

8 Equality in ADT and OOP ADT和OOP中的"等价性"

Wang Zhongjie

rainy@hit.edu.cn

June 7, 2021

Objective of this lecture

- Understand the properties of an equivalence relation. 等价关系
- Understand equality for immutable types defined in terms of the abstraction function and observations. 站在观察者角度,利用AF,定义不可变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

- Equivalence Relation
- Equality of Immutable Types
- == 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





1 Equivalence Relation

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, $\sqrt{9}$, and 3 are alternative expressions for the same ideal mathematical value.

Equivalence Relation (关系等价性)

- An equivalence is a relation E □ T x T that is:
 - reflexive: E(t,t) ∀ t∈T 自反
 - symmetric: E(t,u) ⇒ E(u,t) 对称
 - transitive: E(t,u) ∧ E(u,v) ⇒ E(t,v) 传递
 - 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
- For a boolean-valued binary operation like == or equals(), the equivalence E is the set of pairs (x,y) for which the operation returns true.
- So for ==, these properties can also be written as:
 - Reflexive: t==t, ∀ t∈T
 - Symmetric: t==u ⇒ u==t
 - Transitive: t==u ∧ u==v ⇒ t==v



2 Equality of Immutable Types

Using AF to define the quality (利用AF定义等价性)

- Using an abstraction function (AF).
 - 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

- An equivalence relation induces an abstraction function (the relation partitions T, so f maps each element to its partition class). 用等价关系引入抽象函数。
- The relation induced by an abstraction function is an equivalence relation. 用抽象涵数定义的关系是等价关系。

Using observation to define the equality (用观察方法定义等价性)

- Another 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.
 - 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 \in {1,2} is true, and 1 \in {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

站在外部观察者角度:对两个对象调用<mark>任何相同的操作</mark>,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!

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 1: Duration

Here's a simple example of an immutable ADT.

```
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;
    }
}
```

Now which of the following values should be considered equal?

```
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 abstractionfunction definition of equality, and the observational equality definition.

Example 2: LetterSet

Since there're no any operations except the constructor, we cannot use the observational equality.

Using the abstraction-function definition of equality for LetterSet:

- new LetterSet("abc")
- new LetterSet("aBc")
- new LetterSet("")
- new LetterSet("1a2b3c")

```
/** Immutable type representing a subset of the letters a-z, ignoring case */
class LetterSet {
    private String s;
   // Abstraction function:
   // AF(s) = the subset of the letters {a..z} that are found in s
                (ignoring alphabetic case and non-letters)
    // Rep invariant:
/** @return the size of this set */
                                     new LetterSet("a") == new LetterSet("b")
public int size() { ... }
/** @param letter must be a letter 'a'...'z' or 'A'...'Z'
 * @return true iff this set contains letter, ignoring alphabetic case */
public boolean contains(char letter) { ... }
/** @return the length of the string that this set was constructed from */
public int length() { ... } new LetterSet("a") <> new LetterSet("aa")
/** @return the union of this and that */
public LetterSet union(LetterSet that) { ... }
/** @return true if and only if all the letters in this set are lowercase */
public boolean isAllLowercase() { ... }
/** @return the first letter in s */ 🗙
public char first() { ...
```

Example 3: Myline

```
/** Immutable type representing lines in the plane that are neither horizontal nor vertical.
                              Using the observational definition of equality:
public class MyLine {
   private double a;
                                new MyLine(new int[] { 0,0, 1,1 })
   private double b;
                              new MyLine(new int[] { 0,0, -1,-1 })
                              new MyLine(new int[] { 5,10, 6,11 })
   // Abstraction function:
                              new MyLine(new int[] { 0,0, 1,5 })
        TODO
   // Rep invariant:
                              new MyLine(new int[] { 0,0, 1,1, 2,2 })
        TODO
    * Make a MyLine that passes through all the points (p[2i], p[2i+1]).
    * p must contain at least 2 points and all points in p must be collinear
    * on a line that is neither horizontal nor vertical.
    * For example, MyLine({ 0,0, 1,2 }) passes through both (0,0) and (1,2).
   public MyLine(int[] p) { ... }
   /** Get the slope of the line. */
   public double slope() { ... }
```



object

equality

equals()

isEqual:

Equals()

==

n/a

referential

equality

==

is

==

Java

C#

Objective C

== vs. equals()

- Java has two different operations for testin 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.

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!!!
```

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)



4 Implementing equals()

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:

这不是override, 而是overload!

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

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. Static Type Checking

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.

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

A better 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();
}
```

- The first method overrides and replaces the equals(Object) method inherited from Object.
- It tests the type of the that object passed to it to make sure it's a Duration, and then calls a private helper method sameValue() to test equality.

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

```
public final class PhoneNumber {
  private final short areaCode;
  private final short prefix;
  private final short lineNumber;
                                     进行类型比较
                                     和null值判定
  @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
                                            严格来说,在没有
            && pn.areaCode == areaCode;
                                            AF的情况下直接在
                                            equals()中判断每
                                            个域的等价性,是
                                                不正确的
```

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.

instanceof

Do not do this.
This code is
bad.

```
public void doSomething(Account acct) {
  long adj = 0;
  if (acct instanceof CheckingAccount) {
      checkingAcct = (CheckingAccount) acct;
      adj = checkingAcct.getFee();
  } else if (acct instanceof SavingsAccount) {
      savingsAcct = (SavingsAccount) acct;
      adj = savingsAcct.getInterest();
       Never(?) use instanceof in a superclass to check type
                        against subclass
```

Use polymorphism多态性 to avoid instanceof

```
public interface Account {
   public long getMonthlyAdjustment();
public class CheckingAccount implements Account {
   public long getMonthlyAdjustment() {
      return getFee();
public class SavingsAccount implements Account {
   public long getMonthlyAdjustment() {
      return getInterest();
             public void doSomething(Account acct) {
                long adj = acct.getMontlyAdjustment();
```



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; 等价关系: 自反、传递、对称
 - 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; 除非对象被修改了,否则调用多次equals应同样的结果
 - 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. "相等"的对象,其hashCode()的 结果必须一致

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 这些因素结合在一起确保equals是所有对象 的全局等价关系
- 用"是否为等价关系"检验你的equals()是否正确

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. 行为不稳定和不可预测
 - 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.

```
@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 within a clock-skew tolerance private boolean sameValue(Duration that) {
    return Math.abs(this.getLength() - that.getLength()) <= CLOCK_SKEW;
}</pre>
```

Which property of the equivalence relation is violated?

Breaking the Equivalence Relation

```
@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 within a clock-skew tolerance
private boolean sameValue(Duration that) {
   return Math.abs(this.getLength() - that.getLength()) <= CLOCK SKEW;
              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);
                       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. 哈希表是映射的一种表示形式:将键映射到值的抽象数据类型。
 - 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.

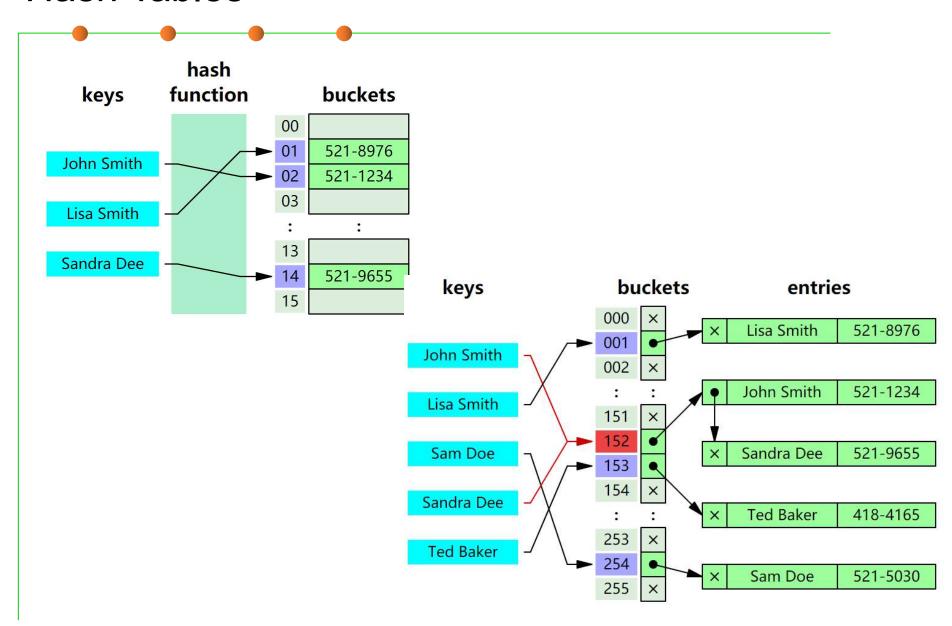
How a hash table works:

- 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.哈希表的rep不变量包含一个基本约束,即键位于由其哈希代码确定的槽中。

Breaking Hash Tables

- Hashcodes are designed so that the keys will be spread evenly over the indices. hashcode的设计使得密钥将均匀地分布在索引上。
- But occasionally a conflict occurs, and two keys are placed at the same index. 但有时会发生冲突,两个键放在同一索引中。
- 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* . 因此,哈希表不是在索引中保存单个值,而是实际保存一个键/值对列表,通常称为哈希桶 .
- A key/value pair is implemented in Java simply as an object with two fields. 一个键/值对在Java中实现为一个包含两个字段的对象。
- On insertion, you add a pair to the list in the array slot determined by the hash code. 插入时,将一对添加到由哈希代码确定的数组槽中的列表中。
- 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



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.从一个应用程序的一个执行到同一应用程序的另一个执 行,这个整数不需要保持一致。
- 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. 等价的对象必须有相同的hashCode
 - 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. 但是,程序员应该知 道,为不相等的对象生成不同的整数结果可能会提高哈希表的性能,不相等的对象, 也可以映射为同样的hashCode,但性能会变差

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

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. 如果两个相等的对象具有不同的哈希码,则它们可能被放置在不同的插槽中。
 - So if you attempt to lookup a value using a key equal to the one with which it was inserted, the lookup may fail.因此,如果尝试使用与插入值相同的键来查找值,则查找可能会失败。
- 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()

- 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. 一个简单而有效的方法来确保契约得到满足,就是hashCode总是返回一些常量值,所以每个对象的 hash码都是相同的。
 - 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.这满足了对象契约,但会对性能产生灾难性的影响,因为每个键都将存储在同一个槽中,并且每个查找都将退化为沿长列表的线性搜索。
 - 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. 标准是为用于确定相等性的对象的每个组件计算一个哈希码(通常通过调用每个组件的hashCode方法),然后将它们组合起来,抛出一些算术运算。
- For Duration, this is easy, because the abstract value of the class is already an integer value:

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

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. Java的最新版本现在有了一个实用方法
 Objects.hash(),使得实现包含多个字段的哈希代码变得更容易。
- 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()

除非你能保证你的ADT不会被放入到Hash类型的集合类中 ②

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() {
        short[] hashArray = {areaCode, prefix, lineNumber};
        return Arrays.hashCode(hashArray);
    }
    ...
}
```

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();
```



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完全相同的, 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> 1 : set) {

set.contains(1) → 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 哈希相/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...

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. 如果某个mutable的对象包含在Set 集合类中,当其发生改变后,集合类的行为不确定 ② 务必小心
- 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. 在JDK中,不同的mutable类使用不同的等价性标准… ❸

Two examples

Date类的equals()的spec: "Two Date objects are equal if and only if the getTime method returns the same long value for both."

观察等价性

- List类的equals()的spec: "Returns true if and only if the specified object is also a list, both lists have the same size, and all corresponding pairs of elements in the two lists are equal. "
 - Two lists are defined to be equal if they contain the same elements in the same order.

 观察等价性
- StringBuilder类的equals继承自Object类 ②

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

行为等价性

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 implemented
                           If Bag is implemented
                             with behavioral
                                                 observational equality
                                                 despite the dangers
                                equality
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)
```

clone() in Object

- clone() creates and returns a copy of this object.
- The precise meaning of "copy" may depend on the class of the object.
- The general intent is that, for any object x:

```
x.clone() != x
x.clone().getClass() == x.getClass()
x.clone().equals(x)
```

从这些contracts中无法确保是deep copy!



- 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?
```

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 equality判 断 Integer对象的 equals()是object equality

```
a.get("c").equals(b.get("c"));
```

What about this code?

```
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("a")== b.get("a"));
```

Numbers between -128 and 127 are true.



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

June 7, 2021