**Object Oriented Development using Java**

OOD Week 1 – Module 10

Inheritance - interfaces

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 an alternative way of doing inheritance using interfaces. You’ll see that interfaces give us significantly more flexibility than we would get using abstract classes.

# 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 an interface is.
* Uses and benefits of interfaces.
* Comparison of interfaces and abstract classes.

# Overview

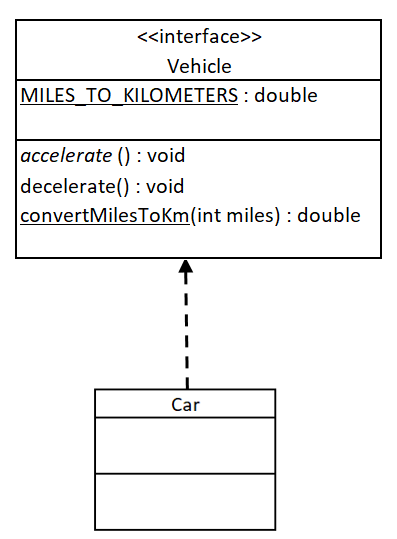
In the previous module we saw how an abstract class can be used to create common functionality amongst multiple classes. We saw that inheritance allows us to create code which is flexible, extensible and easy to maintain.

Interfaces give us an alternative way to do inheritance. In this tutorial, we’ll see that interfaces give us the same major benefits as abstract classes. However they also give us some additional benefits. That said, there are still some things which can be done with an abstract class but not with an interface.

In modern Java development, interfaces tend to be more commonly used than abstract classes.

## A simple example in UML

We’ll start by revisiting the example we used for abstract classes. Only this time we’ll use an interface in place of an abstract class:



The most obvious differences you’ll notice are that the interface has <<interface>> above its name and the arrow pointing from Car is broken. The broken arrow with a solid head represents inheritance from an interface. It’s known as a ‘realisation’.

A slightly more subtle difference is the lack of any access modifiers. The reason for this is that all attributes and methods in an interface are implicitly public. If you try to use the private or protected keywords, you’ll get a compile error.

Notice also that the MILES\_TO\_KILOMETERS attribute is in upper case and underlined, indicating that it’s final and static. In fact all attributes in interfaces are implicitly final and static.

Finally, notice that we have 3 kinds of methods: accelerate() is an abstract method (notice the italics), decelerate() is a default (or non-abstract) method, convertMilesToKm() is a static method.

So far, so good. But right now you’re probably thinking ‘so what?’! The interface looks a lot like the abstract class but you can’t have variables or use the private keyword. Why would we bother to use an interface? We’ll find out why shortly, but before that let’s have a look at the example in code.

## A simple example in code

**public** **interface** Vehicle {

**double** ***MILES\_TO\_KILOMETERS*** = 1.6;

**void** accelerate();

**default** **void** decelerate() {

// generic code to decelerate a vehicle

}

**static** **double** convertMilesToKm(**int** miles) {

**return** miles \* ***MILES\_TO\_KILOMETERS***;

}

}

Notice that there are no access modifiers. Remember that all attributes are implicitly public, so there’s no need to use the public keyword.

Notice also that the MILES\_TO\_KILOMETERS attribute doesn’t use the final or static keywords. Again this is because all attributes in an interface are implicitly final and static.

Finally, notice that the abstract method accelerate() doesn’t use the abstract keyword. This is because interface methods are abstract unless another keyword is used. In contrast with an abstract class, it’s the non-abstract method which needs to be marked out, in this case by using the ‘default’ keyword.

Now let’s look at how a class inherits from (implements) an interface:

**public** **class** Car **implements** Vehicle {

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

// code to accelerate a car

}

}

Notice that the big difference between inheriting from an interface and from an abstract class is the word ‘implements’.

As with inheriting from an abstract class, we have to write the code for the abstract method. The default and static methods are implicitly present.

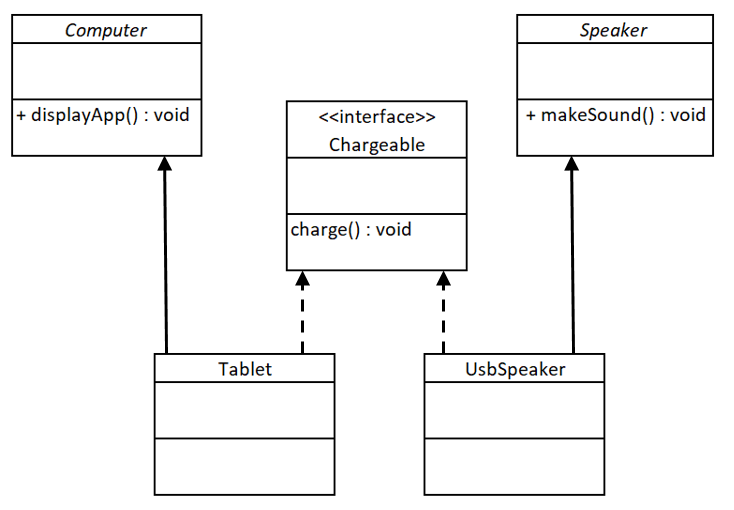
# Benefits of interfaces

In our simple example, we haven’t exactly sold interfaces to you. So far they look like abstract classes but more limited.

Let’s look at some examples of where an interface can do things that an abstract class can’t do.

## Benefits example 1

Take a look at the UML below. Could the Chargeable interface be replaced with an abstract class?



**public** **class** Tablet **extends** Computer **implements** Chargeable {}

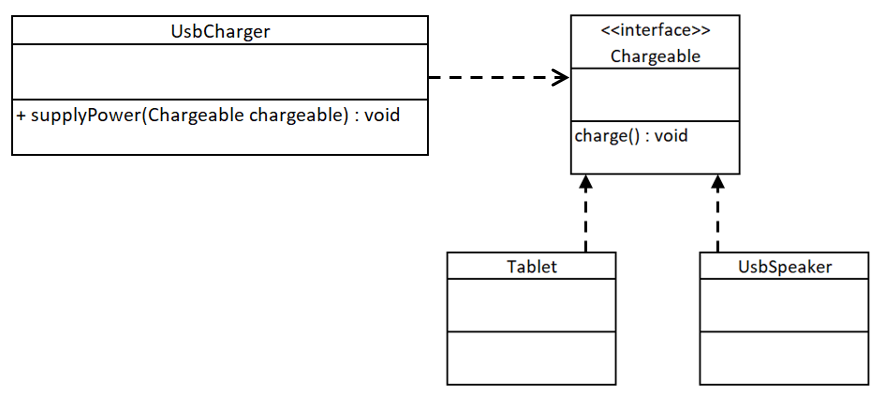
**public** **class** UsbSpeaker **extends** Speaker **implements** Chargeable {}

Hopefully, you’ve worked out that the answer’s ‘no’. You should remember that a class can only have one parent class. Tablet already has Computer as its parent and UsbSpeaker has Speaker as its parent. There’s no possibility of getting them to inherit from a second parent class.

Interfaces get around this problem. Tablet inherits the displayApp() method from Computer and the charge() method from Chargeable. UsbSpeaker inherits the makeSound() method from Speaker and the charge() method from Charageable.

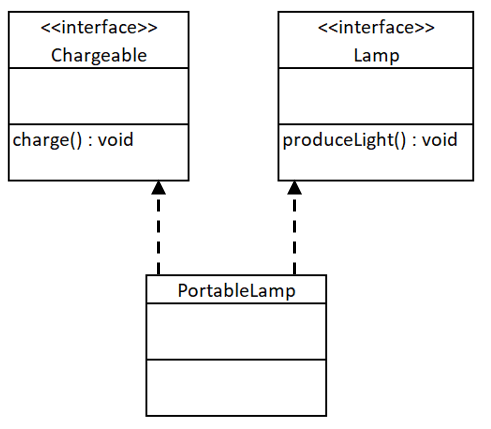
We’ve seen that the main purpose of an abstract class is to define the top of a hierarchy of related classes. The main purpose of an interface is to define common functionality shared between different and possibliy unrelated classes.

As with an abstract class, any object implementing an interface can be treated as if it was an object of the interface. So in the example below, UsbCharger’s supplyPower method can take a tablet object or a UsbSpeaker object as an argument. It will also be able to take objects of any other classes implementing Chargeable.



## Benefits example 2

We’ve seen that the major drawback of class inheritance is that a class can only have a single parent class. That’s not a problem with interfaces. In the example below we can see the PortableLamp class implementing two different interfaces: Chargeable and Lamp. This means that a PortableLamp object could be passed into a method requiring a Chargeable object or a method requiring a Lamp object as an argument.

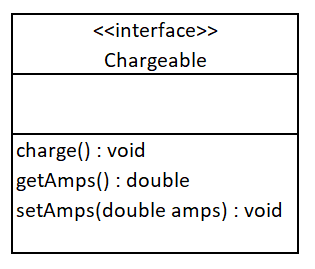


**public** **class** PortableLamp **implements** Chargeable, Lamp {}

Classes can implement any number of interfaces. This means that interfaces give us much more flexibility than an abstract class could.

# Variables

As we’ve seen already, interfaces can’t have variables. However there’s no reason why they couldn’t be used to define getters and setters:



**public** **interface** Chargeable {

**void** charge();

**void** setAmps(**double** amps);

**double** getAmps();

}

In this case the amps variable would need to be created in each class implementing Chargeable. The code in the getters and setters would then be need to be written to manage this variable.

# Summary

## Differences between abstract classes and interfaces

Abstract classes allow us to define a hierarchy of closely related classes. Interfaces allow us to define common functionality across multiple classes which are otherwise unrelated.

A class can only inherit from a single class, but it can inherit from multiple interfaces.

An abstract class can have variables, an interface can’t.

An abstract class can have a constructor, an interface can’t.

## Similarities between abstract classes and interfaces

Like an abstract class, an interface allows us to write flexible code which is extensible and easy to maintain.

Abstract classes and interfaces can both have abstract, non-abstract (default) and static methods.

Neither abstract classes nor interfaces can be instantiated.