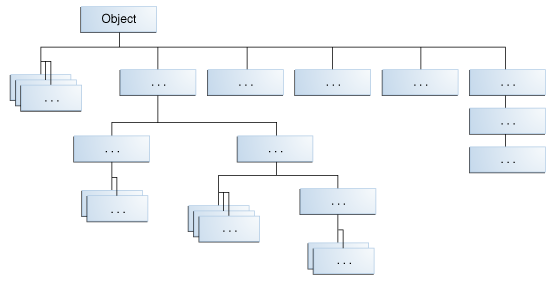
**Inheritance**

* Classes can be derived from other classes, thereby *inheriting* fields and methods from those classes.
* Java does not have multiple inheritance of classes, only interfaces
* Inheritance is specified through the extends keyword
* Is-A relationship. This is different from a Has-A relationship

**Why is this useful?** Enables new objects to take on the properties of existing objects

**Object**

* Object is defined in the java.lang package
* All classes inherit from object
* It defines and implements behavior common to all classes, including the ones individual programmers write.
* E.g. equals(), toString(), hashcode()



**Subclass**

A subclass inherits all public methods of its parent. Private instance variables are not inherited, but can still be accessed via public getters and setters that have been inherited.

* The inherited fields can be used directly, just like any other fields.
* You can declare a field in the subclass with the same name as the one in the superclass, thus *hiding* it (not recommended).
* You can declare new fields in the subclass that are not in the superclass.
* The inherited methods can be used directly as they are.
* You can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus *overriding* it.
* You can write a new *static* method in the subclass that has the same signature as the one in the superclass, thus *hiding* it.
* You can declare new methods in the subclass that are not in the superclass.
* You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.

Remember that unless explicitly called, there is always an implicit super call to the no-args constructor

**instanceof Operator**

* Checks whether the object is an instance of the specified type (class or subclass or interface)
* Returns true or false
* Returns false also if obj is null
* Syntax:

boolean example = obj instanceof Object;

**Overriding**

* An instance method in a subclass with the same signature (name, plus the number and the type of its parameters) and return type as an instance method in the superclass overrides the superclass's method
* Can hide a method by writing a new static method with the same signature
* E.g.writing our own toString() overrides Object’s toString()
* The @Override tag is useful but not necessary. You can use it to have the compiler check you
* Write your overrides in such a way that maximizes code resuse!

**Overloading**

* Defining two or more methods within the same class that share the same name but have different parameters is called overloading methods.
* E.g. constructor chaining