




# Everything is an object

Almost, but all objects are of type  
Object!



# In Java, every class is actually a subclass of Object

...or has a superclass which has Object as superclass...

There is a class called `java.lang.Object` in the Java API. The designers of Java have decided that this class is the superclass of all classes.

This means that even if we write our own class, it will too be a subclass of Object (you may think of subclass as subtype).

Object is the most abstract abstraction we can have in Java - it is so abstract that it merely describes what is common between all instances of any class.

This level of abstraction focuses on purely programmatical features, what we can do with instances/objects in general.

# Any instance can be viewed as being of type Object

Since Object is the super-super-duper class of all classes, we can actually choose to view any instance as being of this type.

This means that this would be perfectly legal (if we have a class called Passport, and a class called Member):

```
Object obj = new Passport();  
Object anotherObj = new Member();
```

But now we can only use the references as if they truly were “only” of Object type. We can only call methods declared in `java.lang.Object`!

# Advantages of the Object abstraction

It is convenient to know that any object of any type can represent itself as a String. The method `toString()` declared in `Object` guarantees this! And since every class ultimately is a subclass of `Object`, every instance of any class will have a `toString()` method.

However, the implementation in the class `Object` is not so fancy, since it is declared in a class which is very general. In fact, the only thing we get back when calling the version of `toString()` which we get from `Object`, is the qualified class name and an "@" and some hexadecimal number (all as a String of course).

# Another advantage is equality check

Another good thing with having every instance inheriting behavior from `Object`, is that every object has a method for checking if it is “equal” to any other object (passed as parameter to the `equals` method).

A companion to `equals()` is `hashCode()`, which produces a unique `int` number for an object, and the deal here is that to objects which are considered equal, also returns the same `hashCode()` number.

Using `hashCode()` is often more efficient than using the `equals()` method.

As with `toString()` the implementations of `equals()` and `hashCode()` we get from `Object`, are not so sophisticated. We'll soon look at how to deal with this.

# How is it syntactically expressed to inherit Object?

To inherit a class in Java can be expressed in two ways. The way this chapter deals with, is called “extending a class” using the keyword `extends`.

So, for instance `java.lang.String` extends `java.lang.Object`. In fact, all classes which do not explicitly extend some other class extends `Object`.

When we write a class, we could actually say in the class declaration:

```
public class MyClass extends Object
```

But because of the rule that all classes implicitly extend `Object`, we don't have to.

# If we'd want to, we could extend some other class

Using the keyword `extends`, we could explicitly extend some other class than `Object`, which then would create a new subtype (subclass) of that explicitly named class.

If we go back to our example of writing a file manager, and the abstractions of `File` and the subtype `MediaFile` (with in turn its subtypes `AudioFile` and `VideoFile`), the `MediaFile` class could be declared as:

```
public class MediaFile extends File
```

(Note that there are alternative ways to achieve this flexibility and hierarchy!)

# What methods are not declared in Object?

Since Object is an abstraction of a runtime object of any type in the programming language Java, it has a limited set of methods declared.

There is, for instance, no `thumbnail()` method in Object, since having a method for rendering a thumbnail image, is not typical for every Java object of any class...

Also, there is no `changeEmail()` method in Object, since the Java API designers didn't anticipate that every class would declare an instance variable for an email address and thus a need for changing it ;-)

Therefore the methods in Object are more general and less specific.



# So methods are “inherited”?

Yes, the methods in a superclass are inherited in subclasses. That’s just a technical way of saying that a class which extends (inherits from) a superclass, gets all the (public and protected) methods from the superclass, without the need of repeating the same code declaring the methods.

And the methods inherited, are inherited “as-is”, so if we want to change them, we actually have to write that code ourselves (as we will soon see).

Constructors are not inherited. Private methods and variables are “inherited” but not accessible from the subclasses, since private stuff is only usable from code in the very same class.