对于所有对象都通用的方法

8、覆盖 equals 时请遵守通用约定

覆盖 equals 方法看起来似乎很简单，但是有许多覆盖方式会导致错误，并且后果非常严重，最容易避免这类问题的办法就是不覆盖 equals 方法，在这种情况下，类的每个实例都只与它自身相等。如果满足了以下任何一个条件，这就正是所期望的结果。

(1)类的每个实例本质上都是唯一的。对于代表活动实体而不是值（value）的类来说确实如此，例如 Thread。Object 提供的 equals 实现对于这些类来说是正确的行为。

(2)不关心类是否提供了“逻辑相等(logical equality)”的测试功能。如 Java.util.Random 覆盖了 equals，以检查两个 Random 实例是否产生相同的随机序列，但是设计者并不认为客户需要或者期望这样的功能。在这样的情况下，从 Object 继承得到的 equals 实现已经足够了。

(3)超类已经覆盖了 equals，从超类继承过来的行为对于子类来说也是合适的。例如大多数 Set 实现都从 AbstractSet 继承 equals 实现，类似的有 List 和 Map 等。

(4)类是私有的或是包级私有的，可以确定它的 equals 方法永远不会被调用。在这种条件下，无疑应该覆盖 equals 方法，以防止它被意外调用。

在覆盖 equals 方法时，必要要遵守 Object 中关于 equals 的通用约定：

(1)**自反性(reflexive)。**对于任何非 null 的引用值 x，x.equals(x)必须返回 true。

(2)对称性(symmetric)。对于任何非 null 引用值 x 和 y，当且仅当 y.equals(x)返回 true 时，x.equals(y)必须返回 true。

(3)传递性(transitive)。对于任何非 null 引用值 x、y、z，如果 x.equals(y)返回 true，并且 y.equals(z)也返回 true，那么 x.equals(z)也必须返回 true。

(4)一致性(consistent)。对于任何非 null 的引用值 x 和 y，只要 equals 的比较操作在对象中所用的信息没有被修改，多次调用 x.equals(y)就会一致的返回 true 或 false。

(5)非空性(not null)。对于任何非 null 的引用值 x，x.equals(null)必须返回 false。

实现高质量 equals 方法的诀窍：

(1)使用==操作符检查“参数是否为这个对象的引用”。如果是，就返回 true,这只不过是一种性能优化，如果比较操作很昂贵时就值得这么做。

(2)使用 instanceof 操作符检查“参数是否为正确的类型”。一般来说，所谓正确的类型指的是 equals 方法所在的那个类。

(3)把参数转换成正确的类型。

(4)对于该类中的每个“关键”域，检查参数中的域是否与该对象中对应域相匹配。

(5)当编写完成 equals 方法之后，应该考虑它是否是对称的、传递的、一致的。

针对写 equals 的告诫：

(1)覆盖 equals 时总是要覆盖 hashCode。

(2)不要企图让 equals 方法过于智能。过度去寻找各种等价关系很容易陷入麻烦之中。

(3)不要将 equals 声明中的 Object 对象替换成其他的类型。如果替换以后就是重载了 Object 类的 equals 类，而不是重写了。@Override 可以防止范这种错误。

9、覆盖 equals 时总要覆盖 hashCode

Demo

```
public final class PhoneNumber {
  private final short areaCode;
  private final short prefix;
  private final short lineNumber;

  public PhoneNumber(int areaCode, int prefix, int lineNumber) {
    rangeCheck(areaCode, 999, "area code");
    rangeCheck(prefix, 999, "prefix");
    rangeCheck(lineNumber, 9999, "line number");
    this.areaCode = (short) areaCode;
    this.prefix = (short) prefix;
    this.lineNumber = (short) lineNumber;
  }

  private static void rangeCheck(int arg, int max, String name) {
    if (arg < 0 || arg > max)
      throw new IllegalArgumentException(name + ": " + arg);
  }

  @Override
  public boolean equals(Object o) {
    if (o == this)
      return true;
    if (!(o instanceof PhoneNumber))
      return false;
    PhoneNumber pn = (PhoneNumber) o;
    return pn.lineNumber == lineNumber && pn.prefix == prefix
        && pn.areaCode == areaCode;
  }
  public static void main(String[] args) {
    Map<PhoneNumber, String> m = new HashMap<PhoneNumber, String>();
    m.put(new PhoneNumber(707, 867, 5309), "Jenny");
    System.out.println(m.get(new PhoneNumber(707, 867, 5309)));
  }
```

以上代码错误答案是输出"Jenny"，但它实际却返回 null。由于没有覆盖 hashCode 方法，从而导致两个相等的实例具有不相等的散列码，违反了 hashCode 的约定。

10、始终要覆盖 toString

java.lang.Object 的 toString 方法的实现：

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

可以看出，它返回的字符串只类类名加上一个“@符号”，后面是十六进制形式的 hashCode，这些信息对我们来说用处不大，所以为了提供更好的关于类和对象的说明，我们应该总是覆盖 toString()方法来提供更加清晰的说明。

(1)虽然遵守 toString 的约定并不像遵守 equals 和 hashCode 的约定那么重要，但是，提供好的 toString 实现可以使类用起来更舒适。当对象被传递给 println、print、字符串联操作符(+)以及 assert 或者被调试器打印出来时，toString 方法都会被自动调用。

(2)在实际应用中，toString 方法返回对象中包含的所有值得关注的信息。如果对象太大，或者对象中包含的状态信息难以用字符串表达，这样做就有点不切合实际。在这种情况下，toString 应该返回一个摘要信息难以用字符串来表达，这样做就有点不切合实际。在这种情况下，toString 应该返回一个摘要信息。

(3)在实现 toString 的时候，必须要做出一个很重要的决定：是否在文档中制定返回值的格式。对于值类(value class)，比电话号码、矩阵类，也建议这么做。指定格式的好处是，它可以被用作一种标准的、明确的、适合人阅读的对象表示法。

(4)指定 toString 返回值的格式也有不足之处：如果这个类已经被广泛使用，一旦指定格式，就必须始终如一的坚持这种格式。

11、谨慎地覆盖 clone

Clone 提供一种语言之外的机制：无需调用构造器就可以创建对象。

(1)Cloneable 接口与 Object.clone 方法

Cloneable 接口的目的是作为对象的一个 mixin 接口(混合型接口)，表明这样的对象允许克隆(clone)。遗憾的是 Cloneable 接口里并没有 clone 方法，其实它什么方法都没有，跟 Serializable 接口一样，都是占着茅坑不拉屎。它只是声明该对象可以被克隆，具体行为由类设计者决定。如果类设计者忘记提供一个良好的 clone 方法或根本不提供 clone 方法，那么类客户使用时必定会出错，这样 Cloneable 接口并没达到它的目的。

(2)Cloneable 接口的作用

它决定了 Object 中受保护的 clone 方法实现的行为：如果一个类实现了 Cloneable，Object 的 clone 方法就返回该对象的逐域拷贝，否则就返回 CloneNotSupportException 异常。这是接口的一种极端非典型的用法，不值得效仿。通常情况下，实现接口是为了表明类可以为它的客户做些什么。然而，对于 Cloneable 接口，它改变了超类中受保护的方法的行为。

(3)Object 中 Clone 方法的通用约定

(a) x.clone() != x;

(b) x.clone().getClass() = x.getClass();

(c) x.clone().equals(x);

(4)如果类的每个域包含一个基本类型的值，或者包含一个指向不可变对象的引用，那么被返回的对象则正是所需要的对象，如 PhoneNumber 类：

```
public class PhoneNumber implements Cloneable{
    private final int areaCode;
    private final int prefix;
    private final int lineNumber;
    public PhoneNumber(int areaCode, int prefix, int lineNumber) {
        rangeCheck(areaCode, 999, "area code");
        rangeCheck(prefix, 999, "prefix");
        rangeCheck(lineNumber, 9999, "line number");
        this.areaCode = areaCode;
        this.prefix = prefix;
        this.lineNumber = lineNumber;
    }
```

```
    private static void rangeCheck(int arg, int max, String name) {
        if(arg < 0 || arg > max) {
            throw new IllegalArgumentException(name + ": " + arg);

        }
    }

    @Override
    public boolean equals(Object o) {
        if(o == this)
            return true;
        if(!(o instanceof PhoneNumber))
            return false;
        PhoneNumber pn = (PhoneNumber)o;
        return pn.lineNumber == lineNumber
                && pn.prefix == prefix
                && pn.areaCode == areaCode;
    }

    @Override
    public PhoneNumber clone() {     // 只需要简单地调用 super.clone() 而不用做进一步的处理。
        try {
            return (PhoneNumber) super.clone();
        } catch(CloneNotSupportedException e) {
            throw new AssertionError();
        }
    }
}
```

(5)如果对象中包含的域引用了可变的对象：

```
public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITAL_CAPACITY = 16;

    public Stack() {
        elements = new Object[DEFAULT_INITAL_CAPACITY];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }
```

```
public Object pop() {
    if(size == 0) {
        throw new EmptyStackException();
    }
    Object result = elements[--size];
        elements[size] = null;
        return result;
    }
    private void ensureCapacity() {
        if(elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

如果把这个类做成是可克隆的。如果它的 clone 方法仅仅返回 super.clone()，这样得到的 Stack 实例，在 size 域有正确的值，但它的 elements 域将引用与原始 Stack 实例相同的数组。

为了使 Stack 类中的 clone 方法正常地工作，必须拷贝栈的内部信息：

```
@Override
public Stack clone() {
    try {
        Stack result = (Stack) super.clone();
        result.elements = elements.clone();
        return result;
    } catch (CloneNotSupportedException e) {
        throw new AssertionError();
    }
}
```

12、考虑实现 Comparable 接口

(1) Comparable 接口是用来实现对象排序的

```
public class CompObj implements Comparable<CompObj> {
   private int age;
   private String name;
   public CompObj(int age, String name) {
      this.age = age;
      this.name = name;
   }
   public int getAge() {
      return age;
   }
   public String getName() {
      return name;
   }

   @Override
   public int compareTo(@NonNull CompObj another) {
      return this.age - another.age;
   }
}
```

然后就可以通过调用排序的方法进行排序，如下：

```
List<CompObj> list = ...;//初始化一个 list
Collections.sort(list); //集合排序，就是这么简单
CompObj array[] = ...;//初始化一个数组
Arrays.sort(list);  //数组排序，就是这么简单
```

(2)实现 Comparable 接口的约定

a. 满足对称性。

即对象 A.comparaTo(B) 大于 0 的话，则 B.comparaTo(A)必须小于 0;

b. 满足传递性。

即对象 A.comparaTo(B) 大于 0，对象 B.comparaTo(Z)大于 0，则对象 A.comparaTo(Z)一定要大于 0；

c. 建议 comparaTo 方法和 equals()方法保持一致。

即对象 A.comparaTo(B)等于 0，则建议 A.equals(B)等于 true。

d. 对于实现了 Comparable 接口的类，尽量不要继承它，而是采取复合的方式。