Object 类起始于JDK1.0， 是类层次结构的根，Java中所有的类（包括标准容器类，比如数组）从根本上都继承自这个类，也就是直接或间接继承Object，所以Object类是Java中唯一没有父类的类。

方法摘要

| **返回类型** | **函数** | **解释说明** |
| --- | --- | --- |
| protected Object | clone() | 创建并返回此对象的一个副本。 |
| boolean | equals(Object obj) | 指示其他某个对象是否与此对象“相等”。 |
| protected void | finalize() | 当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。 |
| Class | getClass() | 返回此 Object 的运行时类。 |
| int | hashCode() | 返回该对象的哈希码值。 |
| void | notify() | 唤醒在此对象监视器上等待的单个线程。 |
| void | notifyAll() | 唤醒在此对象监视器上等待的所有线程。 |
| String | toString() | 返回该对象的字符串表示。 |
| void | wait() | 在其他线程调用此对象的 notify() 方法或 notifyAll() 方法前，导致当前线程等待。 |
| void | wait(long timeout) | 在其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者超过指定的时间量前，导致当前线程等待。 |
| void | wait(long timeout, int nanos) | 在其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量前，导致当前线程等待。 |

接下来按照顺序一个一个看。

package java.lang;

* 1

说明一点java.lang包在使用的时候无需显式导入，编译时由编译器自动导入。

源码中并没有Object的**构造方法**，但是，同样的，编译器在编译期间会给Object一个默认的空的构造方法（事实上，所有的Java类，只要类中没有构造方法，编译器都会默认的给一个空构造方法，若已有构造方法，则不会添加）：

public Object(){}

* 1

1、**getClass()**

private transient Class<?> shadow$\_klass\_;

public final Class<?> getClass() {

return shadow$\_klass\_;

}

返回这个Object的运行时Class对象。这个Class对象被所代表的类的static synchronized 方法锁定。   
实际结果类型是 Class < ? extends |X|>，其中 |X| 表示清除表达式中的静态类型，该表达式调用 getClass。 例如，以下代码片段中不需要强制转换：

Number n = 0;

Class<? extends Number> c = n.getClass();

2、**hashCode()**

public int hashCode() {

int lockWord = shadow$\_monitor\_;

final int lockWordStateMask = 0xC0000000; // Top 2 bits.

final int lockWordStateHash = 0x80000000; // Top 2 bits are value 2 (kStateHash).

final int lockWordHashMask = 0x0FFFFFFF; // Low 28 bits.

if ((lockWord & lockWordStateMask) == lockWordStateHash) {

return lockWord & lockWordHashMask;

}

return System.identityHashCode(this);

}

这个方法返回一个整型值（hash code value），如果两个对象被equals()方法判断为相等，那么它们就应该拥有同样的hash code。Object类的hashCode()方法为不同的对象返回不同的值，Object类的hashCode值表示的是对象的地址。

hashCode的一般性契约（需要满足的条件）如下：

1. 在Java应用的一次执行过程中，如果对象用于equals比较的信息没有被修改，那么同一个对象多次调用hashCode()方法应该返回同一个整型值。应用的多次执行中，这个值不需要保持一致，即每次执行都是保持着各自不同的值。
2. 如果equals()判断两个对象相等，那么它们的hashCode()方法应该返回同样的值。
3. 并没有强制要求如果equals()判断两个对象不相等，那么它们的hashCode()方法就应该返回不同的值。即，两个对象用equals()方法比较返回false，它们的hashCode可以相同也可以不同。但是，应该意识到，为两个不相等的对象产生两个不同的hashCode可以改善哈希表的性能。

简单理解：   
获得该对象的hash值，Java虚拟机规范并没有规定这个方法的具体实现，只是规定了同一个对象两次调用（任何条件情形下）这个方法返回的int值要想等（但并没有规定两个不同对象hash值一定不相同），具体实现由各个JVM厂商自己实现，所以返回的值意义并不一定（这里特指Object的hashCode方法），有可能返回的是对象的内存地址，也有可能是某个特定计算公式计算出来的值。

当你覆写（override）了equals()方法之后，必须也覆写hashCode()方法，反之亦然。

官网中说，重写equals方法，必重写hashCode，其实不然，若确定所有地方都没有用到类似Map的地方，就不必重写hashCode，因为Map的诸多方法是有用到hashCode方法判断两对象是否相等，而若你仅仅是自己用来判断两个对象是否equals，也就不必重写hashCode（当然，还要确定其他地方不会用到hashCode的地方，比如，以后用，别人用等，不过一般的，推荐重写hashCode方法，这样保证任何地方都不会因此出错）。   
　　若hash值不相等，则两个对象肯定不等（不equals）；   
　　若hash值相等，两个对象不一定相等（不一定equals）。   
　　equals相等，hash值肯定想等，也就是说，hash值相等时equals相等的必要条件。   
　　hashCode方法一般用来判断两个对象equals前置条件，用来排除，这样做的原因是，hashCode方法速度快，不相等的可快速否决掉，若hash相同，则再调用equals判断。

3、**equals()**

public boolean equals(Object obj) {

return (this == obj);

}

则上或则说语义上，设计目的上，equals的作用意义，是用来比较两个对象是否相等，这里是我们通常理解的相等：即两个对象其内容是否相等，而不是程序上来看，两个对象是否是同一个对象，即比较其内存地址；如果想比较两个对象是否是同一个对象（这里是说两个引用是否指向同一个对象），直接用==比较即可（==比较的就是对象的内存地址）。但这里重要的是，对于Object来说，它并不能知道子类是如何判断他们的两个实例是如何equals的，所以，默认的equals实现，比较的是两对象内存地址，即，若子类不重写equals方法，其作用等同于==。

“==”运算符判断两个引用是否指向同一个对象。对于Object类的equals()方法来说，它判断调用equals()方法的引用于传进来的引用是否一致，即这两个引用是否指向的是同一个对象。即Object类中的equals()方法等价于==。只有当继承Object的类覆写（override）了equals()方法之后，继承类实现了用equals()方法比较两个对象是否相等，才可以说equals()方法与==的不同。

equals()方法需要具有如下特点：

* 自反性（reflexive）：任何非空引用x，x.equals(x)返回为true。
* 对称性（symmetric）：任何非空引用x和y，x.equals(y)返回true当且仅当y.equals(x)返回true。
* 传递性（transitive）：任何非空引用x和y，如果x.equals(y)返回true，并且y.equals(z)返回true，那么x.equals(z)返回true。
* 一致性（consistent）：两个非空引用x和y，x.equals(y)的多次调用应该保持一致的结果，（前提条件是在多次比较之间没有修改x和y用于比较的相关信息）。

约定：对于任何非空引用x，x.equals(null)应该返回为false。

并且覆写equals()方法时，应该同时覆写hashCode()方法，反之亦然。

4、**clone()**

protected Object clone() throws CloneNotSupportedException {

if (!(this instanceof Cloneable)) {

throw new CloneNotSupportedException("Class " + getClass().getName() +

" doesn't implement Cloneable");

}

return internalClone();

}

本地Clone方法，用于对象的复制。   
克隆对象，克隆一个与原先对象所有字段值相等的对象，从而获得一个新的对象，需要注意的是：想要使用这个方法，对象类型必须实现Cloneable接口，否则会报错，原因是Object的clone方法有对对象类型验证，如没实现则报错抛异常； clone方法返回的是一个新的对象，这个对象的创建不是通过new（除非你像下面那样不通过Object的clone方法重写）指令，而是JVM通过其他指令创建的； clone有深度clone和浅clone，这主要是针对类中间具有引用类型而言划分的，详情可参看：Java clone深度解析。

5、internalClone()

private native Object internalClone();

6、toString()

public String toString() {

return getClass().getName() + "@" + Integer.toHexString(hashCode());

}

toString这个方法算是Object比较常用的方法了，它的意义是提供将类的字段以String形式格式化输出这一功能。当打印引用，如调用System.out.println()时，会自动调用对象的toString()方法，打印出引用所指的对象的toString()方法的返回值，因为每个类都直接或间接地继承自Object，因此每个类都有toString()方法。当然，同样的，Object不可能知道子类的字段信息，所以，默认toString输出的是：全路径类名+@+hash值。若你想要输出类的字段信息，需要重写toString方法，将该类字段信息以你自己的格式输出。

7、notify()

public final native void notify();

唤醒在此对象监视器上等待的单个线程。

8、notifyAll()

public final native void notifyAll();

唤醒在此对象监视器上等待的所有线程。

9、wait(long millis)

public final void wait(long millis) throws InterruptedException {

wait(millis, 0);

}

在其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者超过指定的时间量前，导致当前线程等待。

10、wait(long millis, int nanos)

public final native void wait(long millis, int nanos) throws InterruptedException;

在其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量前，导致当前线程等待。

11、wait()

public final native void wait() throws InterruptedException;

在其他线程调用此对象的 notify() 方法或 notifyAll() 方法前，导致当前线程等待。换句话说，此方法的行为就好像它仅执行 wait(0) 调用一样。当前线程必须拥有此对象监视器。该线程发布对此监视器的所有权并等待，直到其他线程通过调用 notify 方法，或 notifyAll 方法通知在此对象的监视器上等待的线程醒来。然后该线程将等到重新获得对监视器的所有权后才能继续执行。

12、finalize()

protected void finalize() throws Throwable { }

当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。在对象被GC（垃圾回收，详情可参考：Java GC概述）之前被调用（JVM主动调用），你可以重写这个方法，然后在这个对象回收之前做某些动作，这个方法对于这个对象来说只能调用一次，为什么会这么说呢？对象都回收了，没了，难道不是当然只能调用一次？不是这样的，若你理解了Java GC原理便知道，若当你在finalize方法中，将这个对象重新赋予了强引用，GC这个对象将失败，这个对象将继续存活，而下次这个对象又成为可回收对象了，GC回收这个对象的时候，这个对象的finalize方法将不会再执行。

另外，需要区分的是：finalize不是C/C++中的析构函数，更不是释放内存的方法，它只是提供了在回收一个对象之前做某些操作，如果你熟悉C ，那你知道C 允许你为一个类定义一个撤消函数（destructor ），它在对象正好出作用域之前被调用。Java不支持这个想法也不提供撤消函数。finalize() 方法只和撤消函数的功能接近。当你对Java 有丰富经验时，你将看到因为Java使用垃圾回收子系统，几乎没有必要使用撤消函数。而且，在设计之初，这个方法就是为了兼容C/C++程序员习惯（对的，貌似就是这样），后来设计者也说，这是个失败的设计，所以，可以的话，在实践中忘掉这个方法吧。