**Object Oriented Development using Java**

OOD Week 1 – Module 11

Polymorphism

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 using polymorphism to create different versions of the same method. You’ll become familiar with the two different types of polymorphism and see some concrete applications of the principle.

# How long will the tutorial take to complete?

1 hour

# What should you have already completed?

Modules 1 to 9 (up to and including Inheritance – 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 is polymorphism?
* Overriding / late binding / runtime polymorphism
* Overriding the equals() and hashCode() methods
* Overloading / early binding / compile time polymorphism
* Horizontal constructor chaining

# What is polymorphism?

Polymorphism means ‘multiple forms’. In Java it means that there can be multiple forms of the same method. It’s one of the 4 pillars of Object Oriented Programming. There are 2 forms of polymorphism:

* Overriding – also known as late binding or runtime polymorphism
* Overloading – alson known as early binding or compile time polymorphism.

# Overriding

Overriding is related to inheritance. It’s something that you will only find in child classes. Here’s a simple example:

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

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

// code to accelerate a car

}

}

Our car class has an accelerate() method containing code to make a car accelerate. But what if we create a child class of car called ElectricCar? In reality an electric car accelerates very differently to a regular car. If we’re going to simulate an electic car properly we need some different code in our accelerate() method. To do this we’re going to override the accelerate() method in the ElectricCar class:

**public** **class** ElectricCar **extends** Car{

@Override

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

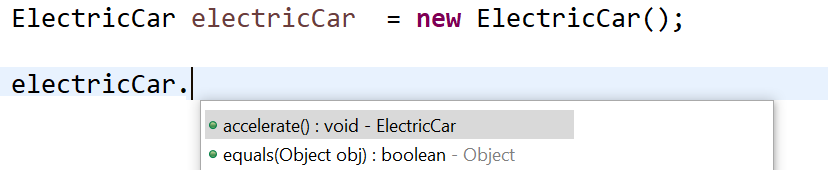
// code to accelerate an electric car

}

}

An override is very simple, it’s just a method in the child class with an identical header to the method in the parent class. The only difference is the code within the method.

If we make an object of our ElectricCar class, we’ll find that the only version of the accelerate() method that’s available is the one defined in ElectricCar:



The version of accelerate() we defined in ElectricCar has replaced the version we defined in Car.

## The @Override annotation

You might be wondering what the @Override annotation above the accelerate() method in ElectricCar does:

@Override

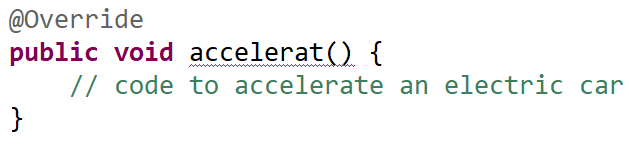
**public** **void** accelerate() {

// code to accelerate an electric car

}

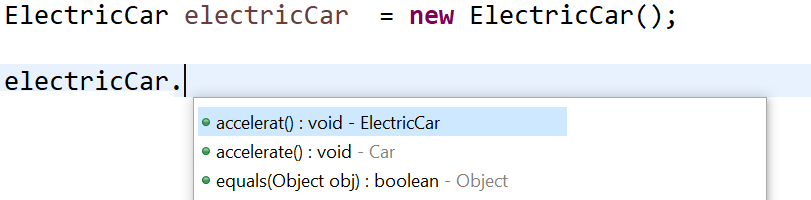
In our example, if you delete @Override, you won’t notice any difference, the code will still compile and run the same way. So what’s the point of it?

The purpose of @Override is to check that a method in a child class really is overriding a method in its parent class. Effectively it’s a way of checking for typos. Here’s an example:



If you miss-spell accelerate (in this case there’s an ‘e’ missing), @Override will force a compile error. This is because the name of the method in ElectricCar doesn’t match the name of the method in Car.

Without the @Override this would compile and we wouldn’t notice a problem. However the accelerate() method in Car wouldn’t’ have been overridden and we’d have two similarly named methods instead:



## Alternative terms for overriding

Overriding is also known as late binding or runtime polymorphism. Here’s a simple example to explain why:

**public** **class** Race {

**public** **void** startRace(Car car) {

car.accelerate();

}

}

The startRace() method takes a Car object as an argument. At the time of writing the method we have no idea whether a regular car or an electric car will be passed into the method. This means that we won’t know which version of the accelerate() method will actually be called until an object is passed into the startRace() method. This will only happen when the application is running. Hence runtime polymorphism, or late binding.

# Overriding the equals() and hashCode() methods

In this section we’re going to look at two of the most commonly overriden methods in Java. Before we look at how to override them, let’s start by looking at what the equals() method is used for.

### Equals

By now you will probably have used equals() many times to check if two String objects contain the same text. Hopefully you’ll remember that == is used to compare primitives, but equals() is used to compare objects.

Here’s an example. Let’s start by creating two identical car objects. We’ll pass their make and engine size into the constructor:

Car car1 = **new** Car("Ford",2.0);

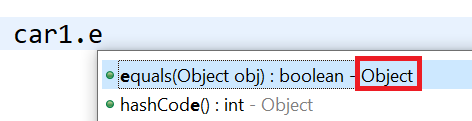
Car car2 = **new** Car("Ford",2.0);

Let’s now use the equals() method to see if the two cars have the same attributes:

System.***out***.println(car1.equals(car2));

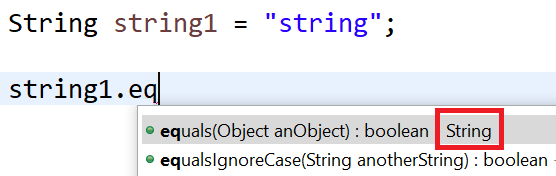
You’re probably expecting the equals() method to return true, but actually it returns false. This wasn’t a problem when you used the equals() method to compare two Strings. So what’s the problem?

Let’s look a bit more closely:



We can see that Car’s equals() method is defined in the Object class.

But String’s equals() method is defined in String. The equals() method has been overridden in the String class.



So why was the method overridden in String. Let’s have a look at the source code for the Object class:

**public** **boolean** equals(Object obj) {

**return** (**this** == obj);

}

You can see that the equals() method defined in the Object class doesn’t do much. All it does is compare two objects using the == operator. The method in Object is really just acting as a placeholder. To make it work for a class you’ve written you’ll need to override the method.

The overridden method needs to compare each of the attributes within the two objects. This can be quite long if the class has a lot of attributes. The good news is that your IDE can automatically generate the equals() method for you. If you’re using Eclipse, just go to the Source menu and choose ‘Generate hashCode() and equals() methods’.

### HashCode

The hashCode() method is another method defined in the Object class. Like equals() it’s a way of checking if two objects contain the same attributes. It’s commonly used by some of the built in Collection classes that you’ll see in week 2 of OOD.

The hashCode() method calculates the hash code for an object. The hash code is a slightly less reliable way than equals() of checking if two objects contain the same data. If two objects are identical, they will have the same hash code. If two objects are different they will probably have different hash codes. Note the word ‘probably’. The hash code is an int and so has a limited number of possible values. The number of possible object values is infinite. So it’s possible for two different objects to have the same hash code.

Let’s try getting the hash codes of our two identical Car objects:

Car car1 = **new** Car("Ford",2.0);

Car car2 = **new** Car("Ford",2.0);

System.***out***.println(car1.hashCode());

System.***out***.println(car2.hashCode());

Despite the objects having the same attributes, we get two different hashCodes:

2018699554

1311053135

As with equals() the hashCode() method needs to be overridden in classes you’ve written if it’s to work properly. As with equals() the easiest way to override the hashCode() method is to get your IDE to automatically generate it.

# Overloading

Unlike overriding, overloading is not related to inheritance and can be done in any class. An overloaded method is one where there are two or more versions of the method but which take different arguments.

An example that you’re probably familiar with is the substring() method of the String class. The two versions are as follows:

string.substring(4);

string.substring(2, 5);

The first version creates a String containing all of the characters of the original String from position 4 onwards.

The second version creates a String containing the characters 2,3 & 4 of the original String.

Let’s have a look at how we can overload the accelerate() method of our Car class:

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

**public** **void** accelerate(**int** speedLimit) {}

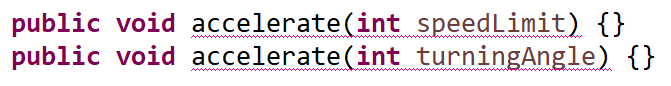
If our code calls accelerate() with no arguments, the first version will run. If it’s called with an int argument, the second version will run().

We can have as many overloaded versions of a method as we like, as long as the datatypes of their arguments are different. Both of these would work:

**public** **void** accelerate(**int** speedLimit, **double** gradient) {}

**public** **void** accelerate(**double** windSpeed, **int** speedLimit) {}

But these wouldn’t:



## Alternative terms for overloading

Overloading is also known as early binding or compile time polymorphism.

**public** **class** Race {

**public** **void** startRace(Car car) {

car.accelerate(70);

}

}

We’ve coded our startRace() method to use the version of accelerate which takes an int. It doesn’t matter which type of car gets passed into the startRace() method at runtime, the int version of the accelerate() method will always run. Remember that with overriding we didn’t know which method would run until the application ran. With overloading we know when we write the code which version of the accelerate() method is going to be used. Hence the terms early binding or compile time polymorphism.

## Widening

In the example below we’ve got 3 overloaded methods, one takes a byte, one takes a short and the other takes a long:

**public** **class** Test {

**public** **void** overloadedMethod(**byte** number) {

System.***out***.println("byte");

}

**public** **void** overloadedMethod(**short** number) {

System.***out***.println("short");

}

**public** **void** overloadedMethod(**long** number) {

System.***out***.println("long");

}

}

In our main method we’re going to call overloadedMethod with the argument 5. Which of the three versions of the method do you think is going to run?

**public** **static** **void** main(String[] args) {

Test test = **new** Test();

test.overloadedMethod(5);

}

Chances are you thought that the ‘short’ or ‘byte’ versions were going to run. If you thought that you’d be wrong. Actually, it’s the ‘long’ version that runs.

Why is this? The number 5 will fit easily into either a byte or a short. Well it all comes down to the fact that int is the default datatype for whole numbers. Java sees the number 5 as an int. It then looks at our three overloaded methods to see which of them can handle an int. It’s not bothered about the actual value of the argument.

In this case an int value might not fit into a byte or a short but it definitely will fit into a long. For this reason the long version of the method is chosen.

# Overloading constructors

We’ve seen that we can overload methods. This also applies to constructors. Here’s an example:

**public** Car(**double** engineSize) {

// code to set attributes

}

**public** Car(**double** engineSize, **int** numberOfDoors) {

// code to set attributes

}

**public** Car() {

// no code in this constructor

}

Each constructor sets different combinations of attributes. Notice that the third constructor doesn’t take any arguments or set any attribute values. It does the same thing as the default constructor. You might wonder what the point is of making a constructor which does nothing, but you’ll see in later weeks there are some frameworks which require classes to have a no args constructor.

# Horizontal constructor chaining

In the inheritance model you saw how constructors in child classes can be chained to constructors in parent classes using the super() keyword. This was known as vertical constructor chaining.

We’re now going to look at something called horizontal constructor chaining. This is where constructors in the same class are chained together using the this() keyword.

Let’s start with our first constructor:

**public** Car(**double** engineSize) {

**if** (engineSize <= 0) {

**this**.engineSize = 1;

} **else** {

**this**.engineSize = engineSize;

}

}

The constructor includes a check to make sure that we don’t try to set a zero or negative engine size. If we pass in zero or a negative number the engine size will be set to a default value of 1.

Now let’s add in a second overloaded constructor:

**public** Car(**double** engineSize, **int** numberOfDoors) {

**this**(engineSize);

**if** (numberOfDoors < 1) {

**this**.numberOfDoors = 1;

} **else** {

**this**.numberOfDoors = numberOfDoors;

}

}

To avoid duplicating the code that checks the engine size, our second constructor simply passes the engineSize argument to the first constructor to deal with using this(engineSize). The second constructor now sets a default number of doors if the number of doors passed in is less than 1.

Let’s add in a third and final constructor:

**public** Car(**double** engineSize, **int** numberOfDoors, String model) {

**this**(engineSize,numberOfDoors);

**this**.model = model;

}

Although this constructor sets the most attributes, it does the least work. It passes the engineSize and numberOfDoors to the second constructor using this(engineSize,numberOfDoors).

Using horizontal chaining has significantly reduced the amount of code we’ve needed to write in our overloaded constructors.