Overriding methods

Changing what we have inherited from e.g. Object

We inherit e.g. toString(), equals() and hashCode()

But as we said before, the versions of these methods as defined in java.lang.Object are pretty simplistic.

Don’t worry. There is a solution to this “problem”!

Since we are happy to have a toString() method but feel the need to refine it a little to better serve our own class, we can simply override it.

Overriding a method is the way to change the behavior of one method, but preferably without changing the meaning of that method.

Overriding toString()

The version of toString() we inherited from Object simply returned a String with the class name and an @ and a hexadecimal number.

This is probably not a very good summary of instances of our own objects as a String. Let's say we have declared a class Member with simply two instance variables, name and email.

A better representation as a String of a Member instance would be the name and some delimiter and the email, e.g.:

Peter Jackson, peter@jurassic.org

How to override toString()

To override a method means declaring it like it was declared in the superclass but changing the actual code in the method body. The toString() implementation for the Member class would then look like this:

```
public String toString(){
  return name + ", " + email;
}
```

Because the version in `java.lang.Object` is defined with `public` access, `String` return type, the name `toString` and no parameters. All these part must match (with some exceptions we skip here) in order to override the method.

Now, we've overridden it. What's the use?

The usable thing about overriding it, is that code which deals with any type of object, safely can call toString() on any instance (via a reference). If the actual class (type) of the object has its own version of the method toString(), that code is the code which will be executed at runtime!

In fact, if we now did this:

```
Object o = new Member("Adam", "adam@email.com");
System.out.println(o);
```

The println method would eventually (in the end) call the version of toString() which we have overridden in the Member class. Even though the reference o is declared as of type Object.

Overriding allows for flexibility

Going back to our file manager example, overriding the thumbnail() method in the class File (if we pretend it has such a method) in the classes AudioFile and VideoFile and ImageFile would allow our application to loop through a list of File objects (viewed as being of type File) and call the thumbnail() method on each of them with a different result depending on the implementation in the overridden version in the different subclasses respectively.

Overriding equals - a few words

In short, there is a rule you should obey when overriding equals(), and it is to also override hashCode().

This is because two objects which are considered equals (the equals method on one object with the other as parameter returns true), should also return the same int number when hashCode() is called.

hashCode() should be a unique number (preferably) for an object. The reason for this rule, is that many methods expect this rule to be followed, and may use hashCode() for some reason, like when checking if an “equal” object exists in some colletion, like the key set of a Properties object.

There are excellent tutorials on hashCode etc

Use a search engine to find one!

A method can get a parameter as Object

One additional (potential) benefit of extending Object and overriding for instance toString() in a subclass, is that there might exist a method somewhere which we can call with any instance regardless of its class type.

Such a method could accept a parameter of type Object, and then make use of the fact that it knows it can call toString() on the parameter and at least get a String back (perhaps the boring string with @ but still a String).

In fact, println() in PrintStream has exactly this situation! Let’s look a little more on that....

What does println do? It calls String.valueOf

```
// in class PrintStream:
    public void println(Object x) { // can print any object!
        String s = String.valueOf(x);
        synchronized (this) {
            print(s);
            newLine();
        }
    }
// in class String:
    public static String valueOf(Object obj) {
        return (obj == null) ? "null" : obj.toString();
    }
```

Accessing the inherited version

You can use the keyword `super` to access stuff in the parent class.

Let's say we want to use the inherited version of toString() as part of our own toString():

```
public String toString(){
  return super.toString() + " with name: " +
         name + " and email: " + email;
}

// Member@2608fb65 with name: Ben Afflec and email: a@b.c
```