**Object Oriented Development using Java**

OOD Week 1 – Module 9

Inheritance - classes

Tutorial

© FDM Group Ltd 2020. All Rights Reserved.

Any unauthorised reproduction or distribution in part  
or in whole will constitute an infringement of copyright.

# What does this tutorial cover?

This tutorial will introduce you to the idea of classes inheriting attributes and behaviours from other classes. You’ll see how inheritance can be used to create loosely coupled dependencies between classes and how this can make our code more flexible, maintainable and easy to extend.

# How long will the tutorial take to complete?

1 hour

# What should you have already completed?

Modules 1 to 7 (up to and including Classes)

# What do you need?

In order to complete this tutorial exercise you will need:

* Java Development Kit 1.8 or above
* Apache Maven
* Eclipse IDE Kepler or above

# What does this Tutorial cover?

* What inheritance is.
* Uses of inheritance.
* The Object class.
* Vertical constructor chaining.
* The protected access modifier.
* Final classes.
* Abstract classes.

# What is inheritance?

Inheritance is a process where one class (the child) acquires the attributes and behaviours of another class (the parent). We’ll see that inheritance gives us major benefits in terms of making our code more flexible, easy to maintain and extend.

Inheritance is easy to spot. It’s always an ‘is a’ relationship between 2 classes:

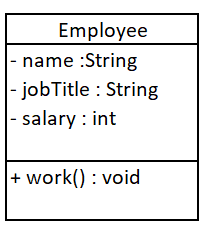
Manager is an employee.

House is a building.

Car is a vehicle.

Smart phone is a phone.

Let’s start with a simple example. We’ll create a class called Employee:



Let’s say that we now want to create another class called Manager. If you think about it, a manager is just another employee, but with some extra attributes and behaviours.

We could write a Manager class from scratch, but it’s much easier and more efficient to inherit attributes and behaviours from the Employee class:

**public** **final** **class** Manager **extends** Employee {

**private** ArrayList<Employee> team = **new** ArrayList<>();

**private** String teamName;

**public** **void** manage() {

System.***out***.println(getJobTitle()+" "+getName()+" is managing their team");

}

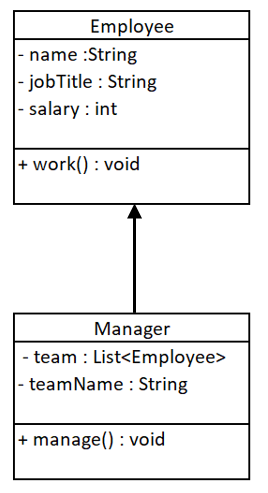
// getters & setters

}

Notice that when we write the Manager class we use the ‘extends’ keyword to make it inherit from the Employee class. Inside the class we only need to create attributes and behaviours which are specific to the Manager class.

When we create an object of the Manager class, all of the methods from both Manager and Employee will be available.

Here’s the Manager & Employee class in UML:



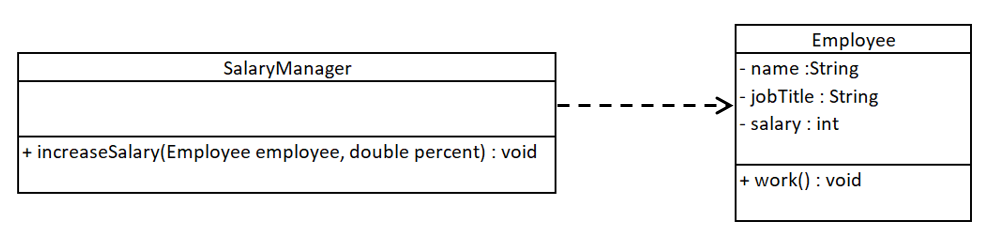
Notice the solid arrow with the solid head pointing from Manager to Employee. This indicates that the Manager class is a child of the Employee class. The technical term for this kind of relationship is a ‘generalisation’.

# Uses of inheritance

Based on what we’ve seen so far the obvious advantage of using inheritance is that it’s saved us from having to re-write code in the Manager class. This however is NOT the main benefit of using inheritance.

Let’s re-iterate that point: **saving time re-writing code is NOT the main benefit of using inheritance!**

OK, so what is the main benefit of using inheritance? The main benefit is that it makes our code significantly more flexible and easier to maintain. Here’s an example:



All objects of Employee’s child classes can be treated as if they were Employee objects. Therefore the increaseSalary() method of the SalaryManager class can take Manager objects as an argument as well as Employee objects. All of these objects will have the getSalary() and setSalary() methods that increaseSalary() needs to use. In fact we can write as many child classes of Employee as we like and all of their objects can be fed into increaseSalary() as an argument.

Without inheritance, we’d have to write one version of increaseSalary() for Employees and another version for Managers. If there were any other types such as ‘Contractor’ we’d need to write an increaseSalary method for them too. Inheritance means we only need one version of the increaseSalary() method which can handle Employee and all of its child classes.

Employee employee = **new** Employee();

Manager manager = **new** Manager();

SalaryManager salaryManager = **new** SalaryManager();

salaryManager.increaseSalary(employee, 5.5);

salaryManager.increaseSalary(manager, 3.9);

This flexibility doesn’t just include method arguments. Objects of a child class can be used in place of member variables or return types of its parent class.

# The Object class

The Object class is a built in class which is the ancestor of all classes in Java.

If you write a custom class without using the ‘extends’ keyword, your class will automatically have Object as its parent. This means that your class will inherit all of Object’s methods. It also means that any child classes of your class will also inherit all of Object’s methods. In fact every class in Java inherits 9 methods from the Object class.

Here are the 9 methods in Object:

* equals(Object obj)
* getClass()
* hashCode()
* notify()
* notifyAll()
* toString()
* wait()
* wait(long timeout)
* wait(long timeout, int nanos)

# Constructors

Constructors are the only part of a parent class which is not inherited by a child class. This means that if a parent class has a custom constructor, we must also write a custom constructor in the child class.

To complicate things, the constructor in the child class must call the constructor in its parent class using the super() keyword. Here’s an example:

**public** Employee(String name, String jobTitle, **int** salary) {

**super**();

**this**.name = name;

**this**.jobTitle = jobTitle;

**this**.salary = salary;

}

Notice that Employee’s constructor calls the default constructor of its parent (the Object class) using super(). Now if we look at the Manager class:

**public** Manager(String name, String jobTitle, **int** salary, String teamName) {

**super**(name, jobTitle, salary);

**this**.teamName = teamName;

}

Manager calls the constructor in Employee using super(name, jobTitle, salary).

Try commening out super(name, jobTitle, salary) in Manager. You’ll find that the Manager class no longer compiles.

If you try commenting out super() in Employee, there won’t be a compile error as the Java compiler implicitly inserts a call to super() if you haven’t written one.

Use of the super() keyword is known as ‘vertical constructor chaining’ as it links together constructors in parent and child classes.

# The protected access modifier

You should already be familiar with the private and public access modifiers. An additional access modifier which can be used is ‘protected’.

If an attribute or a method is marked as protected it can be accessed by other classes in the same package. It can also be accessed by any child classes.

Below is an example:

**public** **class** Employee {

**private** String name;

**protected** String jobTitle;

**private** **int** salary;

// getters & setters

}

In the Manager class the Employee attributes can be accessed as follows:

**public** **void** manage() {

System.***out***.println(jobTitle+" "+getName()+" is managing their team");

}

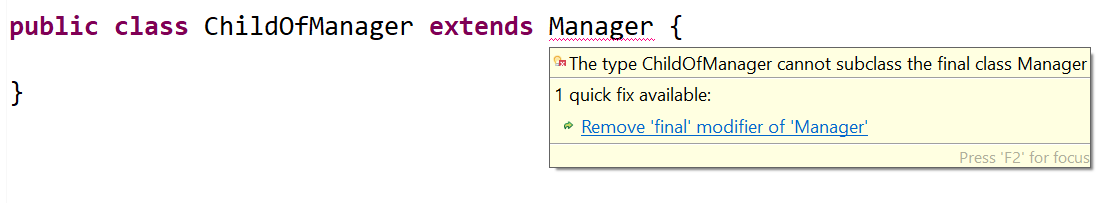
Notice that the protected attribute, jobTitle is being accessed directly in the same way as it would if it was an attribute of the Manager class. Name on the other hand needs to be accessed indirectly through the getName() method. This is because private attributes are not visible outside the class they’re declared in.

# Final classes

The final keyword can be used in a class declaration to stop that class from having children. This can be used to define the bottom of a hierarchy of classes. In the example below, we’re going to make Manager a final class:

**public** **final** **class** Manager **extends** Employee {}

If we now try to create a child of Manager, we get a compile error:



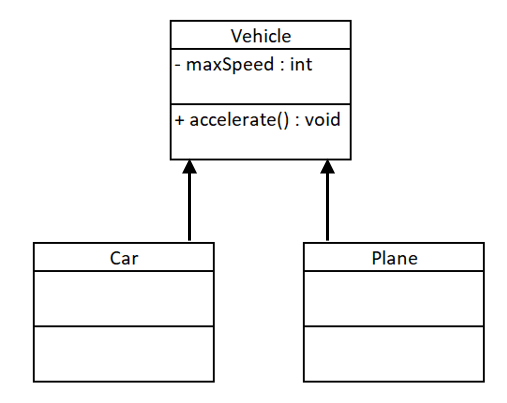
Some built in Java classes such as String are final. Others such as ArrayList are not.

Try making a child class of String and a child class of ArrayList. One will give you a compile error and the other won’t.

# Abstract classes

We’ve seen in our earlier examples that child classes are a more specific type of their parent class. For instance Manager is a specific type of Employee.

If we look at things from the opposite direction, we could say that the parent class is the more general type. Let’s now look at another hierarchy of parent and child classes:



In this case Car and Plane are children of the Vehicle class. Car and Plane are specific types of Vehicle, but Vehicle is a very general type. In fact, Vehicle is so general that it’s highly unlikely that we’d want to create an object of type Vehicle. Just think about reality: you’ve probably seen plenty of cars and planes, but it’s unlikely you’ve ever seen a generic vehicle driving the streets!!

This is where abstract classes come in. An abstract class is a class which a class which cannot be used to make an object. In fact it only exists for inheritance purposes. Its job is to define the top of a hierarchy of related classes.

Making an abstract class is easy, we just add the ‘abstract’ keyword to its definition:

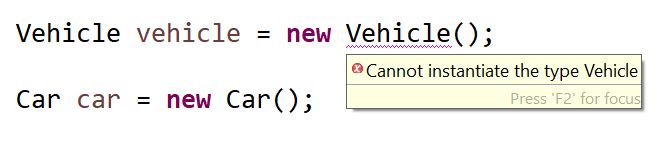
**public** **abstract** **class** Vehicle {}

Let’s now make a child class of Vehicle:

**public** **class** Car **extends** Vehicle {}

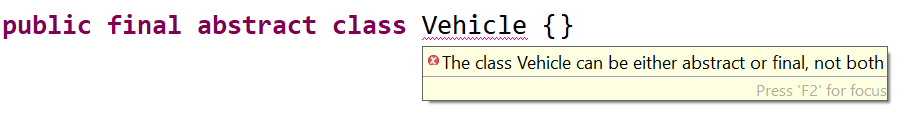
So far everything is the same as we saw in our Employee/Manager example, except for the abstract keyword in the definition of the parent class.

Let’s now try creating objects of the Vehicle class and the Car class:



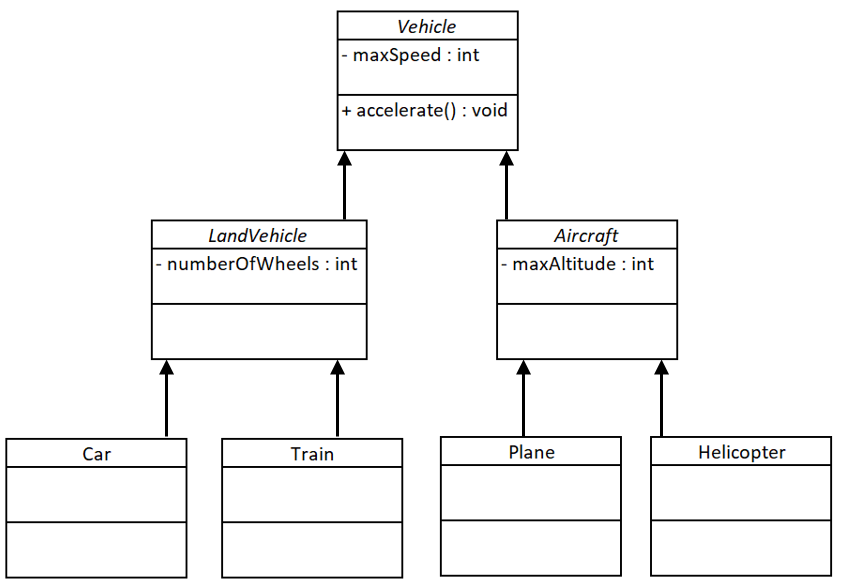
Notice that we get a compile error when we try to make an object of the Vehicle class.

At this stage it’s worth pointing out that you can’t have a final abstract class. Stop for a moment and think about why this is:



The only purpose of an abstract class is to have children, but a final class can’t have children. Therefore a class can’t use both of these keywords.

In UML an abstract class is represented by having its name in italics:



We can see in the UML above that there can be multiple abstract classes in an inheritance hierarchy. At each layer of the hierarchy, classes become more specific and gain extra attributes and behaviours.

The classes at the bottom of the hierarchy (Car, Train, Plane & Helicopter) are known as ‘concrete classes’. This means that we can create objects from them.

Finally, it’s worth noticing that each class can only have a single parent, but can have multiple children.

# Abstract methods

Let’s look at the methods in our abstract Vehicle class:

**public** **abstract** **class** Vehicle {

**private** **int** maxSpeed;

**public** **void** accelerate() {

// generic code to make a vehicle accelerate

}

**public** **int** getMaxSpeed() {

**return** maxSpeed;

}

**public** **void** setMaxSpeed(**int** maxSpeed) {

**this**.maxSpeed = maxSpeed;

}

}

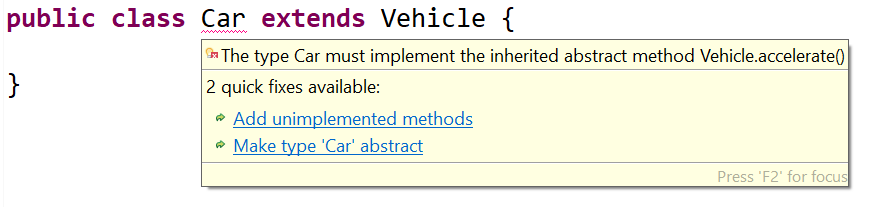
The methods getMaxSpeed() and setMaxSpeed() do something very specific i.e. accessing the maxSpeed variable. These will be inherited by all the child classes and will do the same job for each of them.

The accelerate method could cause us a problem though. In our Vehicle class we’ve written some very generic code in the method. The big problem here is that every type of vehicle accelerates differently. The car uses an internal combustion engine (or an electic engine). Planes use a jet engine. The generic code we’ve written for accelerate in the Vehicle class is unlikely to be suitable for any of the child classes. In fact there was really no point in writing the accelerate code in the Vehicle class at all.

In this case we should make accelerate an abstract method. This means that the method in Vehicle will have no body and therefore no code:

**public** **abstract** **void** accelerate();

When we create the Car class and the Plane class we’ll be forced to write code for the accelerate method which is suitable for each of these classes. This is known as ‘implementing’ the method. Until we implement the abstract method, the child class won’t compile.



Each child class will now have its own implementation of the accelerate() method:

**public** **class** Car **extends** Vehicle {

**public** **void** accelerate() {

// code to accelerate a car

}

}

**public** **class** Plane **extends** Vehicle {

**public** **void** accelerate() {

// code to accelerate a plane

}

}