L35

Wednesday, February 2, 2022 12:00 AM

Overiding toString() & equals() methods, scope

Overiding toString() & equals() methods

Q1. The toString present in the Object class ensures that an object of any class in Java can be converted into a String representation.

If we do not override the toString method, by default the toString method present in the Object's class will be called (which is of no much use).

See and retype the below code and observe the output to understand the usage of toString method.

You will also notice that when an object reference is concatenated with + operator to a string literal, the object reference is automatically converted to a string by invoking the toString method.

That is how in the statement System.out.println("a : " + a);, "a : " + a is resulting in "a : " + a.toString().

```
package q11279;
public class ToStringExample {
    public static void main(String[] args) {
        A a = new A(4);
        System.out.println("a.toString() : " + a.toString());
        System.out.println("a : " + a);
    }
}
class A {
    private int value;
    public A(int value) {
        this.value = value;
    }
    public String toString() {
        return "The value is : " + value;
    }
}
```

Q2. If we do not override the equals method, by default the equals method present in the Object's class will be called.

The implementation of equals present in the root class Object, only verifies if the references are same. It does not verify if the contents are same. Cont ent verification is the responsibility of the overriding implementation.

See and retype the below code and observe the output to understand the usage of equals() method in case of A and B classes.

After you see the output you will realize that **since we did not override** the equals method in class B, the equals method invocations on class B instances, b1.equals(b2) and b1.equals(b3) both return false. This is because b1, b2 and b3 refer to three different objects in memory and the default equals method implementation in Object class will treat them different as they have different addresses.

However the calls a1.equals(a2) and a1.equals(a3) correctly return false and true respectively.

Let us understand the overridden equals method implementation in class A.

1. The below condition will ensure that the method will return true, if both references are actually pointing to same object (address in memory).
   ```
   if (this == otherObject) { // checking if both are pointing to same reference (address)return true;
   }
   ```
2. The below code ensures that we are comparing two references of the same type. Since the equals method accepts a parameter of type Object, compiler will allow us to invoke equals method by passing objects of some other type, (since every class instance is a subtype of Object class). So the below condition will ensure that we are comparing apples to apples and oranges to oranges. If not it will return false.
   ```
   if (otherObject instanceof A) {
           A otherARef = (A) otherObject;return this.value == otherARef.value} else {
           return false;
           // the above line returns false if the object held in reference otherObject is not an instance of type A
   ```
3. If the reference is of type A, then we convert (by type casting) the reference otherObject into a reference of type A. And later we return the value returned by the comparision expression (this.value == otherARef.value), which actually checks the contents present in the field value in both the objects.
   ```
   if (otherObject instanceof A) {A otherARef = (A) otherObject;
           // the above line type casts otherObject which is of type Object to a reference otherARef of type Areturn (this.value == otherARef.value);
           // the above line compares the content of field value in this (current) object to the // content in the value field of otherARef and returns // whatever boolean value the comparison expression evaluates to.} else {
           return false;
   }
   ```

```java
package q11280;
public class EqualsExample {
    public static void main(String[] args) {
        A a1 = new A(4);
        A a2 = new A(5);
        A a3 = new A(4);
        System.out.println("a1.equals(a2) : " + a1.equals(a2));
        System.out.println("a1.equals(a3) : " + a1.equals(a3));
        B b1 = new B(4);
        B b2 = new B(5);
        B b3 = new B(4);
        System.out.println("b1.equals(b2) : " + b1.equals(b2));
        System.out.println("b1.equals(b3) : " + b1.equals(b3));
    }
}
class A {
    private int value;
    public A(int value) {
        this.value = value;
    }
    public boolean equals(Object otherObject) {
        if (this == otherObject) {
            return true;
        }
        if (otherObject instanceof A) {
            A otherARef = (A) otherObject;
            return (this.value == otherARef.value);
        } else {
            return false;
        }
    }
}
class B {
    private int value;
    public B(int value) {
        this.value = value;
    }
}

```

Scope

Q1. Scope can be defined as a portion or block of code in which a variable is visible.

We apply the word scope to variables/references and not so much to methods as methods are members of classes and interfaces.

Variables and references can be declared in:

1. Blocks - these can be if/else/switch, for/while loops, methods and constructors or named blocks. These are also called **local variables**. For example:
   public static void main(String[] args) {
   int x = Integer.parseInt(args[0]); *// x is visible only inside this main method*if (x > 1) { *//if-block-start*int y = 2 * x; *// y is visible only inside this if block*System.out.println("y = " + y);
   } *//if-block-end*}
2. Method and Constructor parameters. For example:
   public int sum(int num1, int num2) { *//method-block-start*return num1 + num2; *// num1 and num2 are only visible inside this method block*} *//method-block-end*
3. Class - as instance fields

class A { *//class-block-start*private String name; *//reference name is visible anywhere inside the class block* public String getName() {
return name; *// example usage 1*}
public String toString() {
return "A [name = " + name + "]"; *// example usage 2*}
} *//class-block-end*

In Java, before using a variable/reference we need to first declare it.

Select all the correct statements for the below code:

class A {
public A(int value1) { *//statement 1*this.value1 = value1; *//statement 2*value2 = value1 * 2; *//statement 3*}
private int getValue2() {
return value2; *//statement 4*}
private int value1; *//statement 5*}

- [ ] Statement 5 is in the wrong location, it should have been declared above statement 1.
- [ ] Statement 2 will give a compilation error, because there is no variable called `this` declared.
- [x] Statements 3 and 4 will produce compiler errors saying `value2` cannot be resolved, since `value2` is neither declared locally in the enclosing block nor as a field in the class.
- [ ] Statement 5 will produce a compilation error because `value1` is not initialized.