

Functions toString and equals in class Object

Function toString in class Object

Function `toString` in class `Object` is defined to return the name of the manilla folder, or object in which it appears. For example, evaluate this expression in DrJava's interactions pane:

```
new Object()
```

↓ and it will evaluate to something like `java.lang.Object@d70b42` —that's the name on the tab of the manilla folder or object, like the `a0` or `a1` that we have been writing from time to time. The name consists of the name of the class (including its package information), the `@` sign, and then 6 hexadecimal digits, whose meaning you don't have to know about at this point.

Object's function toString is inherited in every class but is usually overridden

Since `Object` is the root of the class hierarchy, its function `toString` is inherited by every class, so the function appears in *every* object.

↓ However, if you define function `toString` in a class, your definition overrides the inherited one. So, the `toString` function that we wrote in class `Point` in a previous web lecture overrode the `toString` function that was inherited from class `Object`. So, a call `p.toString()` calls the overriding function.

Later, in module 2, you will see that you may still be able to call the inherited function.

Function equals in class Object

↓ Now let's turn to a discussion of function `equals`, which is also defined in class `Object` —like this:

```
/** = "this object and object ob have the same name on their tabs
    (i.e. since object names are unique, they name the same object)". */
public boolean equals(Object ob) {
    return this == ob;
}
```

↓ We have shown the whole method in the object, to make sure that you understand that it really is there. You may not have seen a use of keyword `this` before. It refers to the name of the object in which is occurs, in this case, so it refers to `a0` itself. Therefore, a call on this method tests whether `ob` contains the name `a0`.

You can see how function `equals` works by evaluating this sequence in DrJava's interaction pane.

```
ob1= new Object();
ob1.equals(ob1) // This is true.
ob1.equals(new Object()) // This is false, because ob1 and the new object are different.
```

↓ Function `equals` is often overridden in newly defined classes, with the convention is that `equals` in a newly defined class should test for equality of all fields of the class. We will return to a discussion of overriding function `equals` in module 2. But for now, we concentrate on testing equality of strings.

Testing for equality of strings

↓ It is a big *mistake* to use `==` to test for equality of `Strings`, because a `String` value is an object, and not a value of a primitive type. For example the expression

```
"1" == "" + "1"
```

evaluates to **false**, even though both sides have the `String` "1" as their value, because `==` tests whether the names of objects, and not their contents, are equal. You can see this with the two objects whose names are in `s0` and `s1`.

The moral of the story is: *do not use == to test for equality of strings!*

↓ But class `String` does override function `equals`, with a function that tests for the equality of actual strings and not of object names. The following function calls all evaluate to true:

Functions toString and equals in class Object

```
"1".equals("1")  
"1".equals("" + "1")  
"1".equals("" + (2-1))
```

Using `==` with strings is a mistake that all Java programmers have made at least once. If you make the mistake, you will find that it's a tough bug to find—unless you are aware that it might happen and can look out for it. So be aware of the problem with using `==` for equality of strings.