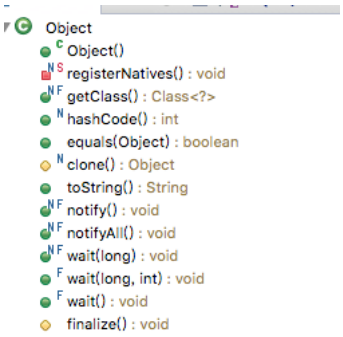
简介

Object 类属于 java.lang 包，此包下的所有类在使用时无需手动导入，系统会在程序编译期间自动导入。Object 类是所有类的基类，当承Object类，也就是说任何类都直接或间接继承此类，Object 类中能访问的方法在所有类中都可以调用

类的结构图



源码

```
package java.lang;

public class Object {

    private static native void registerNatives();
    static {
        registerNatives();
    }

     public final native Class<?> getClass();

     public native int hashCode();

     public boolean equals(Object obj) {
         return (this == obj);
     }

     protected native Object clone() throws CloneNotSupportedException;

     public String toString() {
         return getClass().getName() + "@" + Integer.toHexString(hashCode());
     }

     public final native void notify();

     public final native void notifyAll();

     public final native void wait(long timeout) throws InterruptedException;

     public final void wait(long timeout, int nanos) throws InterruptedException {
         if (timeout < 0) {
             throw new IllegalArgumentException("timeout value is negative");
         }

         if (nanos < 0 || nanos > 999999) {
             throw new IllegalArgumentException(
                                 "nanosecond timeout value out of range");
         }

         if (nanos > 0) {
```

```
            timeout++;
        }

        wait(timeout);
    }

    public final void wait() throws InterruptedException {
        wait(0);
    }

    protected void finalize() throws Throwable { }
}

```

方法说明

类构造器

类构造器是创建Java对象的途径之一，通过new 关键字调用构造器完成对象的实例化，还能通过构造器对对象进行相应的初始化。一个有显示声明，那么系统会默认创造一个无参构造器，在JDK的Object类源码中，是看不到构造器的，系统会自动添加一个无参构造器。我们

```
    Object obj = new Object(); 构造一个Object类的对象
```

registerNatives()

该方法是由native修饰的，native表示该方法的实现java本身并没有完成，而是有c/c++来完成，放在.dll动态库文件中

该方法源码中并没有任何注释说明，而且在静态块中调用了方法。首先明确在类初始化的时候，这个方法被调用执行了。这个调用的具

getClass()

返回Object的运行时class对象，返回的对象是被静态同步方法锁定的对象（这意味着，该类的所有对象中，同时只有一个对象可以获得多态的，可以是调用者的子类

hashCode()

hashCode()也是一个native方法，该方法返回调用对象的hash码。

equals(Object obj)

判断两个对象是不是相等

clone()

创建和返回一个对象的复制

一个对象可以被克隆的前提是该对象代表的类实现了Cloneable接口，否者会抛出一个CloneNotSupportedException异常

这里的复制是指的浅复制

toString()

返回一个表示该对象的字符串，默认实现是：类名@Integer.toHexString(hashCode())

notify()、notifyAll()、wait()、wait(long)、wait(long,int)

这几个方法是多线程编程里面常用的方法

finalize()

这是一个被垃圾收集器调用的方法，当一个对象没有被其他引用指向时，垃圾回收器会清理该对象，在回收该对象之前会调用finalize方
资源清理工作。一个对象只会被调用一次finalize方法。如果finalize方法抛出异常，这个对象的终结将会停止

hashcode与equal详解

equal详解

`equals()`方法是用来判断其他的对象是否和该对象相等的，在Object类中定义如下：

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

很明显是对两个对象的地址值进行的比较。但是我们知道，String 、Math、Integer、Double等这些封装类在使用equals()方法时，已经覆盖

以String类为例

```
public boolean equals(Object anObject) {
    if (this == anObject) {
        return true;
    }
    if (anObject instanceof String) {
        String anotherString = (String)anObject;
        int n = count;
        if (n == anotherString.count) {
            char v1[] = value;
            char v2[] = anotherString.value;
            int i = offset;
            int j = anotherString.offset;
            while (n-- != 0) {
                if (v1[i++] != v2[j++])
                    return false;
            }
            return true;
        }
    }
    return false;
}
```

很明显，这是进行的内容比较，而已经不再是地址的比较

需要注意的是当`equals()`方法被override时，`hashCode()`也要被override。按照一般`hashCode()`方法的实现来说，相等的对象，它们

hashCode详解

在Object中hashCode定义如下：

```
public native int hashCode();
```

说明是一个本地方法，它的实现是根据本地机器相关的。我们可以在自己写的类中覆盖hashcode()方法，比如String、Integer、Double等这例如在String类中定义的hashcode()方法如下：

```
/**
 * Returns a hash code for this string. The hash code for a
 * {@code String} object is computed as
 * <blockquote><pre>
 * s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
 * </pre></blockquote>
 * using {@code int} arithmetic, where {@code s[i]} is the
 * <i>i</i>th character of the string, {@code n} is the length of
 * the string, and {@code ^} indicates exponentiation.
 * (The hash value of the empty string is zero.)
 *
 * @return  a hash code value for this object.
 */
public int hashCode() {
    int h = hash;
    if (h == 0) {
        int off = offset;
        char val[] = value;
        int len = count;

        for (int i = 0; i < len; i++) {
            h = 31 * h + val[off++];
        }
        hash = h;
    }
    return h;
}
```

解释一下这个程序（String的API中写到）： s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]

应用说明

下面以HashSet为例进行分析，我们都知道：在hashset中不允许出现重复对象，元素的位置也是不确定的。在hashset中又是怎样判定判断两个对象是否相等的规则是：

1.判断两个对象的hashCode是否相等
如果不相等，认为两个对象也不相等，完毕
如果相等，转入判断依据 2
2.判断两个对象用equals运算是否相等
如果不相等，认为两个对象也不相等
如果相等，认为两个对象相等（equals()是判断两个对象是否相等的关键）

从结果上来说，只需要判断依据2即可，加入判断依据1 的原因是提高效率

具体示例

字符串的重复检测

```
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class HashSetTest {

    public static void main(String args[]) {
        String s1 = new String("aaa");
        String s2 = new String("aaa");
         System.out.println(s1 == s2);
         System.out.println(s1.equals(s2));
         System.out.println(s1.hashCode());
         System.out.println(s2.hashCode());
         Set hashset = new HashSet();
         hashset.add(s1);
         hashset.add(s2);
         Iterator it = hashset.iterator();
         while (it.hasNext()) {
             System.out.println(it.next());
         }
     }
}


/*输出

false
true
96321
96321
aaa

/*

// 说明：这是因为String类已经重写了equals()方法和hashcode()方法，所以hashset认为它们是相等的对象
```

自定义对象的重复检测

```
import java.util.HashSet;
import java.util.Iterator;

public class HashSetTest {

    public static void main(String[] args) {
        HashSet hs = new HashSet();
        hs.add(new Student(1, "zhangsan"));
        hs.add(new Student(2, "lisi"));
         hs.add(new Student(3, "wangwu"));
         hs.add(new Student(1, "zhangsan"));
```

```
12
13          Iterator it = hs.iterator();
14          while (it.hasNext()) {
15              System.out.println(it.next());
16          }
17      }
18  }
19
20  class Student {
21      int num;
22      String name;
23
24      Student(int num, String name) {
25          this.num = num;
26          this.name = name;
27      }
28
29      public String toString() {
30          return num + ":" + name;
31      }
32  }
33
34  /* 输出
35
36  1:zhangsan
37  3:wangwu
38  2:lisi
39  1:zhangsan
40
41  */
42
43
```

为什么hashset添加了相等的元素呢，这是不是和hashset的原则违背了呢？回答是：没有。因为在根据hashcode()对两次建立的new S时，生成的是不同的哈希码值，所以hashset把他当作不同的对象对待了，当然此时的equals()方法返回的值也不等。

为什么会生成不同的哈希码值呢？上面我们在比较s1和s2的时候不是生成了同样的哈希码吗？原因就在于我们自己写的Student类并没有法，所以在比较时，是继承的object类中的hashcode()方法，而object类中的hashcode()方法是一个本地方法，比较的是对象的地址（引用生成的当然是不同的对象了，造成的结果就是两个对象的hashcode()返回的值不一样，所以Hashset会把它们当作不同的对象对待

怎么解决这个问题呢？答案是：在Student类中重新hashcode()和equals()方法

```
1  class Student {
2      int num;
3      String name;
4
5      Student(int num, String name) {
6          this.num = num;
7          this.name = name;
8      }
9
10      public int hashCode() {
11          return num * name.hashCode();
12      }
```

```
13
14      public boolean equals(Object o) {
15          Student s = (Student) o;
16          return num == s.num && name.equals(s.name);
17      }
18
19      public String toString() {
20          return num + ":" + name;
21      }
22  }
23
24  /*  输出
25  1:zhangsan
26  3:wangwu
27  2:lisi
28  */
```