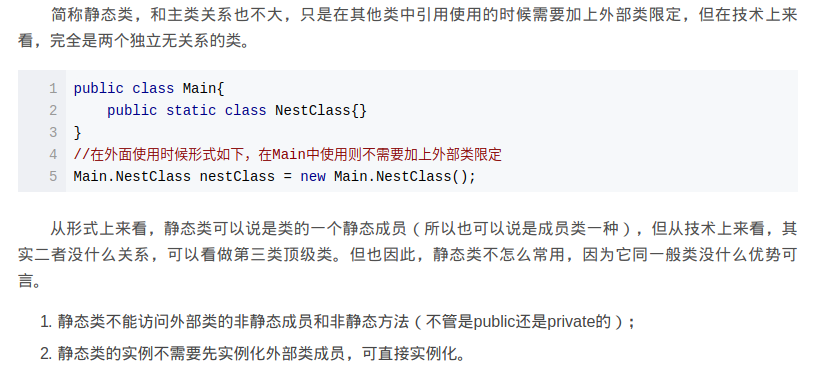
## 嵌套类

### 定义

定义在类（这里还包括接口，下同）里面的类。所以说，以下所有的类都可以称为嵌套类。嵌套类分为两种：静态嵌套类和非静态嵌套类，非静态嵌套类就是内部类（inner class）。

### 静态嵌套类

#### 定义



### 内部类/非静态嵌套类

#### 定义

非静态嵌套类即为内部类。

内部类包括：成员类、局部类、匿名类。

内部类中不能有静态修饰的成员（比如块、字段、方法、接口等），总之不能有static关键字，除了一种情况，那就是静态常量，又因为常量成员字段必须在声明的时候初始化，所以形式只能如：public static final int a = 6;

内部类可以访问外部类任何成员，不管是公有的还是私有的，静态的还是非静态的（并且内部类的成员的名字也可以同外部相同，只不过这样会覆盖掉去外部类的），这是因为每一个内部类都保存了一个对外部类的一个引用。这很好理解，因为你要实例化这个内部类，肯定是通过外部类的一个实例，而内部类保留的这个引用就是这个外部类实例。

内部类命名格式：外部类名称+$+[该种类同名类中该类顺序]+[内部类名称]，例如成员类，成员类不能同名，所以也就没有同名类顺序：com.fcc.test.OuterClass$MemberClass；局部类：com.fcc.test.OuterClass$1LocalClass；匿名类：匿名类没有名称，所以格式如：com.fcc.OuterClass$1。匿名类根据位于地方不同分为：成员匿名类和局部匿名类。

## 内部类

### 参考资料

<https://www.cnblogs.com/lgk1002/p/6069784.html>

### 成员内部类/在一个类（外部类）中直接定义的内部类

#### 定义

作为外部类的一个成员存在，与外部类的属性、方法并列。

成员内部类也是定义在另一个类中，但是定义时不用static修饰。

成员内部类和静态内部类可以类比为非静态的成员变量和静态的成员变量。

成员内部类就像一个实例变量。

它可以访问它的外部类的所有成员变量和方法，不管是静态的还是非静态的都可以。

在外部类里面创建成员内部类的实例：

this.new B()；

在外部类之外创建内部类的实例：

(new Test1()).new B().go();

在内部类里访问外部类的成员：

Test1.this.member

#### 实例代码



### 方法内部类（局部内部类）/在一个方法（外部类的方法）中定义的内部类

#### 定义

定义在方法中，比方法的范围还小。是内部类中最少用到的一种类型。

像局部变量一样，不能被public, protected, private和static修饰。

只能访问方法中定义的final类型的局部变量。

方法内部类在方法中定义，所以只能在方法中使用，即只能在方法当中生成方法内部类的实例并且调用其方法。

#### 实例代码



### 匿名内部类（Anonymous Inner Class）

#### 定义

没有名字的局部内部类，不使用关键字class, extends, implements, 没有构造方法。

#### 注意

匿名内部类不能有构造方法。

匿名内部类不能定义任何静态成员、方法和类。

匿名内部类不能是public,protected,private,static。

只能创建匿名内部类的一个实例。

一个匿名内部类一定是在new的后面，用其隐含实现一个接口或实现一个类。

因匿名内部类为局部内部类，所以局部内部类的所有限制都对其生效。

#### 是否可以extends其他类

不能。

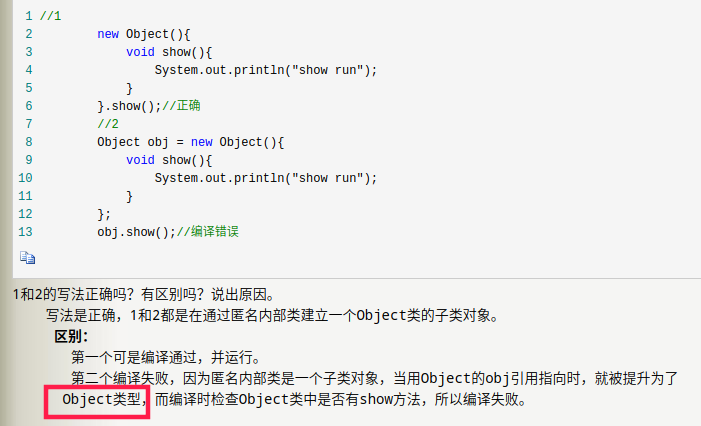
#### 是否可以implements接口

一个内部类可以作为一个接口，由另一个内部类实现。

#### 实例代码



#### 易错代码



## Static Nested Class和Inner Class

### 定义

Nested Class一般是C++的说法。Inner Class一边是Java的说法。

Java内部类和C++嵌套类最大不同就在于是否有指向外部的引用上。

静态嵌套类指的就是内部类添加了static修饰符，但是没有静态内部类这个说法，因为java语言规范定义：定义在类里面的称为嵌套类。

### 注意

创建静态嵌套类，不需要外部对象，而创建内部类则需要一个外部对象。

不能从静态嵌套类内部访问一个外部类对象。

## String类

### String str = new String(“123”)创建了几个对象

<https://blog.csdn.net/Sun1956/article/details/53161560>

如果常量池中没有，则创建了2个对象，一个在常量池中，一个在堆上。

如果常量池中已有，则创建了1个对象。

下图中的第一个字符串在常量池中不存在，所以intern()返回的就是常量池中的。



## Override和Overload

### Override（重写）

父类与子类之间多态性的一种表现。

### Overload（重载）

一个类中多态性的一种表现。

### 构造函数

不能重写，但可以被重载。

## Final类

### 特点

不可以继承。

## equals和hashCode

### 参考资料

<http://www.importnew.com/25783.html>

### equals

public boolean equals(Object obj) {

return (this == obj);

}

#### 自反性：x.equals(x)必须返回true。

#### 对称性：x.equals(y)与y.equals(x)的返回值必须相等。

#### 传递性：x.equals(y)为true，y.equals(z)也为true，那么x.equals(z)必须为true。

#### 一致性：如果对象x和y在equals()中使用的信息都没有改变，那么x.equals(y)值始终不变。

#### 非null：x不是null，y为null，则x.equals(y)必须为false。

### hashCode

public native int hashCode();

#### hashCode()在哈希表中起作用，如java.util.HashMap。

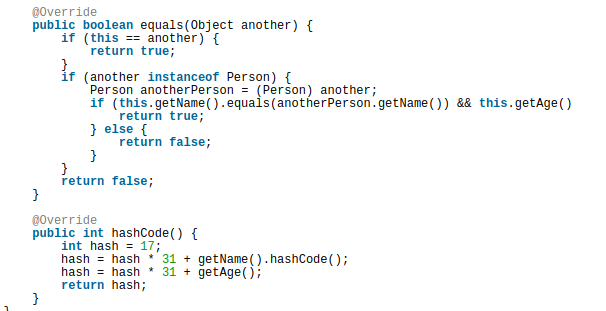
#### 如果对象在equals()中使用的信息都没有改变，那么hashCode()值始终不变。

#### 如果两个对象使用equals()方法判断为相等，则hashCode()方法也应该相等。

#### 如果两个对象使用equals()方法判断为不相等，则不要求hashCode()也必须不相等；但是开发人员应该认识到，不相等的对象产生不相同的hashCode可以提高哈希表的性能。

#### 如何重写hashCode

如果重写了equals()方法，检查条件“两个对象使用equals()方法判断为相等，则hashCode()方法也应该相等”是否成立，如果不成立，则重写hashCode ()方法。



## serialVersionUID

序列化对象时用到的版本号。

# 多线程

## 线程实现方式

### 继承Thread类

### 实现Runnable接口

## 同步实现方式

### 参考资料

<https://www.cnblogs.com/soundcode/p/6295910.html>

### Synchronized

#### 同步方法

#### 同步代码块

### Wait和nitify

一定要有synchronized代码。

### 使用特殊域变量(volatile)实现线程同步

### 使用重入锁实现线程同步

java.util.concurrent.ReentrantLock

### 使用局部变量实现线程同步

### 使用阻塞队列实现线程同步

### 使用原子变量实现线程同步

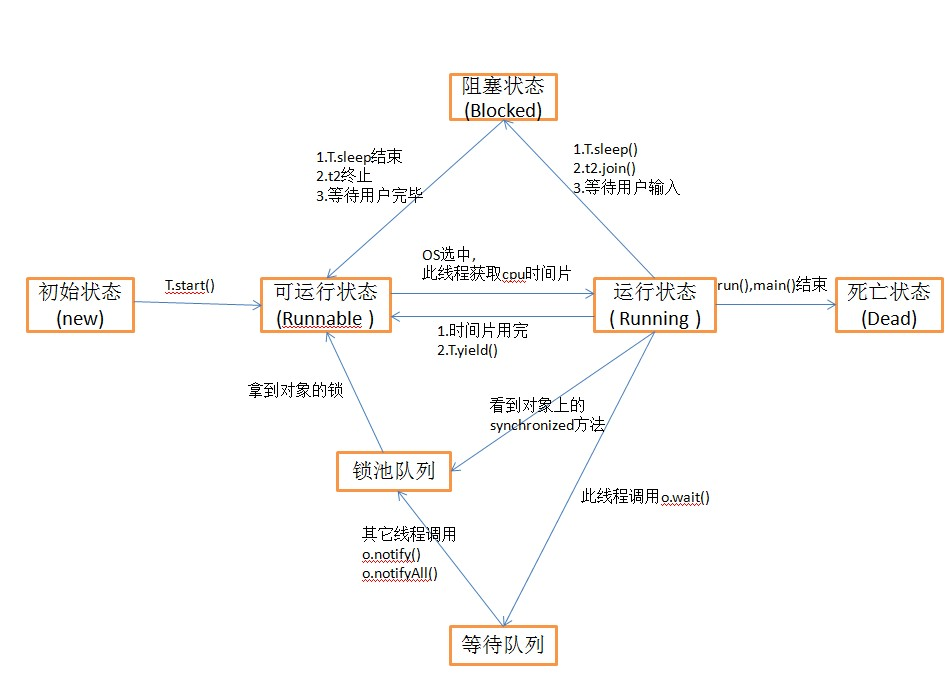
## 线程的状态

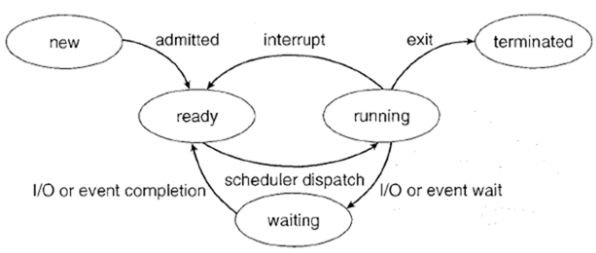
### 参考资料

<https://blog.csdn.net/pange1991/article/details/53860651>

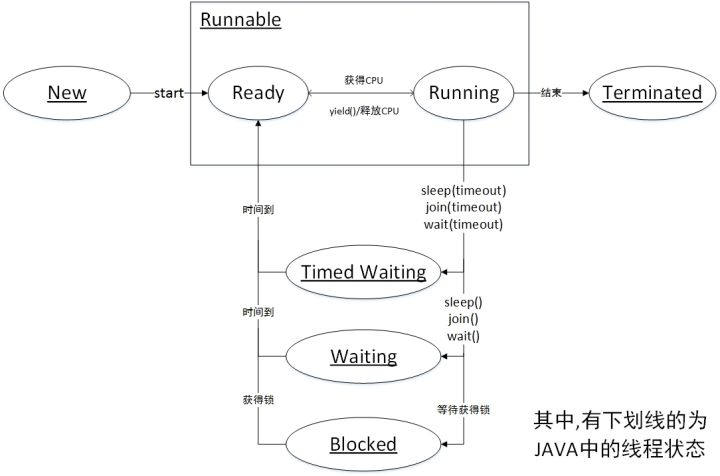
### 状态图

#### 早期5个状态图





#### 新的6个状态图



### 5个状态

#### 新建(NEW)

新创建了一个线程对象。

#### 可运行(RUNNABLE)

线程对象创建后，其他线程(比如main线程）调用了该对象的start()方法。该状态的线程位于可运行线程池中，等待被线程调度选中，获取cpu 的使用权 。

#### 运行(RUNNING)

可运行状态(runnable)的线程获得了cpu 时间片（timeslice） ，执行程序代码。

#### 阻塞(BLOCKED)

阻塞状态是指线程因为某种原因放弃了cpu 使用权，也即让出了cpu timeslice，暂时停止运行。直到线程进入可运行(runnable)状态，才有机会再次获得cpu timeslice 转到运行(running)状态。阻塞的情况分三种：

##### 等待阻塞

运行(running)的线程执行o.wait()方法，JVM会把该线程放入等待队列(waitting queue)中。

##### 同步阻塞

运行(running)的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池(lock pool)中。

##### 其他阻塞

运行(running)的线程执行Thread.sleep(long ms)或t.join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入可运行(runnable)状态。

#### 死亡(DEAD)

线程run()、main() 方法执行结束，或者因异常退出了run()方法，则该线程结束生命周期。死亡的线程不可再次复生。

# 垃圾收集（GC）

## 代码如何请求垃圾收集

### System.gc()

调用的就是下面的方法。

### Runtime.getRuntime().gc()

## 垃圾回收机制

# Java经典问题

## 双重检查加锁问题

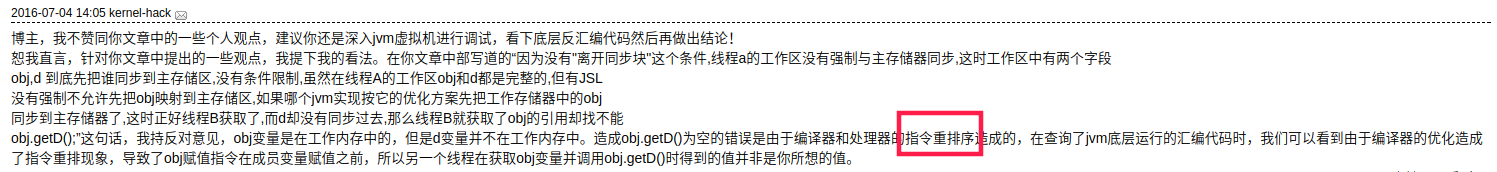
### 参考资料

<https://www.cnblogs.com/programerlrc/articles/5254675.html>

<https://blog.csdn.net/chenchaofuck1/article/details/51702129>

### 详细分析

#### 指令重排问题



### 解析方案

#### 内部私有静态类

public MyObject{

private static class instanceHoder{//内部私有的类，我特别用了小写开头。

static MyObject instance = new MyObject();

}

private static MyObect obj;

private Date d = new Data();

public Data getD(){return this.d;}

public static MyObect getInstance(){

return instanceHoder.instance;

}

}

private static class instanceHoder是类的定义并不会引起初始化。只有在首次调用getInstance时才会加载

instanceHoder类然后初始化instance实例。而静态初始化是由JVM来保证线程安全的，所以整个过程都不需要同

步参与，极大地提高了性能。

#### 线程安全的getInstance()方法

public static synchronized Singleton getInstance()

{

if (instance == null) //1

instance = new Singleton(); //2

return instance; //3

}

#### Volatile和synchronized组合方法

