Chapter 3: Abstract Data Type (ADT) and Object-Oriented Programming (OOP)

3.5 Equality in ADT and OOP

Wang Zhongjie
rainy@hit.edu.cn

March 20, 2018

Objective of this lecture

- Understand equality defined in terms of the abstraction function, an equivalence relation, and observations. 站在观察者角度，利用AF，定义对象之间的等价关系
- Differentiate between reference equality and object equality. 引用等价性和对象等价性
- Differentiate between strict observational and behavioral equality for mutable types. 可变数据类型的观察等价性和行为等价性
- Understand the `Object` contract and be able to implement equality correctly for mutable and immutable types. 理解Object的契约，正确实现等价关系判定

Outline

- **What is and why equality?**
- **Three ways to regard equality**
- `==` **vs.** `equals()`
- **Equality of immutable types**
- **The** `Object` **contract**
- **Equality of Mutable Types**
- **Autoboxing and Equality**

在很多场景下，需要判定两个对象是否“相等”，例如：判断某个Collection中是否包含特定元素。

==和equals()有和区别？如何为自定义ADT正确实现equals()？

Reading

- **MIT 6.031：15**

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

1 What is and why equality?

Equality operation on an ADT

- **ADT is** data abstraction by creating types that are characterized by their operations, not by their representation. ADT是对数据的抽象，体现为一组对数据的操作

- For an abstract data type, the abstraction function (AF) explains how to interpret a concrete representation value as a value of the abstract type, and we saw how the choice of abstraction function determines how to write the code implementing each of the ADT's operations. 抽象函数AF：内部表示→抽象表示

- The abstraction function (AF) gives a way to cleanly define the equality operation on an ADT. 基于抽象函数AF定义ADT的等价操作

Equality of values in a data type?

- In the physical world, every object is distinct – at some level, even two snowflakes are different, even if the distinction is just the position they occupy in space. 现实中的每个对象实体都是独特的
- So two physical objects are never truly “equal” to each other; they only have degrees of similarity. 所以无法完全相等，但有“相似性”
- In the world of human language, however, and in the world of mathematical concepts, you can have multiple names for the same thing. 在数学中，“绝对相等”是存在的
  - So it’s natural to ask when two expressions represent the same thing: 1+2, √9, and 3 are alternative expressions for the same ideal mathematical value.

2 Three ways to regard equality

Using AF or using a relation

- **Using an abstraction function** . Recall that an abstraction function `f`: `R → A` maps concrete instances of a data type to their corresponding abstract values. To use `f` as a definition for equality, we say that `a` equals `b` if and only if `f(a)=f(b)`. AF映射到同样的结果，则等价
- **Using a relation** . An *equivalence* is a relation `E ⊆ T x T` that is:
  - reflexive: `E(t,t) ∀t∈T`
  - symmetric: `E(t,u) ⇒ E(u,t)`
  - transitive: `E(t,u) ∧ E(u,v) ⇒ E(t,v)`
  - To use `E` as a definition for equality, we would say that `a` equals `b` if and only if `E(a,b)`. 等价关系：自反、对称、传递
- These two notions are equivalent.
  - An equivalence relation induces an abstraction function (the relation partitions `T`, so `f` maps each element to its partition class).
  - The relation induced by an abstraction function is an equivalence relation.

Using observation

- A third way we can talk about the equality between abstract values is in terms of what an outsider (a client) can observe about them:
- **Using observation**. We can say that two objects are equal when they cannot be distinguished by observation – every operation we can apply produces the same result for both objects. 站在外部观察者角度
  - Consider the set expressions `{1,2}` and `{2,1}`. Using the observer operations available for sets, cardinality |…| and membership ∈, these expressions are indistinguishable:
    - `|{1,2}| = 2` and `|{2,1}| = 2`
    - `1 ∈ {1,2}` is true, and `1 ∈ {2,1}` is true
    - `2 ∈ {1,2}` is true, and `2 ∈ {2,1}` is true
    - `3 ∈ {1,2}` is false, and `3 ∈ {2,1}` is false
- In terms of ADT, "observation" means calling operations on the objects. So two objects are equal if and only if they cannot be distinguished by calling any operations of the abstract data type.

Example: Duration

- Here's a simple example of an immutable ADT.

```java
public class Duration {
    private final int mins;
    private final int secs;
    // rep invariant:
    //    mins >= 0, secs >= 0
    // abstraction function:
    //    represents a span of time of mins minutes and secs seconds

    /** Make a duration lasting for m minutes and s seconds. */
    public Duration(int m, int s) {
        mins = m; secs = s;
    }
    /** @return length of this duration in seconds */
    public long getLength() {
        return mins*60 + secs;
    }
}
```

仅代表一个时间长度

equals(Object that)

- Now which of the following values should be considered equal?

```java
Duration d1 = new Duration (1, 2);
Duration d2 = new Duration (1, 3);
Duration d3 = new Duration (0, 62);
Duration d4 = new Duration (1, 2);
```

Think in terms of both the abstraction-function definition of equality, and the observational equality definition.

3 == vs. `equals()`

== vs. equals()

- **Java has two different operations for testing equality, with different semantics.**
  - **The == operator compares references.** It tests referential equality. Two references are == if they point to the same storage in memory. In terms of the snapshot diagrams, two references are == if their arrows point to the same object bubble. 引用等价性
  - **The `equals()` operation compares object contents** – in other words, object equality. 对象等价性
- **The `equals` operation has to be defined appropriately for every abstract data type. 在自定义ADT时，需要重写Object的equals()**
  - When we define a new data type, it's our responsibility to decide what object equality means for values of the data type, and implement the `equals()` operation appropriately.

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

The == operator vs. equals method

- **For primitives you *must* use** **== 对基本数据类型，使用==判定相等**
- **For object reference types 对对象类型，使用`equals()`**
  - The == operator provides *identity semantics* 如果用==，是在判断两个对象身份标识 ID是否相等（指向内存里的同一段空间）
  - Exactly as implemented by `Object.equals`
  - Even if `Object.equals` has been overridden, this is seldom what you want!
  - You should (almost) always use `.equals`
- Using == on an object reference is a **bad smell in code**

```
if (input == "yes") // A bug!!!
```

?

内容等价性.

string

Tips for overriding a method

- **If you want to override a method:**
  - Make sure signatures match
  - Use `@Override` so compiler has your back
  - *Do* copy-and-paste declarations (or let IDE do it for you)

HARBIN INSTITUTE OF TECHNOLOGY

4 Equality of immutable types

Equality of Immutable Types

- **The `equals()` method is defined by `Object`, and its default implementation looks like this:**

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

- **The default meaning of `equals()` is the same as referential equality.** 在Object中实现的缺省equals()是在判断引用等价性
- **For immutable data types, this is almost always wrong.** 这通常不是程序员所期望的
- **We have to override the `equals()` method, replacing it with our own implementation.** 因此，需要重写

`equals()` for this example

- `equals()` **for the class** `Duration`**:**

```
public class Duration {
    ...
    // Problematic definition of equals()
    public boolean equals(Duration that) {
        return this.getLength() == that.getLength();
    }
}
```

这不是override，
而是overload！

- **How about this code?**

```
Duration d1 = new Duration (1, 2);
Duration d2 = new Duration (1, 2);
Object o2 = d2;
d1.equals(d2)
d1.equals(o2)
```

???

Even though `d2` and `o2` end up referring to the very same object in memory, you still get different results for them from `equals()` .

- **Why?**

What's going on?

- **The class `Duration` has overloaded the `equals()` method, because the method signature was not identical to `Object`'s.**
- **We actually have two `equals()` methods in `Duration`:**
  - An implicit `equals(Object)` inherited from `Object`
  - The new `equals(Duration)`.

```
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;
    }
}
```

- **Java compiler selects between overloaded operations using the compile-time type of the parameters. Dynamic dispatch!**

What's going on?

- **If we pass an `Object` reference, as in `d1.equals(o2)`, we end up calling the `equals(Object)` implementation.**
- **If we pass a `Duration` reference, as in `d1.equals(d2)`, we end up calling the `equals(Duration)` version.**
- **This happens even though `o2` and `d2` both point to the same object at runtime! Equality has become inconsistent.**

```
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;
    }
}
```

```
Duration d1 = new Duration (1, 2);
Duration d2 = new Duration (1, 2);
Object o2 = d2;
d1.equals(d2) → true
d1.equals(o2) → false
```

Overload vs. override

- **It's easy to make a mistake in the method signature, and overload a method when you meant to override it.**
- **Java's annotation `@Override` should be used whenever your intention is to override a method in your superclass.**
- **With this annotation, the Java compiler will check that a method with the same signature actually exists in the superclass, and give you a compiler error if you've made a mistake in the signature.**

```java
@Override
public boolean equals (Object thatObject) {
    if (!(thatObject instanceof Duration)) return false;
    Duration thatDuration = (Duration) thatObject;
    return this.getLength() == thatDuration.getLength();
}
```

What does this print?

```
public class Name {
   private final String first, last;
   public Name(String first, String last) {
      if (first == null || last == null)
          throw new NullPointerException();
      this.first = first;
      this.last = last;
   }
   public boolean equals(Name o) {
      return first.equals(o.first) && last.equals(o.last);
   }
   public int hashCode() {
      return 31 * first.hashCode() + last.hashCode();
   }
   public static void main(String[] args) {
      Set<Name> s = new HashSet<>();
      s.add(new Name("Mickey", "Mouse"));
      System.out.println(s.contains(new Name("Mickey", "Mouse")));
   }
}
```

How about this?

```
public class Name {
   private final String first, last;
   public Name(String first, String last) {
      if (first == null || last == null)
          throw new NullPointerException();
      this.first = first;
      this.last = last;
   }
   @Override public boolean equals(Object o) {
      if (!(o instanceof Name))
           return false;
      Name n = (Name) o;
      return n.first.equals(first) && n.last.equals(last);
   }
   public int hashCode() {
      return 31 * first.hashCode() + last.hashCode();
   }
   public static void main(String[] args) {
      Set<Name> s = new HashSet<>();
      s.add(new Name("Mickey", "Mouse"));
      System.out.println(s.contains(new Name("Mickey", "Mouse")));
   }
}
```

equals Override Example

操作符

```
public final class PhoneNumber {

    private final short areaCode;
    private final short prefix;
    private final short lineNumber;

    @Override
    public boolean equals(Object o) {
        if (!(o instanceof PhoneNumber)) // Does null check
            return false;

        PhoneNumber pn = (PhoneNumber) o;
        return pn.lineNumber == lineNumber
               && pn.prefix == prefix
               && pn.areaCode == areaCode;
    }
    ...
}
```

instanceof()

- **The `instanceof` operator tests whether an object is an instance of a particular type.**
- **Using `instanceof` is dynamic type checking, not the static type checking.**
- **In general, using `instanceof` in object-oriented programming is a bad smell. It should be disallowed anywhere except for implementing equals .**
- **This prohibition also includes other ways of inspecting objects' runtime types.**
  - For example, `getClass()` is also disallowed.

5 The `Object` contract

The contract of `equals()` in `Object`

- **When you override the `equals()` method, you must adhere to its general contract:**
  - `equals` must define an equivalence relation – that is, a relation that is reflexive, symmetric, and transitive;
  - `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;
  - for a non-null reference `x`, `x.equals(null)` should return `false`;
  - `hashCode()` must produce the same result for two objects that are deemed equal by the `equals` method.

The `equals` contract

- The equals method implements an **equivalence relation**:
  - **Reflexive**: For any non-null reference value `x`, `x.equals(x)` must return true.
  - **Symmetric**: For any non-null reference values `x` and `y`, `x.equals(y)` must return true if and only if `y.equals(x)` returns true.
  - **Transitive**: For any non-null reference values `x, y, z`, if `x.equals(y)` returns true and `y.equals(z)` returns true, then `x.equals(z)` mus return true.
  - **Consistent**: For any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
  - For any non-null reference value x, `x.equals(null)` must return false.
- **`equals` is a global equivalence relation over all objects.**

The `equals` contract in English

- **Reflexive** – every object is equal to itself
- **Symmetric** – if `a.equals(b)` then `b.equals(a)`
- **Transitive** – if `a.equals(b)` and `b.equals(c)`, then `a.equals(c)`
- **Consistent**– equal objects stay equal unless mutated
- **"Non-null"** – `a.equals(null)` returns false
- Taken together these ensure that equals is a global **equivalence relation** over all objects

Breaking the Equivalence Relation

- **We have to make sure that the definition of equality implemented by `equals()` is actually an equivalence relation as defined earlier: reflexive, symmetric, and transitive.**
  - 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.

```
private static final int CLOCK_SKEW = 5; // seconds

@Override
public boolean equals (Object thatObject) {
    if (!(thatObject instanceof Duration)) return false;
    Duration thatDuration = (Duration) thatObject;
    return Math.abs(this.getLength() - thatDuration.getLength()) <= CLOCK_SKEW;
}
```

Which property of the equivalence relation is violated?

Breaking the Equivalence Relation

```java
private static final int CLOCK_SKEW = 5; // seconds

@Override
public boolean equals (Object thatObject) {
    if (!(thatObject instanceof Duration)) return false;
    Duration thatDuration = (Duration) thatObject;
    return Math.abs(this.getLength() - thatDuration.getLength()) <= CLOCK_SKEW;
}
```

不传递

```java
Duration d_0_60 = new Duration(0, 60);
Duration d_1_00 = new Duration(1, 0);
Duration d_0_57 = new Duration(0, 57);
Duration d_1_03 = new Duration(1, 3);
```

```java
d_0_60.equals(d_1_00)
d_1_00.equals(d_0_60)
d_1_00.equals(d_1_00)
d_0_57.equals(d_1_00)
d_0_57.equals(d_1_03)
d_0_60.equals(d_1_03)
```

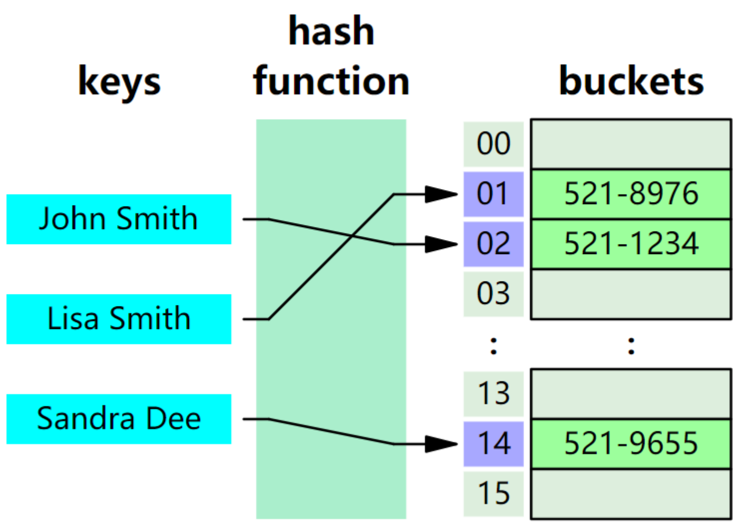
Breaking Hash Tables

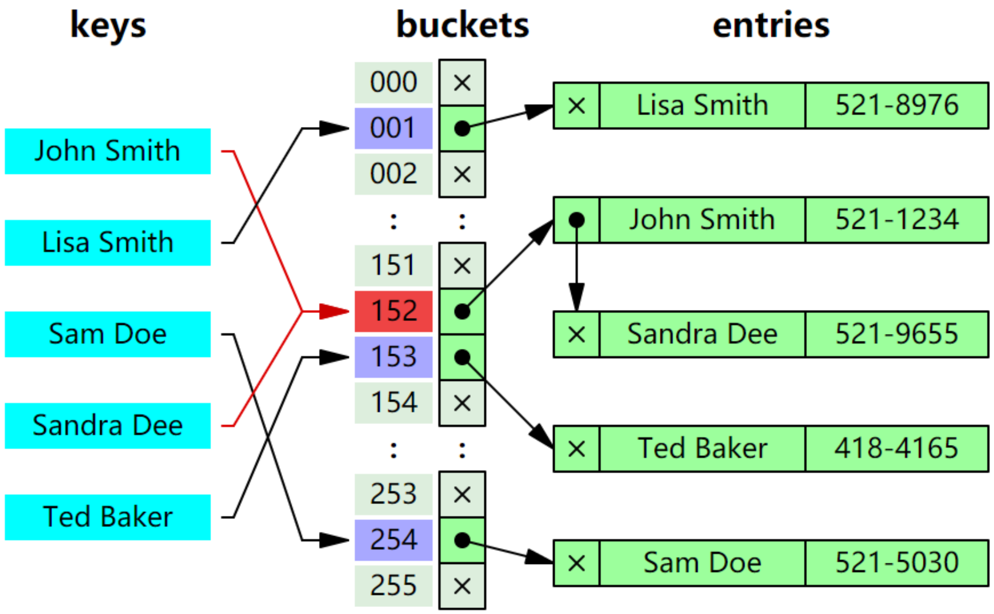
- **A hash table is a representation for a mapping: an abstract data type that maps keys to values.**
  - Hash tables offer constant time lookup, so they tend to perform better than trees or lists. Keys don't have to be ordered, or have any particular property, except for offering `equals` and `hashCode` .
- **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.
  - 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.
- **The rep invariant of a hash table includes the fundamental constraint that keys are in the slots determined by their hash codes.**

Breaking Hash Tables

- 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.

Hash Tables


hash
function
keys
buckets
John Smith
Lisa Smith
Sandra Dee
00
01
02
03
:
13
14
15
521-8976
521-1234
:
521-9655


keys
buckets
entries
John Smith
Lisa Smith
Sam Doe
Sandra Dee
Ted Baker
000
001
002
:
151
152
153
154
:
253
254
255
Lisa Smith
521-8976
John Smith
521-1234
Sandra Dee
521-9655
Ted Baker
418-4165
Sam Doe
521-5030

The hashCode contract

- 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.
  - This integer need not remain consistent from one execution of an application to another execution of the same application.
- **If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.**
  - It is not required that if two objects are unequal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results.
  - However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

The `hashCode` contract in English

- Equal objects **must** have equal hash codes
  - If you override equals you must override `hashCode`
- Unequal objects **should** have different hash codes
  - Take all value fields into account when constructing it
- Hash code must not change unless object mutated

Overriding hashCode()

- **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.
- **The standard 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:**

```
@Override
public int hashCode() {
    return (int) getLength();
}
```

Breaking Hash Tables

- **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.
- **`Object`'s default `hashCode()` implementation is consistent with its default `equals()` :**

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

In our example…

- For `Duration` , since we haven't overridden the default `hashCode()` yet, we're currently breaking the `Object` contract:

```
Duration d1 = new Duration(1, 2);
Duration d2 = new Duration(1, 2);
d1.equals(d2) → true
d1.hashCode() → 2392
d2.hashCode() → 4823
```

- `d1` and `d2` are equal, but they have different hash codes.
- How to fix it?

Overriding hashCode()

- **Recent versions of Java now have a utility method `Objects.hash()` that makes it easier to implement a hash code involving multiple fields.**

- **Note that if you don't override `hashCode()` at all, you'll get the one from `Object`, which is based on the address of the object.**
- **If you have overridden `equals` , this will mean that you will have almost certainly violated the contract – 两个equal的objects，一定要有同样的hashcode.**
- **A general rule:**

Always override `hashCode()` when you override `equals()`.

hashCode override example

```
public final class PhoneNumber {

    private final short areaCode;
    private final short prefix;
    private final short lineNumber;

    @Override
    public int hashCode() {
        int result = 17; // Nonzero is good
        result = 31 * result + areaCode; // Constant must be odd
        result = 31 * result + prefix; // " " " "
        result = 31 * result + lineNumber; // " " " "

        return result;
    }
    ...
}
```

Alternative hashCode override

- **Less efficient, but otherwise equally good!**

```
public final class PhoneNumber {

   private final short areaCode;
   private final short prefix;
   private final short lineNumber;

   @Override
   public int hashCode() {
      return arrays.hashCode(areaCode, prefix, lineNumber);
   }
   ...
}
```

An example

```
class Person {
  private String firstName;
  private String lastName;
  ...

  public boolean equals(Object obj) {
      if (!(obj instanceof Person)) return false;
      Person that = (Person) obj;
      return this.lastName.toUpperCase().
               equals(that.lastName.toUpperCase());
  }

  public int hashCode() {
      // TODO
  }
}
```

```
  return 42;
  return firstName.toUpperCase();
  return lastName.toUpperCase().hashCode();
? return firstName.hashCode() + lastName.hashCode();
```

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

6 Equality of Mutable Types

Equality of Mutable Types

- **Equality**: two objects are equal when they cannot be distinguished by observation.
- **With mutable objects, there are two ways to interpret this:**
  - 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**, since it tests whether the two objects "look" the same, in the current state of program. 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 *any* observation, even state changes. This interpretation allows calling any methods on the two objects, including mutators. This is called **behavioral equality** , since it tests whether the two objects will "behave" the same, in this and all future states. 行为等价性：调用对象的任何方法都展示出一致的结果
- **Note: for immutable objects**, observational and behavioral equality are identical, because there aren't any mutator methods.

Equality in Java for mutable type

- **For mutable objects, it's tempting to implement strict observational equality. 对可变类型来说，往往倾向于实现严格的观察等价性**
  - Java uses observational equality for most of its mutable data types (such as `Collections`), but other mutable classes (like `StringBuilder`) use behavioral equality.
  - If two distinct `List` objects contain the same sequence of elements, then `equals()` reports that they are equal.

- But using observational equality leads to subtle bugs, and in fact allows us to easily break the rep invariants of other collection data structures. 但在有些时候，观察等价性可能导致bug，甚至可能破坏RI

An example

- **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!
}
```

What's going on?

- `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)，你用乘机的时候就无法匹配到你的身份证照片了…

What's going on?

- **Great care must be exercised if mutable objects are used as set elements.**
- **The behavior of a set is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is an element in the set.**

- **The Java library is unfortunately inconsistent about its interpretation of `equals()` for mutable classes. Collections use observational equality, but other mutable classes (like `StringBuilder`) use behavioral equality.**

Lessons learned from this example

- `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对象的两个方法即可。
- 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。

The Final Rule for `equals()` and `hashCode()`

- **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.

Exercise

- `Bag<E>` **is a mutable ADT representing what is often called a multiset, an unordered collection of objects where an object can occur more than once. It has the following operations:**

```
/** make an empty bag */
public Bag<E>()

/** modify this bag by adding an occurrence of e, and return
this bag */
public Bag<E> add(E e)

/** modify this bag by removing an occurrence of e (if any),
and return this bag */
public Bag<E> remove(E e)

/** return number of times e occurs in this bag */
public int count(E e)
```

Exercise

```
Bag<String> b1 = new Bag<>().add("a").add("b");
Bag<String> b2 = new Bag<>().add("a").add("b");
Bag<String> b3 = b1.remove("b");
Bag<String> b4 = new Bag<>().add("b").add("a");
```

	If `Bag` is implemented with **behavioral equality**	If `Bag` implemented **observational equality** despite the dangers
`b1.count("a") == 1`		
`b1.count("b") == 1`		
`b2.count("a") == 1`	`b1.equals(b2)`	`b1.equals(b2)`
`b2.count("b") == 1`	`b1.equals(b3)`	`b1.equals(b3)`
`b3.count("a") == 1`	`b1.equals(b4)`	`b1.equals(b4)`
`b3.count("b") == 1`	`b2.equals(b3)`	`b2.equals(b3)`
`b4.count("a") == 1`	`b2.equals(b4)`	`b2.equals(b4)`
`b4.count("b") == 1`	`b3.equals(b1)`	`b3.equals(b1)`

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

7 Autoboxing and Equality

Autoboxing and Equality

- **Primitive types and their object type equivalents, e.g.,** `int` **and** `Integer`**.**
- **If you create two Integer objects with the same value, they'll be** `equals()` **to each other.**

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

- But what if `x==y`? ----- `False` (because of referential equality)
- But what if `(int) x == (int) y`? ----`True`
- What's the result of this code?

```
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?
```

Autoboxing and Equality

放入Map的时候，
自动将int 130转为
了Integer

```
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?
```

取出来的时候，得
到的是Integer类型

对Integer类型的==，
是reference
equivalence判断

Summary

Summary

- **Equality is one part of implementing an abstract data type (ADT).**
  - Equality should be an equivalence relation (reflexive, symmetric, transitive).
  - Equality and hash code must be consistent with each other, so that data structures that use hash tables (like HashSet and HashMap) work properly.
  - The abstraction function is the basis for equality in immutable data types.
  - Reference equality is the basis for equality in mutable data types; this is the only way to ensure consistency over time and avoid breaking rep invariants of hash tables.

Summary

- **Safe from bugs**
  - Correct implementation of equality and hash codes is necessary for use with collection data types like sets and maps. It's also highly desirable for writing tests. Since every object in Java inherits the Object implementations, immutable types must override them.
- **Easy to understand**
  - Clients and other programmers who read our specs will expect our types to implement an appropriate equality operation, and will be surprised and confused if we do not.
- **Ready for change**
  - Correctly-implemented equality for immutable types separates equality of reference from equality of abstract value, hiding from clients our decisions about whether values are shared. Choosing behavioral rather than observational equality for mutable types helps avoid unexpected aliasing bugs.

The end

March 20, 2018