# Object

## getClass

返回该对象运行时的 class 对象，返回的 Class 对象是由所表示的类的静态同步方法锁定的对象。

## hashCode

返回该对象的 hashcode，该方法对hash表提供支持，例如 HashMap。 对于该方法有几点需要注意： - 在运行中的Java应用，如果用在 equals 中进行比较的信息没有改变，那么不论何时调用都需要返回一致的int值。这个hash值在应用的两次执行中不需要保持一致。 - 如果两个对象根据 equals 方法认为是相等的，那么这两个对象也应该返回相等的 hashcode。 - 不要求两个不相等的对象，在调用 hashCode 方法返回的结果是必须是不同的。然而，程序员应该了解不同的对象产生不同的 hashcode 能够提升哈希表的效率。 Object的hashcode对不同的对象，尽可能返回不同的 hashcode 。这通常通过将对象的内部地址转换为整数来实现，但Java编程语言不需要此实现技术。

### Arrays.hashCode

Arrays.hashCode 是一个数组的浅哈希码实现，深哈希可以使用 deepHashCode。并且当数组长度为1时，Arrays.hashCode(object) = object.hashCode 不一定成立

### 31

不论是String、Arrays在计算多个元素的哈希值的时候，都会有31这个数字。主要有以下两个原因： - 31是一个不大不小的质数，是作为 hashCode 乘子的优选质数之一。 > 另外一些相近的质数，比如37、41、43等等，也都是不错的选择。那么为啥偏偏选中了31呢？请看第二个原因。

* 31可以被 JVM 优化，

$$31\*i=(i<<5)−i$$

* 。

上面两个原因中，第一个需要解释一下，第二个比较简单，就不说了。一般在设计哈希算法时，会选择一个特殊的质数。至于为啥选择质数，我想应该是可以降低哈希算法的冲突率。

在 Effective Java 中有一段相关的解释：

选择数字31是因为它是一个奇质数，如果选择一个偶数会在乘法运算中产生溢出，导致数值信息丢失，因为乘二相当于移位运算。选择质数的优势并不是特别的明显，但这是一个传统。同时，数字31有一个很好的特性，即乘法运算可以被移位和减法运算取代，来获取更好的性能：

$$31\*i==(i<<5)−i$$

，现代的 Java 虚拟机可以自动的完成这个优化。

## equals

判定两个对象是否相等。equals和hashCode需要同时被overwrite

## clone

创建一个该对象的副本，并且对于对象 x 应当满足以下表达式：

x.clone() != x
x.clone().getClass() == x.getClass()
x.clone().equals(x)

## toString

## wait

当前线程等待知道其他线程调用该对象的 notify 或者 notifyAll方法。当前线程必须拥有该对象的 monitor。线程释放该对象monitor的拥有权，并且等待到别的线程通知等待在该对象monitor上的线程苏醒。然后线程重新拥有monitor并继续执行。在某些jdk版本中，中断和虚假唤醒是存在的，所以wait方法需要放在循环中。

synchronized (obj) {
 while (<condition does not hold>)
 obj.wait();
 ... // Perform action appropriate to condition
}

该方法只能被拥有该对象monitor的线程调用。

### 虚假唤醒（spurious wakeup）

虚假唤醒就是一些obj.wait()会在除了obj.notify()和obj.notifyAll()的其他情况被唤醒，而此时是不应该唤醒的。

注意 Lock 的 Conditon.await 也有虚假唤醒的问题

解决的办法是基于while来反复判断进入正常操作的临界条件是否满足

同时也可以使用同步数据结构：BlokingQueue

#### 解释

虚假唤醒（spurious wakeup）是一个表象，即在多处理器的系统下发出 wait 的程序有可能在没有 notify 唤醒的情形下苏醒继续执行。

以运行在 Linux 的 hotspot 虚拟机上的 java 程序为例， wait 方法在 jvm 执行时实质是调用了底层 pthread\_cond\_wait/pthread\_cond\_timedwait 函数，挂起等待条件变量来达到线程间同步通信的效果，而底层 wait 函数在设计之初为了不减慢条件变量操作的效率并没有去保证每次唤醒都是由 notify 触发，而是把这个任务交由上层应用去实现，即使用者需要定义一个循环去判断是否条件真能满足程序继续运行的需求，当然这样的实现也可以避免因为设计缺陷导致程序异常唤醒的问题。

## notify

唤醒一个等待在该对象monitor上的线程。如果有多个线程等待，则会随机选择一个线程唤醒。线程等待是通过调用wait方法。

唤醒的线程不会立即执行，直到当前线程放弃对象上的锁。唤醒的线程也会以通常的方式和竞争该对象锁的线程进行竞争。也就是说，唤醒的线程在对该对象的加锁中没有任何优先级。

该方法只能被拥有该对象monitor的线程调用。线程拥有monitor有下面三种方式： - 执行该对象的 synchronized 方法 - 执行以该对象作为同步语句的synchronized方法体 - 对于class对象，可以执行该对象的static synchronized方法

在同一时间只能有一个线程能够拥有该对象monitor

## finalize

当 GC 认为该对象已经没有任何引用的时候，该方法被GC收集器调用。子类可以 overwrite 该方法来关闭系统资源或者其他清理任务。

finalize 的一般契约是，如果 Java 虚拟机确定不再有任何方法可以通过任何尚未死亡的线程访问此对象，除非由于某个操作，它将被调用通过最终确定准备完成的其他一些对象或类来完成。 finalize 方法可以采取任何操作，包括使该对象再次可用于其他线程；但是，finalize 的通常目的是在对象被不可撤销地丢弃之前执行清理操作。例如，表示输入/输出连接的对象的 finalize 方法可能会执行显式 I/O 事务，以在永久丢弃对象之前断开连接。

类 Object 的 finalize 方法不执行任何特殊操作;它只是正常返回。 Object 的子类可以覆盖此定义。

Java 编程语言不保证哪个线程将为任何给定对象调用 finalize 方法。但是，可以保证，调用 finalize 时，调用 finalize 的线程不会持有任何用户可见的同步锁。如果 finalize 方法抛出未捕获的异常，则忽略该异常并终止该对象的终止。在为对象调用 finalize 方法之后，在 Java 虚拟机再次确定不再有任何方法可以通过任何尚未死亡的线程访问此对象之前，不会采取进一步操作，包括可能的操作通过准备完成的其他对象或类，此时可以丢弃该对象。

对于任何给定对象，Java 虚拟机永远不会多次调用 finalize 方法。 finalize 方法抛出的任何异常都会导致暂停此对象的终结，但会被忽略。

### 缺陷

* 一些与 finalize 相关的方法，由于一些致命的缺陷，已经被废弃了，如 System.runFinalizersOnExit() 方法、Runtime.runFinalizersOnExit()方法。
* System.gc() 与 System.runFinalization() 方法增加了finalize方法执行的机会，但不可盲目依赖它们。
* Java 语言规范并不保证 finalize 方法会被及时地执行、而且根本不会保证它们会被执行。
* finalize 方法可能会带来性能问题。因为JVM通常在单独的低优先级线程中完成finalize的执行。
* 对象再生问题： finalize 方法中，可将待回收对象赋值给GC Roots可达的对象引用，从而达到对象再生的目的。
* finalize 方法至多由GC执行一次(用户当然可以手动调用对象的 finalize 方法，但并不影响GC对 finalize 的行为)。