哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

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

3.5 Equality in ADT and OOP

Xu Hanchuan
xhc@hit.edu.cn

April 3, 2019

Outline

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

Objective of this lecture

- **Understand equality defined in terms of the abstraction function, an equivalence relation, and observations. 等价性的三种理解**
- **Differentiate between reference equality and object equality 引用等价和对象等价的区别**
- **Differentiate between strict observational and behavioral equality for mutable types. 可变类型的观察性等价和行为性等价的区别**
- **Understand the Object contract and be able to implement equality correctly for mutable and immutable types. 理解Object类的契约，能够正确实现可变类型和不可变类型的等价**

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.
- For an abstract data type, the ***abstraction function*** 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.
  AF决定了ADT中各操作的实现
- **The abstraction function (AF) gives a way to cleanly define the equality operation on an ADT. 可通过AF判定ADT中操作的等价**

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.现实世界中任何对象都是独一无二的
- So two physical objects are never truly "equal" to each other; they only have degrees of similarity. 现实世界中任意两个对象是不相等的
- In the world of human language, however, and in the world of mathematical concepts, you can have multiple names for the same thing. 在数学概念中，可以用多个名字指代相同的事物
- 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. 什么情况下两个概念是相同的？

2 Three ways to regard equality

Using AF or using a relation

- **Using an abstraction function** . Recall that an abstraction function `AF: R → A` maps concrete instances of a data type to their corresponding abstract values. To use `AF` as a definition for equality, we would say that `a` equals `b` if and only if `AF(a)=AF(b).`
- **Using a relation** . An *equivalence* is a relation `E ⊆ T x T` that is:
  - Reflexive自反的: `E(t,t) ∀t∈T`
  - symmetric对称的: `E(t,u) ⇒ E(u,t)`
  - transitive传递的: `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)`.
- These two notions are equivalent.
  - An equivalence relation induces an abstraction function (the relation partitions T, so AF 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. 从观察的角度，对两个对象的任何同一操作都会得到相同的结果**
  - Consider the set expressions `{1,2}` and `{2,1}`. Using the observer operations available for sets, cardinality基数 `|...|` 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.

```
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 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.
  - **The equals() operation compares object contents – in other words, object equality.**

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

- **The equals operation has to be defined appropriately for every abstract data type.**
- **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. 定义新数据类型时，需要考虑等价的含义，然后实现equals()方法。**

The == operator vs. equals method

- **For primitive data type you *must* use ==**
- **For object reference types**
  - The == operator provides *identity semantics*
  - In java, == just tests reference identity, it doesn't compare object contents.
  - 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

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

- Using == on an object reference is a **bad smell in code**

```
StringBuilder a = new StringBuilder("yes");
StringBuilder b = new StringBuilder("yes");
if (a==b) {  // a bug!!
    System.out.println("a and b are the same");
}
```

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")));
   }
}
```

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

true or fasle?

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")));
   }
}
```

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

true or fasle?

Tips for overriding a method

- **If you want to override a method:**
  - Make sure signatures match
  - Use `@Override` so compiler has your back **让compiler 协助检查覆盖方法和被覆盖的方法签名是否完全一致**
  - *Do* copy-and-paste declarations (or let IDE do it for you)

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.**
- **For immutable data types, this is almost always wrong.**
- **We have to override the `equals()` method, replacing it with our own implementation.**

`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();
    }
}
```

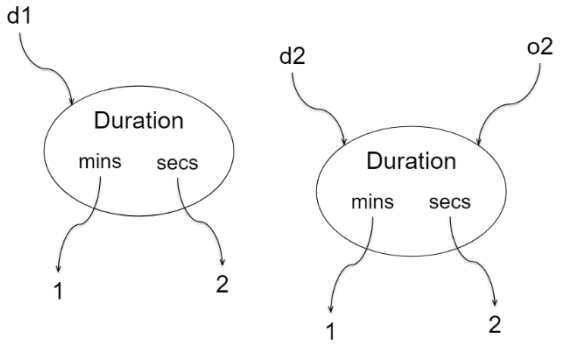
- **How about this code?**

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

???

- **Why?**

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



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.**

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. 即使o2和d2在运行时指向了统一对象，但结果不同，出现了等价的不一致问题**

```
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();
}
```

`equals` Override Example

```
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))
         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.

除了用于实现equals()方法，尽可能避免使用 instanceof和getClass()在运行时检测对象的类型，存在安全隐患，可以使用更安全的替代方法！

5 The `Object` contract

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

- **When you override the equals method of Object, you must adhere to its general contract: 重写Object类中equals()时，需要满足约定：**
  - `equals` must define an equivalence relation – that is, a relation that is reflexive, symmetric, and transitive; 满足等价性(自反，对称，传递)
  - `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，在比较中用到的信息没有被修改的情况下，多次比较结果应始终相同
  - 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. 使用**equals**方法判定相等的两个对象，其**hashCode**必须产生相同的结果。

The `equals` contract

- The equals method implements an **equivalence relation**:
  - **Reflexive**: For any non-null reference value x, x.equals(x) must return true. 自反性
  - **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. 对称性
  - **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. 传递性
  - **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. 一致性
  - For any non-null reference value x, x.equals(null) must return false. 空值处理
- **`equals` is a global equivalence relation over all objects.** **equals是所有对象的全局等价关系(对所有对象都生效)**

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;
}
```

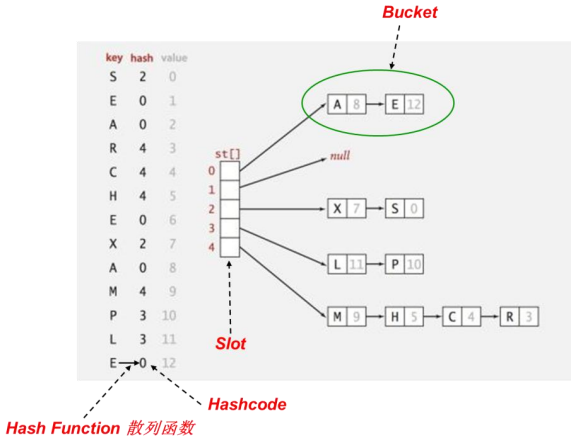
Which property of the equivalence relation is violated?

Breaking Hash Tables

- **A hash table is a representation for a mapping: an abstract data type that maps keys to values. 哈希表实现了键-值之间的映射**
  - 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. **哈希表的查询时间为常数(线性)，性能优于trees或lists，且不需要排好序或其他特殊要求，除了需要提供equals和hashCode方法。**

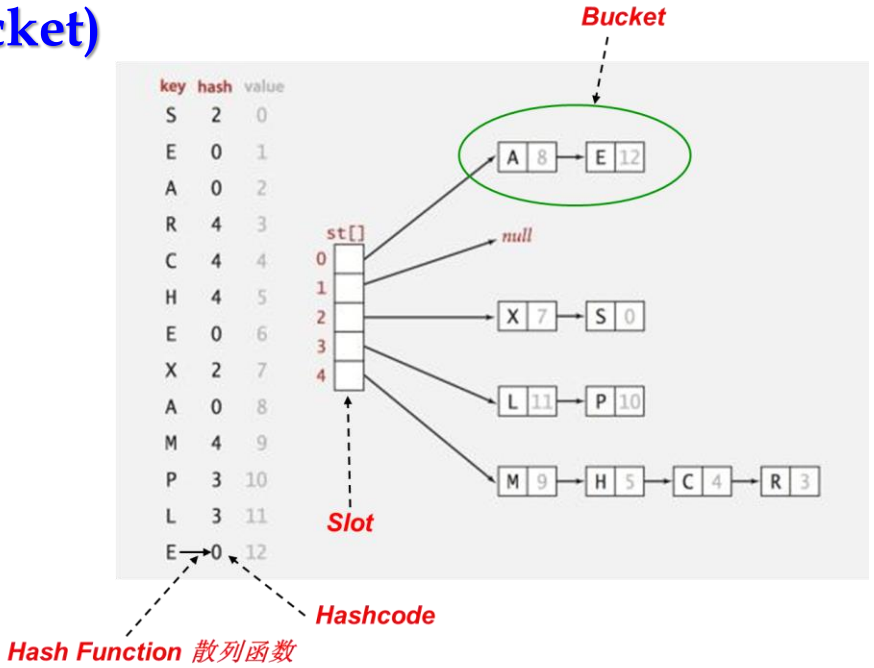
Breaking Hash Tables

- **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. 包含一个数组
  - 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. 键值对中的key被映射为hashcode，对应到数组的index， hashcode决定了数据被存储到数组的那个位置
- **The rep invariant of a hash table includes the fundamental constraint that keys are in the slots determined by their hash codes.**



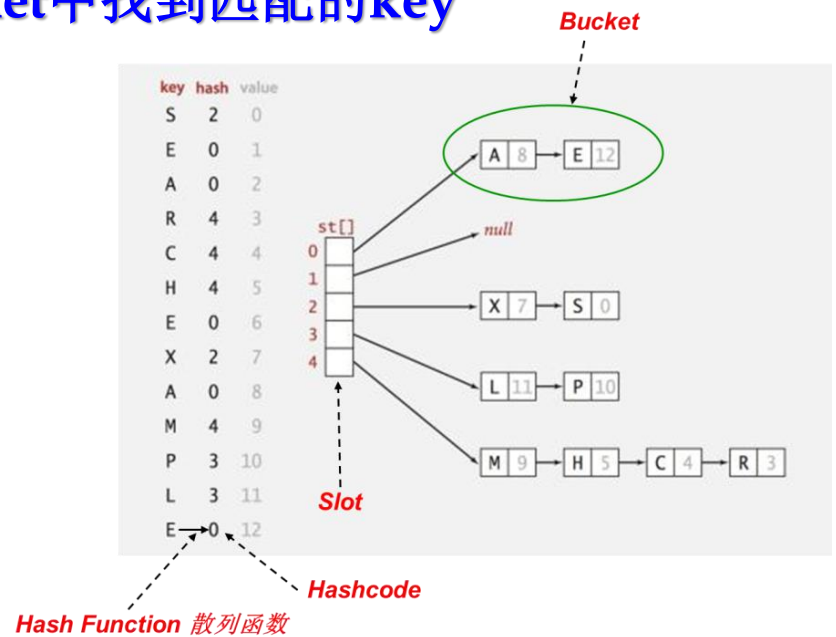
Breaking Hash Tables

- Hashcodes are designed so that the keys will be spread evenly over the indices. **hashcode设计时会尽量确保均匀分布到数组**
- 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***. **当多个key散列到同一个index时(冲突)，哈希表维护一个列表来记录这些键值对(bucket)**

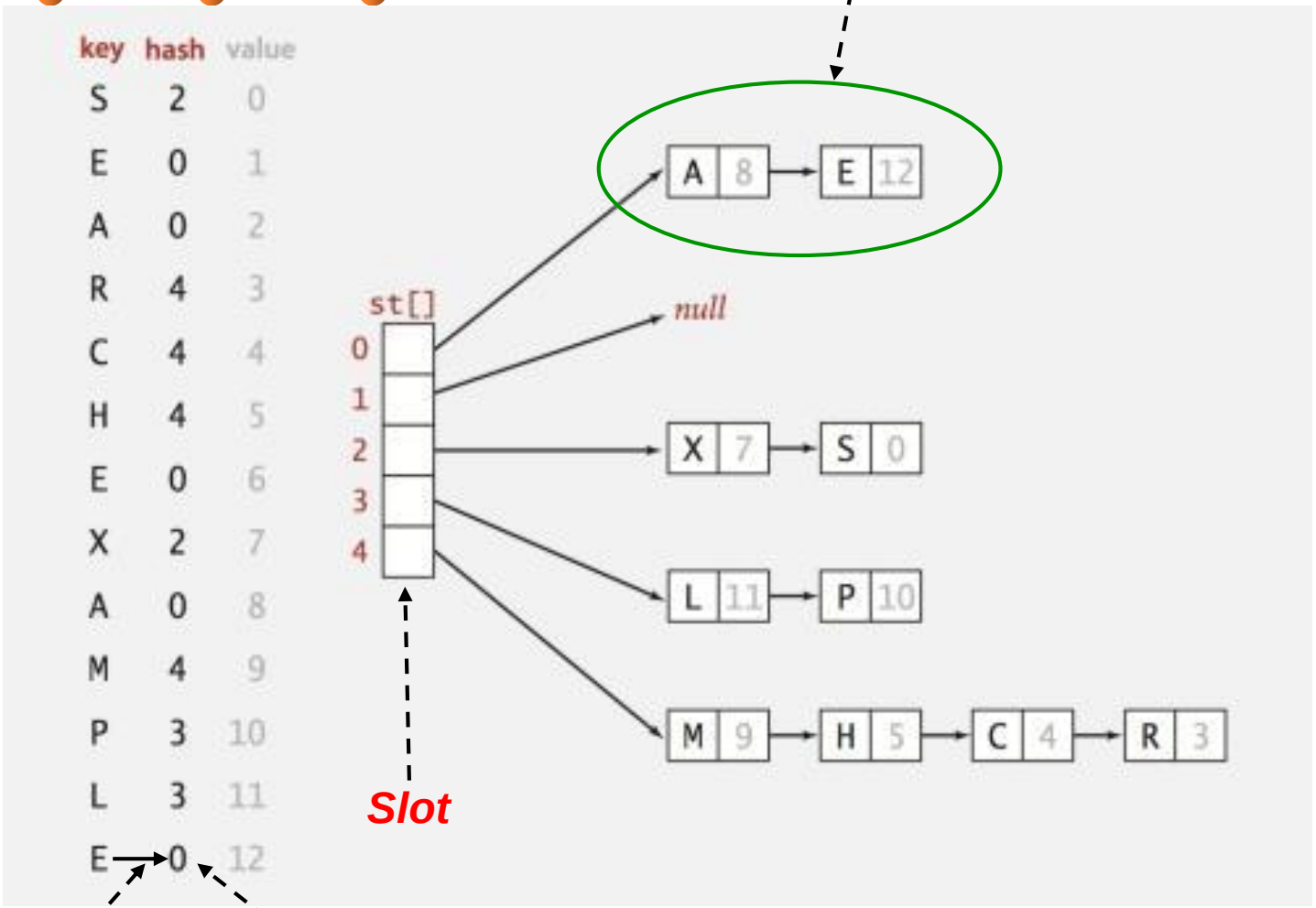


Breaking Hash Tables

- 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. **首先利用hashCode()产生的hashcode确定slot(index)**
- 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. **再用equals()方法在bucket中找到匹配的key**



Breaking 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.** 只要比较操作用到的信息没有被修改，那么对这同一个对象调用多次，**hashCode()**方法必须始终返回相同整数。
  - 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
- **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.** 如果**equals**比较相等，则要求**hashcode**相等；如果**equals**不等，则**hashcode**是否相等均可，但最好不等**(**提升性能**)**
  - 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

  两个对象equals操作相等，则hashcode必须相等。因此，重写equals时，必须重写hashCode。
- Unequal objects **should** have different hash codes **能够提升性能(避免了调用不必要的equals操作)**
  - Take all value fields into account when constructing it **构造hashcode时考虑对象的所有字段，以避免不相等对象产生相同hashcode**
- Hash code must not change unless object mutated. **除非对象可变，否则hashcode不能修改**

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. 最简单方法：让所有对象的hashCode为同一常量，符合contract，但降低了hashTable效率**
  - 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. 通过equals计算中用到的所有字段的hashCode组合出新的hashCode**
- **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 */; }
}
```

大部分语言中都是采用对象的内存地址作为默认**hashcode**

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. 此方法可以根据多个字段生成hashcode**

- **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. So as a general rule:**

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

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")));
    }
}
```

(a) `true`
(b) `false`
(c) It varies
(d) None of the above

- `Name` overrides `hashCode` but not `equals`! The two `Name` instances are thus unequal.

How do you fix it?

- Replace the overloaded equals method with an overriding equals method.

```
@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);
}
```

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);
   }
   ...
}
```

Recommended reading materials

- **Item 10: Obey the general contract when overriding equals**
- **Item 11: Always override hashCode when you override equals**

In 《Effective Java》 Third Edition

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:**
  - **observational equality** means that two references cannot be distinguished by code that doesn't change the state of either object, i.e., by calling only observer, producer, and creator methods. This tests whether the two references "**look**" the same in the current state of the program. 观察等价：在不改变对象状态的情况下(不使用**mutator**)，无法区分对象
  - **behavioral equality** means that two references cannot be distinguished by *any code*, even if a mutator is called on one but not the other. This tests whether the two references will "**behave**" the same, in this and all future states. 行为等价：改变一个对象而不改变另外一个时，仍然无法区分对象
- **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 use observational equality. 对于可变对象，使用观察等价性貌似可行，但是会带来潜在的问题。**
- **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.

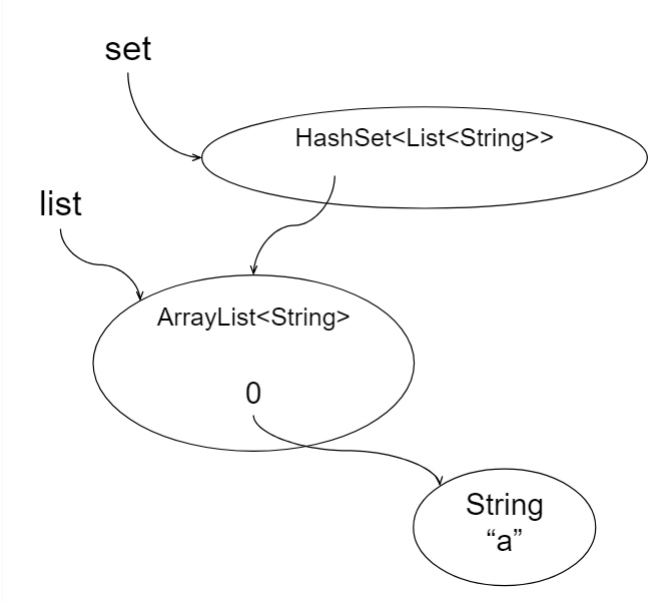
An example

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

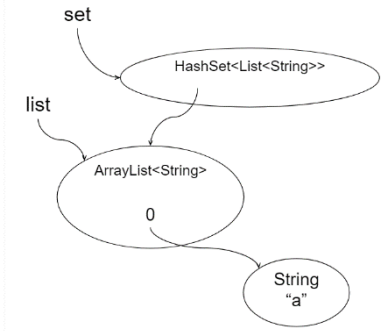
```
List<String> list = new ArrayList<>();
list.add("a");

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
```

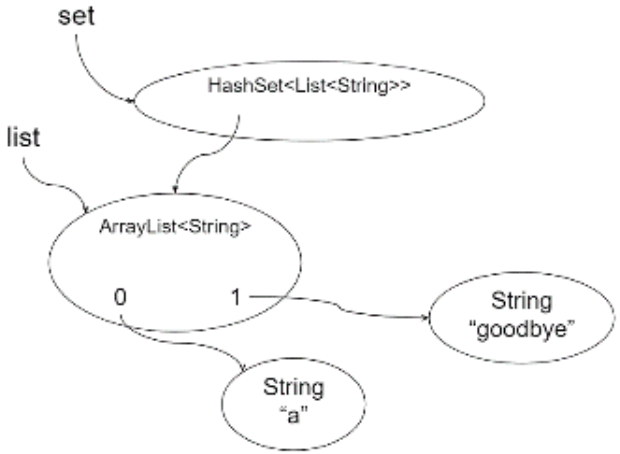


An example



- **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 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.**
  - Note: 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. 格外关注：可变对象作为集合元素的时候，对象值的变化会对相等比较产生影响。

Lessons learned from this example

- `equals()` should implement behavioral equality. Equals()应该实现行为等价
- In general, that means that two references should be `equals()` if and only if they are aliases for the same object. 两个引用相等意味着它们指向了同一个对象
- So mutable objects should just inherit `equals()` and `hashCode()` from `Object`.可变类型应该继承Object类的equals()和hashCode()
- 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()`. 客户端如果需要判断两个对象的观察等价性，可以重新定义一个方法

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

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()`. **不可变类型应该重写equals()和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. **可变类型不应该重写equals()和hashCode()，采用Object默认实现的即可。**

8 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= Integer.valueOf(300);
Integer y= Integer.valueOf(300);
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

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

- But what if x==y? ----- **True**
- **To improve performance, Java caches Integer objects from -127 to 127.**

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.

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

The end

April 3, 2019