**Object Oriented Development**

Topic of Discussion

Tutorial

**Comparators**

© 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 discuss methods

# How long will the tutorial take to complete?

2 hours

# 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
* Any extra software should go here (Git, SQL, etc.)

# What does this tutorial cover?

* Introduction
* The Comparable interface
* The Comparator interface
* Using the Comparable interface
* The Collections class

## Introduction

When you use Collections you may require a customised sort​.

​

Java provides two interfaces to do this:​

* Comparable​
* Comparator​

​

## The Comparable interface

Comparable is an interface with a single method: ​

**public** **int** compareTo(Type otherObject){

return someValue;

}

​

A class that implements this interface allows an object to be compared with another object. The return values can be one of the following:​

​

* 0 if they are the same​
* a positive value​
* a negative value​

​

**public** **class** User **implements** Comparable<User> {

**private** String username;

**public** **int** compareTo(User u) {

// Code to compare a User object to another User // object

}

}

To use this you would write the code:

User user1 = **new** User();

User user2 = **new** User();

user1.compareTo(user2);

## The Comparator interface

If your class implements the Comparable interface then it can become bloated. ​This is because the class has to include code to store member variables as well as code for comparing against other objects of that type. This breaks Single Responsibility.​

Comparator is a similar interface and it contains the following method:​

​

​

**public** **int** compare(Type object1, Type object2) {

**return** someValue;

}

A class that implements this allows you to compare two objects.​

​

* The return values can be one of the following:​
* 0 if they are the same​
* a positive value​
* a negative value​
* This is the preferred technique because it does not break Single Responsibility​

​

​

**public** **class** UserComparator **implements** Comparator<User> {

**public** **int** compare(User u1, User u2) {

// Code to compare Users goes in here

}

}

User user1 = **new** User();

User user2 = **new** User();

UserComparator userComparator = **new** UserComparator();

userComparator.compare(user1, user2);

**Using the Comparator interface**

Lets look at how the Comparator is actually used. The first part is the class which we want to compare:​

​

**public** **class** Car {

**private** **int** engineSize;

**public** Car(**int** engineSize) {

**this**.engineSize = engineSize;

}

**public** **int** getEngineSize() {

**return** engineSize;

}

}

Next, lets write code which compares Car objects.​

When the method returns a negative value, it means "the two items are in the right order and do not need swapping".

In this case, it is a car with a smaller engine **<** a car with a larger engine. ​

* This would reflect ascending order (the first car, car1 has a larger engine than the second car, car2) ​, so the cars are running from those with smaller engine sizes to those with larger ones.

**public** **class** CarComparator **implements** Comparator<Car> {

**public** **int** compare(Car car1, Car car2) {

**if** (car1.getEngineSize() < car2.getEngineSize())

**return** -1;

**else** **if** (car1.getEngineSize() > car2.getEngineSize())

**return** 1;

**else**

**return** 0;

}

}

The final part of the code is to create some cars in a List and call the sort method in the Collections class.​

Pass it two arguments:​

* the list of cars​
* A Comparator object​

​

CarComparator carComparator = **new** CarComparator();

System.***out***.println("sort cars by engine size");

List<Car> carsList = **new** ArrayList<Car>();

carsList.add(**new** Car(1000));

carsList.add(**new** Car(600));

carsList.add(**new** Car(500));

carsList.add(**new** Car(200));

Collections.sort(carsList, carComparator);

**for** (Car eachCar : carsList) {

System.***out***.print("this car has an engine size:");

System.***out***.println(eachCar.getEngineSize());

}

Note that in the code, you are not calling the compare method directly. However the compare method is being called many times indirectly when you call the sort method of the Collections class. When it compares any two of the Car objects in the list the return value indicates whether those Cars are in the right order(-1 is returned) or if they need to be swapped, if they are not. Obviously it is up to the developer to decide on how to order the items, ascending or descending, and based on what attribute.

To switch to descending order, two simple changes are required:

**public** **class** CarComparator **implements** Comparator<Car> {

**public** **int** compare(Car car1, Car car2) {

**if** (car1.getEngineSize() < car2.getEngineSize())

**return** 1;

**else** **if** (car1.getEngineSize() > car2.getEngineSize())

**return** -1;

**else**

**return** 0;

}

}

The code below now says that if first car has a smaller engine size than the second, then the value of 1 is returned, so they need to be switched. This represents descending order:

**if** (car1.getEngineSize() < car2.getEngineSize())

**return** 1;

Let’s write an example which compares an attribute which is a string:

The Car class is similar, we have added a member variable to represent the make of Car, amended the constructor and added a getter method:

**class** Car {

**private** **int** engineSize;

**private** String make;

**public** Car(**int** engineSize, String make) {

**this**.engineSize = engineSize;

**this**.make = make;

}

**public** **int** getEngineSize() {

**return** engineSize;

}

**public** String getMake() {

**return** make;

}

}

The next class is the Comparator:

**class** CarComparator **implements** Comparator<Car> {

**public** **int** compare(Car car1, Car car2) {

String make1=car1.getMake();

String make2=car2.getMake();

**if** (make1.compareTo(make2)<0){

**return** -1;

}

**if** (make1.compareTo(make2)>0){

**return** 1;

**}**

**else**

**return** 0;

}

}

The code below compares the two strings representing the makes of Car. If the first string is “less than” (alphabetically) then –1 is returned. This means they are in the right order. This means they are sorted in alphabetical order(ascending, A->Z). We have used the compareTo method in the String class:

String make1=car1.getMake();

String make2=car2.getMake();

**if** (make1.compareTo(make2)<0){

**return** -1;

}

The remaining code is very similar but is amended for the new constructor:

CarComparator carComparator = **new** CarComparator();

System.***out***.println("sort cars by make");

List<Car> carsList = **new** ArrayList<Car>();

carsList.add(**new** Car(1000,"Renault"));

carsList.add(**new** Car(600, "Ford"));

carsList.add(**new** Car(500, "Vauxhall"));

carsList.add(**new** Car(200, "Citroen"));

Collections.*sort*(carsList, carComparator);

**for** (Car eachCar : carsList) {

System.***out***.print("this car make is:");

System.***out***.println(eachCar.getMake());

}

The code above can also be shortened using lambda expressions, which comes later in the course.

## The Collections class

Collections is a class that provides functionality to operate on collections, for example:​

* Sorting ​
* Searching​
* Reversing the order of entries​