1001 users solved this topic. Latest completion was about 1 hour ago.

Theory: hashCode() and equals()

22 minutes 0 / 4 problems solved

Skip this topic Start practicing

§1. The behavior of hashCode() and equals() methods

Sometimes, you need to compare objects of your custom class with each other. The `java.lang.Object` class, which is the superclass of any class, provides two methods for that: `equals(Object obj)` and `hashCode()`. Their default behavior is the following:

- `boolean equals(Object obj)` checks whether this object and another one are stored in the same memory address;
- `int hashCode()` returns an integer hash code that is unique for each object (object's identity).

Let's look at how they behave. Here's a simple `Person` class with three fields:

```
class Person {

    private String firstName;
    private String lastName;
    private int age;

    // constructor, getters and setters
}
```

There are two objects that basically represent the same person (i.e., the objects are logically equivalent):

```
Person p1 = new Person("John", "Smith", 31);
Person p2 = new Person("John", "Smith", 31);
```

However, the `equals` method considers them to be different since it compares references rather than the values of their fields:

```
System.out.println(p1.equals(p2)); // false
```

The `hashCode` method also says nothing about their equality:

```
System.out.println(p1.hashCode()); // 242131142
System.out.println(p2.hashCode()); // 1782113663
```

Note, you may see other values than 242131142 and 1782113663!

So, the default behavior of methods `equals(Object obj)` and `hashCode()` is not enough to compare objects of a custom class by the values of their fields.

What's interesting is how these methods behave with standard classes, for example, `String` :

```
String person1 = new String("John Smith");
String person2 = new String("John Smith");

System.out.println(person1.equals(person2)); // true

System.out.println(person1.hashCode()); // 2314539
System.out.println(person2.hashCode()); // 2314539
```

If we want to define a similar logic for equality testing in the `Person` class, we should override both of the described methods. It is not enough to just override just one of them.

§2. Overriding equals()

Current topic:

hashCode() and equals()

Topic depends on:

✗ Hiding and overriding

✗ The Object class

✗ Runtime type checking

✓ Hash table

Table of contents:

To test the **logical equality** of objects, we should override the `equals` method of our class. It is not as trivial as it may sound.

There are some math restrictions placed on the behavior of `equals`, which are listed in the documentation for `Object`.

- **Reflexivity:** for any non-null reference value `x`, `x.equals(x)` should return `true`.
- **Symmetry:** for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- **Transitivity:** for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
- **Consistency:** for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided that no information used in `equals` comparisons on the objects is modified.
- **Non-nullity:** for any non-null reference value `x`, `x.equals(null)` should return `false`.

To create a method that satisfies the listed restrictions, first, you need to select the field that you want to compare. Then you should perform three tests inside the `equals` method:

1. if this and other object have the same reference, **the objects are equal**, otherwise — go to step 2;
2. if the other object is `null` or has an unsuitable type, **the objects are not equal**, otherwise — go to step 3;
3. if all selected fields are equal, **the objects are equal**, otherwise, they are **not equal**.

If you do not perform all of these tests, in some cases, the `equals` method will not work properly.

Here is a modified class `Person` that overrides the `equals` method. It uses all three fields in comparison.

```java
class Person {

    private String firstName;
    private String lastName;
    private int age;

    // constructor, getters and setters

    @Override
    public boolean equals(Object other) {
        /* Check this and other refer to the same object */
        if (this == other) {
            return true;
        }

        /* Check other is Person and not null */
        if (!(other instanceof Person)) {
            return false;
        }

        Person person = (Person) other;

        /* Compare all required fields */
        return age == person.age &&
                Objects.equals(firstName, person.firstName) &&
                Objects.equals(lastName, person.lastName);
    }
}
```

In the example above, we use `java.util.Objects.equals(obj1, obj2)` to check if the string fields are equal. This approach allows us to avoid a `NullPointerException`.

Below is an example where we test three objects for equality. Two of the objects represent the same person.

```java
Person p1 = new Person("John", "Smith", 31); // a person
Person p2 = new Person("John", "Smith", 31); // the same person
Person p3 = new Person("Marry", "Smith", 30); // another person

System.out.println(p1.equals(p2)); // true
System.out.println(p2.equals(p3)); // false
System.out.println(p3.equals(p3)); // true (reflexivity)
```

As you can see, now the `equals` method compares two objects and returns `true` if their fields are equal, otherwise — `false`.

§3. Overriding hashCode()

If you override `equals`, a good practice is to override `hashCode()` as well. Otherwise, your class cannot be used correctly in any collection that applies a hashing mechanism (such as `HashMap`, `HashSet` or `HashTable`).

Below are three requirements for the `hashCode()` method (taken from the documentation).

1) Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer doesn't have to remain the same from one execution of an application to another.

```
person1.hashCode(); // 400000 - ok
person1.hashCode(); // 400000 - ok
person1.hashCode(); // 500000 - not ok
```

2) 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.

```
person1.equals(person2); // true

person1.hashCode() == person2.hashCode(); // false - not ok, it must be true
```

3) It is not required for unequal objects to produce distinct hash codes. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

```
person1.equals(person3); // false
person1.hashCode() == person3.hashCode(); // true - will work
```

The simplest implementation of the `hashCode()` method may look as follows:

```
@Override
public int hashCode() {
    return 42;
}
```

It always returns the same value and satisfies both required conditions 1 and 2, but does not satisfy the optional condition 3. Unfortunately, this method is very inefficient for industrial programming since it totally degrades the power of hash-based collections. A good hash function tends to generate different hash codes for unequal objects.

To develop a valid and effective `hashCode` method, we recommend the algorithm proposed by Joshua Bloch in his book "**Effective Java**".

1. Create a `int result` and assign a **non-zero** value (i.e. `17`).
2. For *every field* `f` tested in the `equals()` method, calculate a hash code `code`:
    1. Calculate the integer hash code for `f` :
        - If the field `f` is a `boolean`: calculate `(f ? 0 : 1)`;
        - If the field `f` is a `byte`, `char`, `short` or `int`: calculate `(int) f`;
        - If the field `f` is a `long`: calculate `(int)(f ^ (f >>> 32))`;
        - If the field `f` is a `float`: calculate `Float.floatToIntBits(f)`;
        - If the field `f` is a `double`: calculate `Double.doubleToLongBits(f)` and handle the return value like every long value;
        - If the field `f` is an *object*: use the result of the `hashCode()` method or 0 if `f == null`;
        - If the field `f` is an *array*: see every field as a separate element and calculate the hash value in a *recursive fashion* and combine the values as described next.
    2. Combine the hash value `code` with `result` as follows: `result = 31 * result + code;`.
3. Return `result` as a hash code of the object.

It is important, do **NOT include** fields that are not used in `equals` to **this algorithm**.

Here we apply the described algorithm to the `Person` class.

```
class Person {

    private String firstName;
    private String lastName;
    private int age;

    // constructor, getters and setters

    // overridden equals method

    @Override
    public int hashCode() {
        int result = 17;
result = 31 * result + (firstName == null ? 0 : firstName.hashCode());
result = 31 * result + (lastName == null ? 0 : lastName.hashCode());
        result = 31 * result + age;
        return result;
    }
}
```

Below you can see an example of invoking `hashCode()` for three objects. Two of the objects represent the same person.

```
Person p1 = new Person("John", "Smith", 31);  // a person
Person p2 = new Person("John", "Smith", 31);  // the same person
Person p3 = new Person("Marry", "Smith", 30); // another person

System.out.println(p1.hashCode()); // 409937238
System.out.println(p2.hashCode()); // 409937238
System.out.println(p3.hashCode()); // 689793455
```

As you can see, we have the same hash code for equal objects.

Note, since Java 7, we have an `java.util.Objects.hash(Object... values)` utility method for hashing `Objects.hash(firstName, secondName, age)`. It hides all magic constants and null-checks inside.

§4. Summary

The default behavior of the `equals` method provided by the `java.lang.Object` class checks whether objects references are equal. This is not enough if you would like to compare objects by the values of their fields. In this case, you should override the `equals` method in your class.

The correct implementation should satisfy the following conditions: **reflexivity**, **symmetry, transitivity, consistency,** and **non-nullity**. You should also override the `hashCode` method, taking into account that:

- if two objects are equal, they MUST also have the same hash code;
- if two objects have the same hash code, they do NOT have to be equal too.

While it is good to understand `hashCode()` and `equals()` methods, we do not recommend to implement them manually in industrial programming. Modern IDEs such as IntelliJ IDEA or Eclipse can generate correct implementations for both these methods automatically. This approach will help you to avoid bugs since overriding these methods is quite error-prone.

If you'd like to know more, read the book "Effective Java" by Joshua Bloch. You can also read the IBM article ["Hashing it out"] as an addition to this topic.

Report a typo

83 users liked this theory. **2** didn't like it. **What about you?**

😍 🙂 😐 🙁 😡

Start practicing

Comments (12) Hints (0) Useful links (1) Show discussion