

## 目录

1. 谈谈对 kotlin 的理解.....	2
2. 讲一下常见编码方式? (源码) .....	2
3. Java 的异常体系 (源码) .....	5
4. 谈谈你对解析与分派的认识。(源码) .....	7
5. 修改对象 A 的 equals 方法的签名, 那么使用 HashMap 存放这个对象实例的时候, 会调用哪个 equals 方法? (源码) .....	7

# 1. 谈谈对 kotlin 的理解

<https://www.zhihu.com/question/25289041>

Kotlin 是 JetBrains 开发的针对 JVM、Android 和浏览器的静态编程语言。

## 为什么使用 Kotlin?

- 1、简洁——大大减少您需要编写的样板代码量
- 2、安全——避免整个类的错误，如空指针异常
- 3、多用途——支持多中类型的应用程序

多用途语言允许您构建任何类型的应用程序：

- (1) Android 开发：没有性能影响。运行时非常小。
- (2) 服务器应用：100%兼容所有 JVM 框架。
- (3) JavaScript: 在 Kotlin 中编写代码，并转换为 JavaScript 在 Node.js 或浏览器中运行。

4、可互操作——100%兼容 Java 从而可利用已有代码及 JVM 类库

- 5、多工具支持——自由选择命令行编译器或一级 IDE 支持

Kotlin 语言的特性：

- (1) **实用主义 ( Pragmatic )**：务实、注重工程实践性。Kotlin 是一门偏重工程实践与艺术上的极简风格的语言。

(2) **极简主义 (Minimalist)**: 语法简洁优雅不啰嗦, 类型系统中一切皆是引用 (reference)。

(3) **空安全 (Null Safety)**: 有一个简单完备的类型系统来支持空安全。

(4) **多范式 (multi-paradigm)**: 同时一等支持 OOP 与 FP 编程范式。各种编程风格的组合可以让我们更加直接地表达算法思想和解决问题的方案, 可以赋予我们思考上更大的自由度和灵活性。

(5) **可扩展**: 直接扩展类的函数与属性 (extension functions & properties)。

(6) **高阶函数与闭包 (higher-order functions & closures)**: Kotlin 的类型中, 函数类型 (function type) 也是一等类型 (first class type), 在 Kotlin 中我们可以把函数当成值进行传递。这直接赋予了 Kotlin 函数式编程的特性。

使用 Kotlin 可以写出一些非常优雅的代码。

(7) **支持快速实现 DSL**: 有了扩展函数、闭包等特性的支持, 使用 Kotlin 实现一个 DSL 将会相当简单方便。

## 2. 讲一下常见编码方式? (源码)

### 1. ASCII 码

上个世纪 60 年代, 美国制定了一套字符编码, 对英语字符与二

进制位之间的关系，做了统一规定。这被称为 ASCII 码，一直沿用至今。

注：ASCII 码一共规定了 128 个字符的编码，比如空格“SPACE”是 32（二进制 00100000），大写的字母 A 是 65（二进制 01000001）。这 128 个符号（包括 32 个不能打印出来的控制符号），只占用了一个字节的后面 7 位，最前面的 1 位统一规定为 0。0~31 是控制字符如换行回车删除等，32~126 是打印字符，可以通过键盘输入并且能够显示出来。

## 2、非 ASCII 编码

英语用 128 个符号编码就够了，但是用来表示其他语言，128 个符号是不够的。

注：亚洲国家的文字，使用的符号就更多了，汉字就多达 10 万左右。一个字节只能表示 256 种符号，肯定是不够的，就必须使用多个字节表达一个符号。比如，简体中文常见的编码方式是 GB2312，使用两个字节表示一个汉字，所以理论上最多可以表示  $256 \times 256 = 65536$  个符号。

这里只指出，虽然都是用多个字节表示一个符号，但是 GB 类的汉字编码与后文的 Unicode 和 UTF-8 是毫无关系的。

## 3. Unicode

正如上一节所说，世界上存在着多种编码方式，同一个二进制数字可以被解释成不同的符号。因此，要想打开一个文本文件，就必须

知道它的编码方式，否则用错误的编码方式解读，就会出现乱码。

可以想象，如果有一种编码，将世界上所有的符号都纳入其中。每一个符号都给予一个独一无二的编码，那么乱码问题就会消失。这就是 Unicode，就像它的名字都表示的，这是一种所有符号的编码。

注：

Unicode 当然是一个很大的集合，现在的规模可以容纳 100 多万个符号。每个符号的编码都不一样，比如，U+0639 表示阿拉伯字母 Ain，U+0041 表示英语的大写字母 A，U+4E25 表示汉字“严”。

Unicode 存在的问题：

需要注意的是，Unicode 只是一个符号集，它只规定了符号的二进制代码，却没有规定这个二进制代码应该如何存储。

比如，汉字“严”的 unicode 是十六进制数 4E25，转换成二进制数足足有 15 位（100111000100101），也就是说这个符号的表示至少需要 2 个字节。表示其他更大的符号，可能需要 3 个字节或者 4 个字节，甚至更多。

## 4. UTF-8

互联网的普及，强烈要求出现一种统一的编码方式。UTF-8 就是在互联网上使用最广的一种 unicode 的实现方式。其他实现方式还包括 UTF-16 和 UTF-32，不过在互联网上基本不用。重复一遍，这里的关系是，UTF-8 是 Unicode 的实现方式之一。

UTF-8（8-bit Unicode Transformation Format）是一种针对 Unicode 的可变长度字符编码，又称万国码。由 Ken Thompson 于 1992

年创建。现在已经标准化为 RFC 3629。UTF-8 用 1 到 4 个字节编码 Unicode 字符。用在网页上可以统一页面显示中文简体繁体及其它语言（如英文，日文，韩文）。

UTF-8 最大的一个特点，就是它是一种变长的编码方式。它可以使用 1~4 个字节表示一个符号，根据不同的符号而变化字节长度。

UTF-8 的编码规则很简单，只有二条：

1) 对于单字节的符号，字节的第一位设为 0，后面 7 位为这个符号的 unicode 码。因此对于英语字母，UTF-8 编码和 ASCII 码是相同的。

2) 对于 n 字节的符号 (n>1)，第一个字节的前 n 位都设为 1，第 n+1 位设为 0，后面字节的前两位一律设为 10。剩下的没有提及的二进制位，全部为这个符号的 unicode 码。

下表总结了编码规则，字母 x 表示可用编码的位。

Unicode 符号范围   UTF-8 编码方式	
UTF 字节数	(十六进制)   (二进制)
-----+	
一个字节	0000 0000-0000 007F   0xxxxxxx
两个字节	0000 0080-0000 07FF   110xxxxx 10xxxxxx
三个字节	0000 0800-0000 FFFF   1110xxxx 10xxxxxx 10xxxxxx
四个字节	0001 0000-0010 FFFF   11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

下面， 还是以汉字“严”为例，演示如何实现 UTF-8 编码。

已知“严”的 unicode 是 4E25 (100111000100101)，根据上表，可以发现 4E25 处在第三行的范围内 (0000 0800-0000 FFFF)，因此“严”的 UTF-8 编码需要三个字节，即格式是“1110xxxx 10xxxxxx 10xxxxxx”。然后，从“严”的最后一个二进制位开始，依次从后向

前填入格式中的 x，多出的位补 0。这样就得到了，“严”的 UTF-8 编码是“11100100 10111000 10100101”，转换成十六进制就是 E4B8A5。

注：

### Unicode 与 UTF-8 之间的转换：

通过上一节的例子，可以看到“严”的 Unicode 码是 4E25，UTF-8 编码是 E4B8A5，两者是不一样的。它们之间的转换可以通过程序实现。

在 Windows 平台下，有一个最简单的转化方法，就是使用内置的记事本小程序 Notepad.exe。打开文件后，点击“文件”菜单中的“另存为”命令，会跳出一个对话框，在最底部有一个“编码”的下拉条。

## 5. iso8859-1 编码

属于单字节编码，最多能表示的字符范围是 0-255，应用于英文系列。比如，字母 a 的编码为  $0 \times 61 = 97$ 。很明显，iso8859-1 编码表示的字符范围很窄，无法表示中文字符。

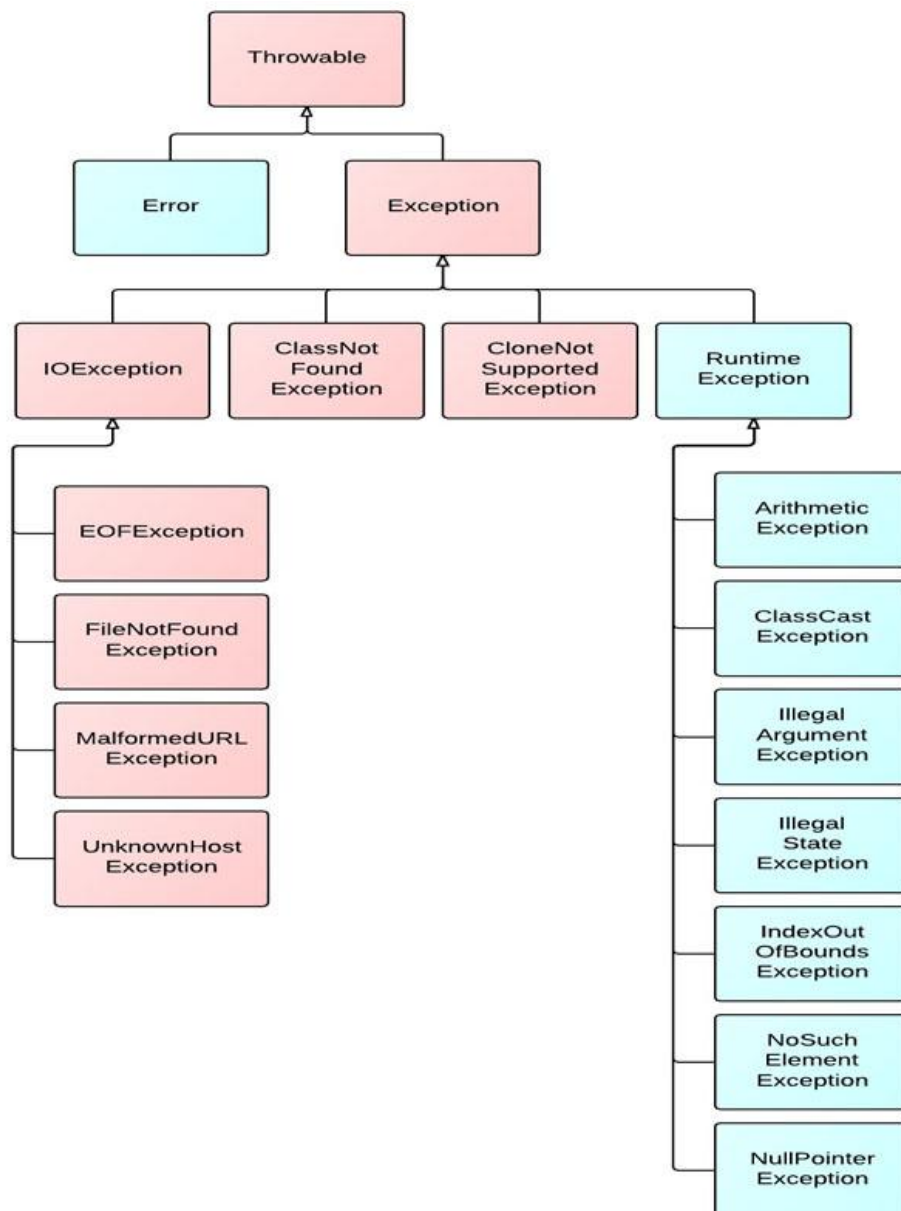
但是，由于是单字节编码，和计算机最基础的表示单位一致，所以很多时候，仍旧使用 iso8859-1 编码来表示。而且在很多协议上，默认使用该编码。（比如，虽然“中文”两个字不存在 iso8859-1 编码，以 gb2312 编码为例，应该是“d6d0 cec4”两个字符，使用 iso8859-1 编码的时候则将它拆开为 4 个字节来表示：“d6 d0 ce c4”（事实上，在进行存储的时候，也是以字节为单位处理的）。而如果是 UTF 编码，则是 6 个字节“e4 b8 ad e6 96 87”。很明显，这种表示方法还需要以另一种编码为基础。）

### 3. Java 的异常体系（源码）

<https://www.jianshu.com/p/93aa1cf26b97>

关于 java 异常的基础知识：

Java 异常以 Throwable 开始，扩展出 Error 和 Exception，而 Exception 又扩展出 RuntimeException 等异常，常见异常见下图：





## Throwable 扩展出两大类 Error 和 Exception

### (1) Error:

Error 是程序代码无法处理的错误，比如 OutOfMemoryError、ThreadDeath 等。这些异常发生时，Java 虚拟机（JVM）一般会选择线程终止退出，其表示程序在运行期间出现了十分严重、不可恢复的错误，应用程序只能中止运行。

### (2) Exception:

#### Exception 分运行时异常和非运行时异常

1) 运行时异常：运行时异常都是 RuntimeException 类及其子类异常，如 NullPointerException、IndexOutOfBoundsException 等，这些异常也是不检查异常，程序代码中自行选择捕获处理，也可以不处理。这些异常一般是由程序逻辑错误引起的，程序代码应该从逻辑角度尽可能避免这类异常的发生。

2) 非运行时异常：所有继承 Exception 且不是 RuntimeException 的异常都是检查异常，如上图中的 IOException 和 ClassNotFoundException，编译器会对其作检查，要么在方法体中声明抛出 checked Exception，要么使用 catch 语句捕获 checked Exception 进行处理，不然不能通过编译。

## 4. 谈谈你对解析与分派的认识。（源

码)

分派是双亲委派模型?

从虚拟机角度了解

<https://www.jianshu.com/p/355ae3bcec41>

解析:

虚拟机将常量池内的符号引用替换为直接引用的过程, 包括类或接口的解析、字段解析、类方法解析、接口方法解析。

5. 修改对象 A 的 equals 方法的签名, 那么使用 HashMap 存放这个对象实例的时候, 会调用哪个 equals 方法? (源码)

<https://www.jianshu.com/p/7d3feb156be4>

```
Set hashSet<C> = new java.util.HashSet<C>();  
hashSet.add(elem1);  
hashSet.contains(elem2);    // returns false!</pre>
```

当 equals 重载时, 这里有 4 个会引发 equals 行为不一致的常见问题:

1. 定义了错误的 equals 方法签名(signature)
2. 重载了 equals 的但没有同时重载 hashCode 的方法。
3. 建立在会变化字域上的 equals 定义。

#### 4. 不满足等价关系的 equals 错误定义

##### 问题 1: 定义错误 equals 方法签名(signature)

看上去非常明显，但是按照这种方式来定义equals就是错误的。

```
// An utterly wrong definition of equals
public boolean equals(Point other) {
    return (this.getX() == other.getX() && this.getY() == other.getY());
}
```

这个方法有什么问题呢？初看起来，它工作的非常完美：

```
Point p1 = new Point(1, 2);
Point p2 = new Point(1, 2);

Point q = new Point(2, 3);

System.out.println(p1.equals(p2)); // prints true

System.out.println(p1.equals(q)); // prints false
```

然而，当我们一旦把这个Point类的实例放入到一个容器中问题就出现了：

```
import java.util.HashSet;

HashSet<Point> coll = new HashSet<Point>();
coll.add(p1);

System.out.println(coll.contains(p2)); // prints false
```

```
Object p2a = p2;
```

现在我们重复第一个比较，但是不再使用p2而是p2a,我们将会得到如下的结果：

```
System.out.println(p1.equals(p2a)); // prints false
```

到底是那里出了了问题？事实上，之前所给出的equals版本并没有覆盖Object类的equals方法，因为他的类型不同。下面是Object的equals方法的定义

```
public boolean equals(Object other)
```

因为 Point 类中的 equals 方法使用的是以 Point 类而非 Object 类做为参数，因此它并没有覆盖 Object 中的 equals 方法。而是一种变化了的重载。在 Java 中重载被解析为静态的参数类型而非运行期的类型，因此当静态参数类型是 Point, Point 的 equals 方法就被调用。然而当静态参数类型是 Object 时，Object 类的 equals 就被调用。

因为这个方法并没有被覆盖，因此它仍然是实现成比较对象标示。这就是为什么虽然 p1 和 p2a 具有同样的 x,y 值，”p1.equals(p2a)” 仍然返回了 false。这也是会什么 HashSet 的 contains 方法返回 false 的原因，因为这个方法操作的是泛型，他调用的是一般化的 Object 上 equals 方法而非 Point 类上变化了的重载方法 equals

**发生原因：没有重载 equals () 方法**

一个更好但不完美的equals方法定义如下：

```
// A better definition, but still not perfect
@Override public boolean equals(Object other) {
    boolean result = false;
    if (other instanceof Point) {
        Point that = (Point) other;
        result = (this.getX() == that.getX() && this.getY() == that.getY());
    }
    return result;
}
```

现在equals有了正确的类型，它使用了一个Object类型的参数和一个返回布尔型的结果。这个方法的实现使用instanceof操作和做了一个造型。它首先检查这个对象是否是一个Point类，如果是，他就比较两个点的坐标并返回结果，否则返回false。

## 问题 2：重载了 equals 的但没有同时重载 hashCode 的方法

如果你使用上一个定义的Point类进行p1和p2a的反复比较，你都会得到你预期的true的结果。但是如果你将这个类对象放入到HashSet.contains()方法中测试，你就有可能仍然得到false的结果：

```
Point p1 = new Point(1, 2);
Point p2 = new Point(1, 2);

HashSet<Point> coll = new HashSet<Point>();
coll.add(p1);

System.out.println(coll.contains(p2)); // 打印 false (有可能)
```

事实上，这个个结果不是100%的false，你也可能有返回ture的经历。如果你得到的结果是true的话，那么你试试其他的坐标值，最终你一定会得到一个在集合中不包含的结果。导致这个结果的原因是Point重载了equals却没有重载hashCode。

注意上面例子的容器是一个 HashSet，这就意味着容器中的元素根据他们的哈希码被放入到”哈希桶 hash buckets”中。

contains 方法首先根据哈希码在哈希桶中查找，然后让桶中的所有

元素和所给的参数进行比较。现在，虽然最后一个 Point 类的版本重定义了 equals 方法，但是它并没有同时重定义 hashCode。因此，hashCode 仍然是 Object 类的那个版本，即：**所分配对象的一个地址的变换**。

所以 p1 和 p2 的哈希码理所当然的不同了，甚至是即使这两个点的坐标完全相同。不同的哈希码导致他们具有极高的可能性被放入到集合中不同的哈希桶中。contains 方法将会去找 p2 的哈希码对应哈希桶中的匹配元素。但是大多数情况下，p1 一定是在另外一个桶中，因此，p2 永远找不到 p1 进行匹配。当然 p2 和 p2 也可能偶尔会被放入到一个桶中，在这种情况下，contains 的结果就为 true 了。

**总结原因：**hashCode 获取的数值还是 Object 对象中的 hashCode 数值，因此，即使内容一样，还是会因为 hashCode 不同而不能匹配为相同（在有包含 Hash 算法的容器中）

事实上，在 Java 中，hashCode 和 equals 需要一起被重定义是众所周知的。此外，hashCode 只可以依赖于 equals 依赖的域来产生值。对于 Point 这个类来说，下面的的 hashCode 定义是一个非常合适的定义。

```
@Override public boolean equals(Object other) {  
    boolean result = false;  
    if (other instanceof Point) {  
        Point that = (Point) other;  
        result = (this.getX() == that.getX() && this.getY() == that.getY());  
    }  
    return result;  
}  
  
@Override public int hashCode() {  
    return (41 * (41 + getX()) + getY());  
}
```

这只是 hashCode 一个可能的实现。x 域加上常量 41 后的结果再乘与 41 并将结果在加上 y 域的值。这样做就可以以低成本的运行时间和低成本代码大小得到一个哈希码的合理的分布(译者注：性价比相对较高的做法)。

增加 hashCode 方法重载修正了定义类似 Point 类等价性的问题。然而，关于类的等价性仍然有其他的问题点待发现。

### 问题 3：建立在会变化字段上的 equals 定义

让我们在 Point 类做一个非常微小的变化



```

public class Point {

    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public void setX(int x) { // Problematic
        this.x = x;
    }

    public void setY(int y) {
        this.y = y;
    }

    @Override public boolean equals(Object other) {
        boolean result = false;
        if (other instanceof Point) {
            Point that = (Point) other;
            result = (this.getX() == that.getX() && this.getY() == that.getY());
        }
        return result;
    }

    @Override public int hashCode() {
        return (41 * (41 + getX()) + getY());
    }
}

```

唯一的区别是x和y域不再是final，并且两个set方法被增加到类中来，并允许客户改变x和y的值。equals和hashCode这个方法的定义现在是基于在这两个会发生变化的域上，因此当他们的域的值改变时，结果也就跟着改变。因此一旦你将这个point对象放入到集合中你将会看到非常神奇的效果。

结果是，集合中不包含 p，但是 p 在集合的元素中！到底发生了什么！当然，所有的这一切都是在 x 域的修改后才发生的，p 最终的



的 hashCode 是在集合 coll 错误的哈希桶中。即，原始哈希桶不再有其新值对应的哈希码。换句话说，p 已经在集合 coll 的是视野范围之外，虽然他仍然属于 coll 的元素。

**问题原因：**hashCode 值可变导致对应的哈希码也可变，故有可能边之前存进去了，但修改后，在原表找不到对应的值从而导致错误

如果你需要根据对象当前的状态进行比较的话，你应该不要再重定义 equals，应该起其他的方法名字而不是 equals。对于我们的 Point 类的最后的定义，我们最好省略掉 hashCode 的重载，并将比较的方法名命名为 equalsContents，或其他不同于 equals 的名字。

（那么 Point 将会继承原来默认的 equals 和 hashCode 的实现，因此当我们修改了 x 域后 p 依然会呆在其原来在容器中应该在位置。）

**问题 4：不满足等价关系的 equals 错误定义**

```
Point p = new Point(1, 2);

ColoredPoint cp = new ColoredPoint(1, 2, Color.RED);

System.out.println(p.equals(cp)); // 打印真 true

System.out.println(cp.equals(p)); // 打印假 false
```

“p等价于cp”的比较这个调用的是定义在Point类上的equals方法。这个方法只考虑两个点的坐标。因此比较返回真。在另外一方面，“cp等价于p”的比较这个调用的是定义在ColoredPoint类上的equals方法，返回的结果却是false，这是因为p不是ColoredPoint，所以equals这个定义违背了对称性。

违背对称性对于集合来说将导致不可以预期的后果，例如：

```
Set<Point> hashSet1 = new java.util.HashSet<Point>();
hashSet1.add(p);
System.out.println(hashSet1.contains(cp)); // 打印 false

Set<Point> hashSet2 = new java.util.HashSet<Point>();
hashSet2.add(cp);
System.out.println(hashSet2.contains(p)); // 打印 true
```

```

public class ColoredPoint extends Point { // Problem: equals not transitive

    private final Color color;

    public ColoredPoint(int x, int y, Color color) {
        super(x, y);
        this.color = color;
    }

    @Override public boolean equals(Object other) {
        boolean result = false;
        if (other instanceof ColoredPoint) {
            ColoredPoint that = (ColoredPoint) other;
            result = (this.color.equals(that.color) && super.equals(that));
        }
        else if (other instanceof Point) {
            Point that = (Point) other;
            result = that.equals(this);
        }
        return result;
    }
}

```

**问题原因：**因两个类的 equals 的对比方法不同，导致 A.equals (B) 和 B.equals (A) 的结果一个真一个为假，不符合对称性原则

```

public class Point {

    private final int x;
    private final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    @Override public boolean equals(Object other) {
        boolean result = false;
        if (other instanceof Point) {
            Point that = (Point) other;
            result =(that.canEqual(this) && this.getX() == that.getX() && this.getY()
        }
        return result;
    }

    @Override public int hashCode() {
        return (41 * (41 + getX()) + getY());
    }

    public boolean canEqual(Object other) {
        return (other instanceof Point);
    }

}

```

这个版本的Point类的equals方法中包含了一个额外的需求，通过canEquals方法来决定另外一个对象是否是满足可以比较的对象。在Point中的canEqual宣称了所有的Point类实例都能被比较。

下面是ColoredPoint相应的实现

```
public class ColoredPoint extends Point { // 不再违背对称性


    private final Color color;

    public ColoredPoint(int x, int y, Color color) {
        super(x, y);
        this.color = color;
    }

    @Override public boolean equals(Object other) {
        boolean result = false;
        if (other instanceof ColoredPoint) {
            ColoredPoint that = (ColoredPoint) other;
            result = (that.canEqual(this) && this.color.equals(that.color) && super.equals(that));
        }
        return result;
    }

    @Override public int hashCode() {
        return (41 * super.hashCode() + color.hashCode());
    }

    @Override public boolean canEqual(Object other) {
        return (other instanceof ColoredPoint);
    }
}
```



在上显示的新版本的Point类和ColoredPoint类定义保证了等价的规范。等价是对称和可传递的。比较一个Point和ColoredPoint类总是返回false。因为点p和着色点cp,“p.equals(cp)”返回的是假。并且,因为cp.canEqual(p)总返回false。相反的比较, cp.equals(p)同样也返回false,由于p不是一个ColoredPoint,所以在ColoredPoint的equals方法体内的第一个instanceof检查就失败了。