# 3.5 Equality in ADT and OOP

一级标题，二级标题，三级标题，四级标题，注，拓展，不懂，Ex

MIT 6.031：15

| Safe from bugs | Easy to understand | Ready for change |
| --- | --- | --- |
| Correct today and correct in the unknown future. | Communicating clearly with future programmers, including future you. | Designed to accommodate change without rewriting. |

## 目录

# 3.5.1 two ways to regard equality

## Using the abstraction function.

a equals b if and only if AF(a) = AF(b).（在抽象函数的值域的表现完全相同）

## Using observation.

two objects are equal if and only if they cannot be distinguished by calling any operations of the abstract data type.（被观察器观察到的结果完全相同）

Ex:

Consider the set expressions {1,2} and {2,1}. Using the observer operations available for sets, cardinality |···| and membership ∈, these expressions are indistinguishable:

A. | {1,2} | = 2 and |{2,1}| = 2
B. 1 ∈ {1,2} is true, and 1 ∈ {2,1} is true
C. 2 ∈ {1,2} is true, and 2 ∈ {2,1} is true
D. 3 ∈ {1,2} is false, and 3 ∈ {2,1} is false
E. ··· and so on

注: Note that “operations of the abstract data type” means the operations that are part of the spec of the ADT.注意当以观察器定义相等时，观察器所指为此 ADT 中规定的观察器。此处注意与 java 语言自带的“==”，“equals”，“hashcode”进行区别。

Ex

For example, Java’s == can detect that two objects representing identical sets are actually stored at different places in memory. System.identityHashCode() computes a hash code based on an object’s memory address, so it could observe the same distinction. But neither of these operations would be part of the spec of the set ADT, so the distinctions they make are not considered when deciding if two set objects are equal by observation.

# 3.5.2Equality of immutable types

## java 中“==”和“equals()”的定义区别

## The “==” operator compares references

The == operator compares references. More precisely, it tests **reference equality**. Two references are == if they point to the same storage in memory. In terms of the snapshot diagrams we’ve been drawing, two references are == if their arrows point to the same object bubble.（“==”所指为两个对象的内存相等）

## The “equals()” operation compares object contents

The equals() operation compares object contents – in other words, **object equality**, in the sense that we've been talking about in this reading. The equals operation has to be defined appropriately for every abstract data type.（比较的是值的内容）

## 拓展：其他编程语言中“==”和“equals()”的区别

| | *reference equality* | *object equality* |
|---|---|---|
| Java | `==` | `equals()` |
| Objective C | `==` | `isEqual:` |
| C# | `==` | `Equals()` |
| Python | `is` | `==` |
| Javascript | `==` | n/a |

注：上述两个拓展仅仅关于 object，在基本数据类型中，“==”和“equals()”有着不同的使用。

注：“==”一般无法改变其含义，其更多指的是 **reference equality**。但是“equals()”是可以改变的，当定义一个 ADT 时，equals 要适当的进行改变。

注：应该总是使用“equals()“来判断相等（如果判断逻辑是内存地址相等，则不需要重写 Object.equals()，此时“equals()”等价于“==”；如果判断逻辑是特定的，则需要重写 Object.equals()，则“equals()“调用的是期望的判断逻辑。无论哪种情况，使用”equals()“都能达到期望的目的）

## 拓展：“equals()”的默认形式

```
public class Object {
    ...
    public boolean equals(Object that) {
        return this == that;
    }
}
```

## 在不可变数据类型中的“equals()“的观察等价性实现

In other words, the default meaning of equals() is the same as reference equality. For

immutable data types, this is almost always wrong. So you have to override the equals() method, replacing it with your own implementation.

"equals()"的默认形式是采用 **reference equality**(引用相等)，这与我们对于"equals()"的定义是不同的，因此我们需要根据需要进行修改。

Ex: A good way to implement equals()

```
@Override
public boolean equals(Object that) {
    return that instanceof Duration && this.sameValue((Duration)that);
}

// returns true iff this and that represent the same abstract value
private boolean sameValue(Duration that) {
    return this.getLength() == that.getLength();
}
```

注: 当需要对"equals"进行覆盖（重写）操作时，尽量进行"@override"的标注

Ex:

```
public class Duration extends Object {
    // explicit method that we declared:
    public boolean equals(Duration that) {
        return this.getLength() == that.getLength();
    }
    // implicit method inherited from Object:
    public boolean equals(Object that) {
        return this == that;
    }
}
```

# 3.5.4 The Object contract of equals(实现 equals 需要遵循)

## Definition

所有的类都继承于 object 类，需要遵守 object 类的规约，故而实现 equals 也需要满足 object 的关于 equals 的规约。

The specification of the Object class is so important that it is often referred to as the Object Contract. The contract can be found in the method specifications for the Object class. Here we will focus on the contract for equals. When you override the equals method, you must adhere to its general contract. It states that:

A. equals must define an equivalence relation – that is, a relation that is reflexive, symmetric, and transitive;（equals 是一个等价关系，必须满足自反，传递，对称）
B. equals must be consistent: repeated calls to the method must yield the same result provided no information used in equals comparisons on the object is modified;（一致性：多次调用结果相同）
C. for a non-null reference x, x.equals(null) should return false;（为了满足对称性，对 null 做出了定义）
D. hashCode must produce the same result for two objects that are deemed equal by the equals method.（hasnCode 必须保持与 equals 一致，原因大概就是该对象可能被放入到由 hashCode 原理形成的容器类型）

## Breaking the equivalence relation

If it isn't, then operations that depend on equality (like sets, searching) will behave erratically and unpredictably. You don't want to program with a data type in which sometimes a equals b, but b doesn't equal a. Subtle and painful bugs will result.（如果你不保证这个，可能会产生极其隐秘的 bug）

Ex：一个传递性破坏的例子

Here's an example of how an innocent attempt to make equality more flexible can go wrong. Suppose we wanted to allow for a tolerance in comparing Duration objects, because different computers may have slightly unsynchronized clocks:

```
@Override
public boolean equals(Object that) {
    return that instanceof Duration && this.sameValue((Duration)that);
}

private static final int CLOCK_SKEW = 5; // seconds

// returns true iff this and that represent the same abstract value with
in a clock-skew tolerance
private boolean sameValue(Duration that) {
    return Math.abs(this.getLength() - that.getLength()) <= CLOCK_SKEW;
}
```

## Breaking hash tables

Now it should be clear why the Object contract requires equal objects to have the same hashcode. If two equal objects had distinct hashcodes, they might be placed in different slots. So if you attempt to lookup a value using a key equal to the one with which it was inserted, the lookup may fail.（这里也就展现了为什么需要保证 hashCode 和 equals 需要保持一致，因为如果两个被判断相等对象，计算出的 hashCode 不同，二者将会被放入不同的槽（哈希桶）中，这与我们所期望的是不一样的，这种情况会在使用与 hash 有关数据结构上出错）

# 拓展：hashCode

HashCode 原理

A hash table is a representation for a mapping: an abstract data type that maps keys to values.（就是一个映射（序对）key 和 value）

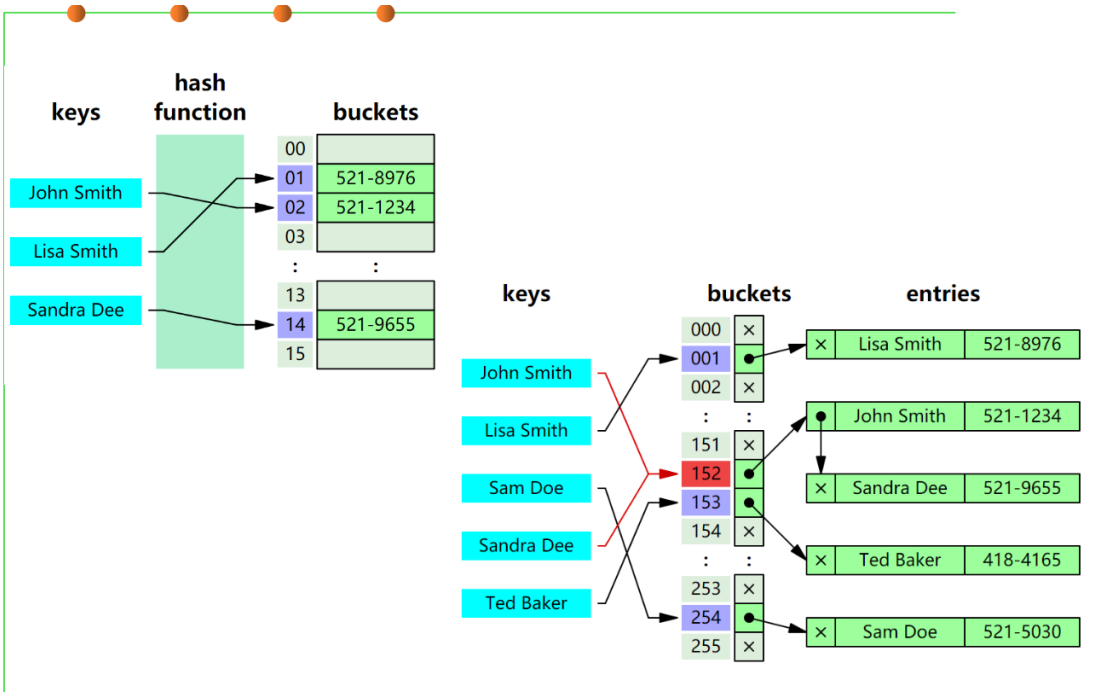
The rep invariant of a hash table includes the fundamental constraint that a key can be found by starting from the slot determined by its hash code.（Hashtable 的 RI 中基本要求就是 key 在 slot 中的位置由 hashcode 确定）

Hash tables offer constant time lookup, so they tend to perform better than trees or lists. Keys don't have to be ordered, or have any particular property, except for offering equals and hashCode.

Here's how a hash table works. It contains an array that is initialized to a size corresponding to the number of elements that we expect to be inserted. （初始化一个需要的特定大小的数组，注意此数组的元素可不是数）

When a key and a value are presented for insertion, we compute the hashcode of the key, and convert it into an index in the array's range (e.g., by a modulo division). The value is then inserted at that index.（当插入一组 key 和 value 时，先计算 key 的哈希值，并且依照哈希值，将 value 插入对应数组的槽之中）

Hashcodes are designed so that the keys will be spread evenly over the indices. But occasionally a conflict occurs, and two keys are placed at the same index. So rather than holding a single value at an index, a hash table actually holds a list of key/value pairs, usually called a hash bucket. A key/value pair is implemented in Java simply as an object with two fields. On insertion, you add a pair to the list in the array slot determined by the hash code. For lookup, you hash the key, find the right slot, and then examine each of the pairs until one is found whose key equals the query key.（一般而言，由 key 计算出的 hashCode 会均匀分布，但是，也会出现冲突，不同的 key 计算出相同的哈希值，为了解决这样的麻烦，故而采用槽（哈希桶）的概念，将之作为数组的元素）




注:

Whenever it is invoked on the same object more than once during an execution of an application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. (程序中多次调用同一对象的 hashCode 方法，都要返回相同值)

This integer need not remain consistent from one execution of an application to another execution of the same application.(不要求程序的多 次执行时相同)

Hashcode 默认形式

```
public class Object {
  ...
  public boolean equals(Object that) { return this == that; }
  public int hashCode() { return /* the memory address of this */; }
}
```

Hashcode 的实现

具体情况进行具体分析，由 equals 的改变进行相符合的 hashcode 的修改。

A simple and drastic way to ensure that the contract is met is for hashCode to always return some constant value, so every object's hash code is the same. This satisfies the Object contract, but it would have a disastrous performance effect, since every key will be stored in the same slot, and every lookup will degenerate to a linear search along a long list.（一个最为简单的 hashcode 的实现形式就是所有对象的 hashcode 均返回一个常数，这样会完全满足上述的 equals 定义，但是这样会极大的降低 hashcode 的性能，他将退化为一个长数组）

The standard way to construct a more reasonable hash code that still satisfies the contract is to compute a hash code for each component of the object that is used in the determination of equality (usually by calling the hashCode method of each component), and then combining these, throwing in a few arithmetic operations. For Duration, this is easy, because the abstract value of the class is already an integer value:（一个 hashcode 的标准实

现形式就是将 object 中的各个组成部分的 hashcode 进行组合，以满足相符合的 equals 的需要）

# 拓展：运算符 instanceof

Definition

The instanceof operator tests whether an object is an instance of a particular type.

如果 object 是 class（该类） or subclass（子类） or interface that this class implenment 或者是某个实现了的接口） 的一个实例，则 instanceof 运算符返回 true。如果 object 不是指定类的一个实例，或者 object 是 null，则返回 false。

dynamic type checking

用法：

In general, using instanceof in object-oriented programming is a bad smell. In good object-oriented programming, instanceof is disallowed anywhere except for implementing equals.（尽量别用，只在实现 equals 是进行使用）

# 3.5.5 The equality of mutable types

## observational equality

observational equality means that two references cannot be distinguished now, in the current state of the program. This tests whether the two references “look” the same for the current state of the object.(仅仅检验在程序运行中某一时刻两个对象的相等)

## behavioral equality

behavioral equality means that two references cannot be distinguished now or in the future, even if a mutator is called to change the state of one object but not the other. This tests whether the two references will “behave” the same, in this and all future states.（保证在程序运行过程中，两个对象一直保持相等，无论 client 是否对其进行变值操作，在某种程度上等价于地址空间判断，不需要重写）

## for mutable types

The lesson we should draw from this example is that equals() should implement behavioral equality. Otherwise instances of the type cannot be safely stored in a data structure that depends on hashing, like HashMap and HashSet.（可变数据类型所使用的是行为等价性而并非是观察等价性，因为采用观察等价性会在 hashcode 的方面产生隐秘的 bug，比如：下面的拓展，而不可变数据类型可以灵活使用）

# 拓展：an example of immutable types Breaking a HashSet's rep invariant

Using observational equality for mutable types seems at first like a reasonable idea.（使用行为相等判断可变 ADT 时很好的）

But the presence of mutators unfortunately leads to subtle bugs, because it means that equality isn't consistent over time. Two objects that are observationally equal at one moment in the program may stop being equal after a mutation.（但是，当对对象进行变值后，可能会产生隐秘的 bug，这是因为因为行为相等性不可保持）

Ex:

This fact allows us to easily break the rep invariants of other collection data structures. Suppose we make a `List`, and then drop it into a `HashSet`:

```
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
```

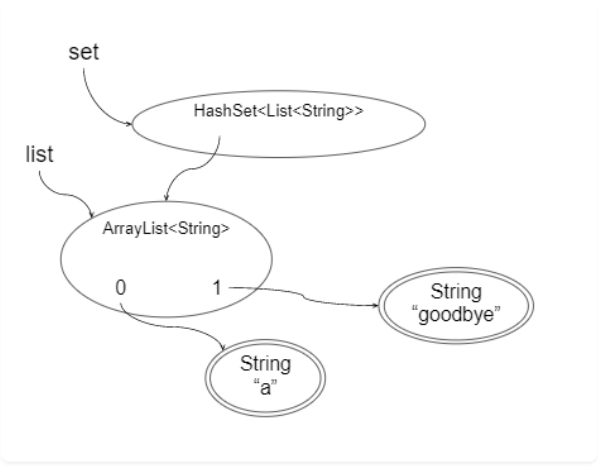
But now we mutate the list:

```
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!
}
```

Reasons:

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 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.（虽然 set 中的 list 的值进行更改，但是这样更改不会造成 list 的存储位置的变化，但其 hashcode 已经被改变）

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.（此处就破坏了哈希表的不变量，形成了隐秘的 bug）

# 3.5.6 Summary

## For immutable types:

A. Behavioral equality is the same as observational equality.
B. equals() must be overridden to compare abstract values.
C. hashCode() must be overriden to map the abstract value to an integer.

## For mutable types:

A. Behavioral equality is different from observational equality.
B. equals() should generally not be overriden, but inherit the implementation from Object that compares references, just like ==.不需要重写 equals，由内存空间判断即可
C. hashCode() should likewise not be overridden, but inherit Object's implementation that maps the reference into an integer.不需要重写 hashcode，返回地址即可
D. if it needs a notion of observational equality (whether two mutable objects "look" the same in the current state), it's better to define a completely new operation, which might be called similar() or sameValue().如果非要判断内容，可以另外构造函数

## all must respect The Object contract

# 拓展：Autoboxing and Equality 的 equals()

definition

The object type implements equals() in the correct way, so that if you create two Integer objects with the same value, they'll be equals() to each other.（基本数据类型的包装对象类型都较好的实现了 equals 方法，其使用的是观察等价性，即比较其中的值，而非地址）

But there's a subtle problem here; == is overloaded. For reference types like Integer, it

implements reference equality.（但是，在其包装类型 equals 的更改中，并未使用重写将“==“也更改为行为等价性比较，而是将之原样保存，其仍然判断地址空间）
Ex:

```
Integer x = new Integer(3);
Integer y = new Integer(3);
x.equals(y) → true
```

```
x == y // returns false
```

```
(int)x == (int)y // returns true
```

注：最后一个转换是十分容易理解的，可以参照我的博客《相等与同一》

```
Map<String, Integer> a = new HashMap(), b = new HashMap();
a.put("c", 130); // put ints into the map
b.put("c", 130);
a.get("c") == b.get("c") → ?? // what do we get out of the map?
```

结果为 false, 因为 map 将 130 进行插入时, 进行了隐式类型转换, 将之由 int 转换为 integer。

```
Map<String, Integer> a = new HashMap<>();
Map<String, Integer> b = new HashMap<>();
a.put("c", 1);
b.put("c", 1);
System.out.println (a.get("c")== b.get("c"));
```

结果为 true，因为 java 在处理 map 的插入操作时，进行了特别的优化处理，当 value 的值在-128 到+127 之间时，对于同一个 int 类型的值，只会申请一个地址空间，而让所有引用指向该地址空间。