Equality of Immutable Types

1. AF映射到同样的结果，则等价
2. 站在外部观察者角度：对两个对象调用任何相同的操作，都会得到相同的结果，则认为这两个对象是等价的。 反之亦然！ （行为等价性）

Equality of Mutable Types

```
  1. 观察等价性：在不改变状态的情况下，两个mutable对象是否看起来一致（When they cannot be distinguished by observation that doesn’t change the state of the objects, i.e., by calling only observer, producer, and creator methods. This is often strictly called observational equality）
  2. 行为等价性：调用对象的任何方法都展示出一致的结果（When they cannot be distinguished by any observation, even state changes.This interpretation allows calling any methods on the two objects, including mutators. This is called behavioral equality）
```

对可变类型来说，往往倾向于实现严格的观察等价性

***but*这样可能会有bug**

- **Suppose we make a `List` , and then drop it into a `Set` :**

```
List<String> list = new ArrayList<>();
list.add("a");

Set<List<String>> set = new HashSet<List<String>>();
set.add(list);
```

- **We can check that the set contains the list we put in it, and it does:**

```
set.contains(list) → true
```

- **But now we mutate the list:** `list.add("goodbye");`
- **And it no longer appears in the set!** `set.contains(list) → false!`
- **It’s worse than that, in fact: when we iterate over the members of the set, we still find the list in there, but `contains()` says it’s not there.**

```
for (List<String> l : set) {
    set.contains(l) → false!
}
```

- `List<String>` **is a mutable object. In the standard Java implementation of collection classes like `List`, mutations affect the result of `equals()` and `hashCode()` .**
- **When the list is first put into the `HashSet`, it is stored in the hash bucket 哈希桶/散列桶 corresponding to its `hashCode()` result at that time.**
- **When the list is subsequently mutated, its `hashCode()` changes, but `HashSet` doesn't realize it should be moved to a different bucket. So it can never be found again.**
- **When `equals()` and `hashCode()` can be affected by mutation, we can break the rep invariant of a hash table that uses that object as a key.**

你在派出所申领了身份证，留了当时的照片；
几个月以后，你“整容”了(mutated)，你坐飞机案件的时候就无法匹配到你的身份证照片了…

简单的来说就是HashSet用的是hash值来找元素，list变了之后它的hash值也变了，但是list已经在HashSet中了，它的位置不会再变了，当set再调用contains时，set还是按原来的HashCode来找，所以找不到

- `equals()` **should implement behavioral equality. 对可变类型，实现行为等价性即可**
- **In general, that means that two references should be `equals()` if and only if they are aliases for the same object. 也就是说，只有指向同样内存空间的objects，才是相等的。**
- **So mutable objects should just inherit `equals()` and `hashCode()` from `Object`. 所以对可变类型来说，无需重写这两个函数，直接继承`Object`的两个方法即可。**
- **For clients that need a notion of observational equality (whether two mutable objects "look" the same in the current state), it's better to define a new method, e.g., `similar()`. 如果一定要判断两个可变对象看起来是否一致，最好定义一个新的方法。**

== vs equals

== 引用等价性

equals 对象等价性

在自定义ADT时，需要重写Object的equals()

override equals方法

```java
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof 原类)) return false;
    将o强转为原类;
    自己的判断;
}
```

```java
 1  public class Name {
 2  private final String first, last;
 3  public Name(String first, String last) {
 4      if (first == null || last == null)
 5      throw new NullPointerException();
 6      this.first = first;
 7      this.last = last;
 8  }
 9  public boolean equals(Name o) {
10      return first.equals(o.first) && last.equals(o.last);
11  }
12  public int hashCode() {
13      return 31 * first.hashCode() + last.hashCode();
14  }
15  public static void main(String[] args) {
16      Set<Name> s = new HashSet<>();
17      s.add(new Name("Mickey", "Mouse"));
18      System.out.println(s.contains(new Name("Mickey", "Mouse")));
19      }
20  }
```

结果：false

原因：contains方法底层，用的是Object来接收s中的变量，所以第18行中的contains中在比较的时候想调用equals方法，但是由于变量的编译类型为Object，所以，由动态绑定原理，jvm会先在Name类中找equals方法(是Object中的equals方法的重写)，没找到，因为Name中的equals方法是overload，所以jvm会调用Name的父类equals方法，所以是false

Using instanceof is dynamic type checking，It should be disallowed anywhere except for implementing equals .

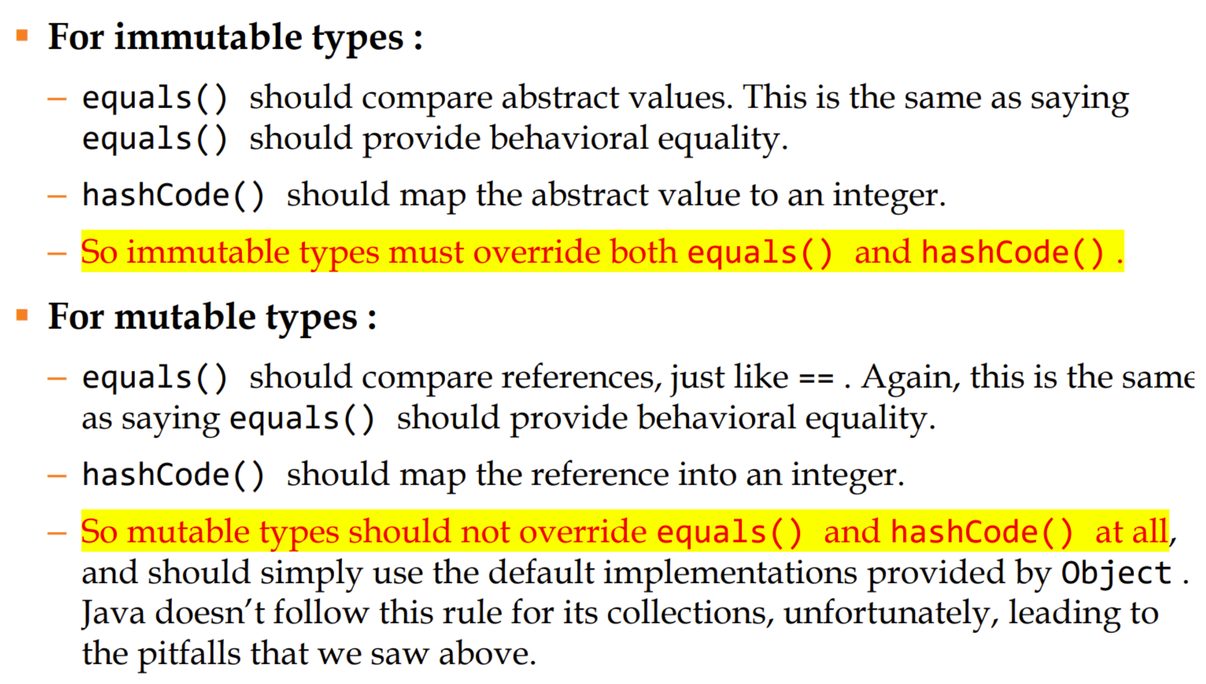
equals函数的contract：

1. equals必须定义一个等价关系：自反、传递、对称
2. 除非对象被修改了，否则调用多次equals应同样的结果
3. for a non-null reference x , x.equals(null) should return false
4. “相等”的对象，其hashCode()的结果必须一致

override hashcode方法

要保证“相等”的对象，其hashCode()的结果必须一致

总结



- **For immutable types :**
  - `equals()` should compare abstract values. This is the same as saying `equals()` should provide behavioral equality.
  - `hashCode()` should map the abstract value to an integer.
  - So immutable types must override both `equals()` and `hashCode()` .
- **For mutable types :**
  - `equals()` should compare references, just like `==` . Again, this is the same as saying `equals()` should provide behavioral equality.
  - `hashCode()` should map the reference into an integer.
  - So mutable types should not override `equals()` and `hashCode()` at all, and should simply use the default implementations provided by `Object` . Java doesn’t follow this rule for its collections, unfortunately, leading to the pitfalls that we saw above.

Unchecked Exception Classes：在编程和编译的时候，IDE与编 译器均不会给出任何错误提示

Checked Exception Classes：在编译阶段会有错误提示

不应该throws Unchecked Exception Classes，应为这是程序员可以处理的

子类型方法可以抛出更具体的异 常，也可以不抛出任何异常

instanceOf：它是dynamic-checking,所以能不用就不用，尽量使用多态来取代它