Collection framework

1. Java Collections

The Java Collections API provide Java developers with a set of classes and interfaces that makes it easier to work with collections of objects, e.g. lists, maps, stacks etc.

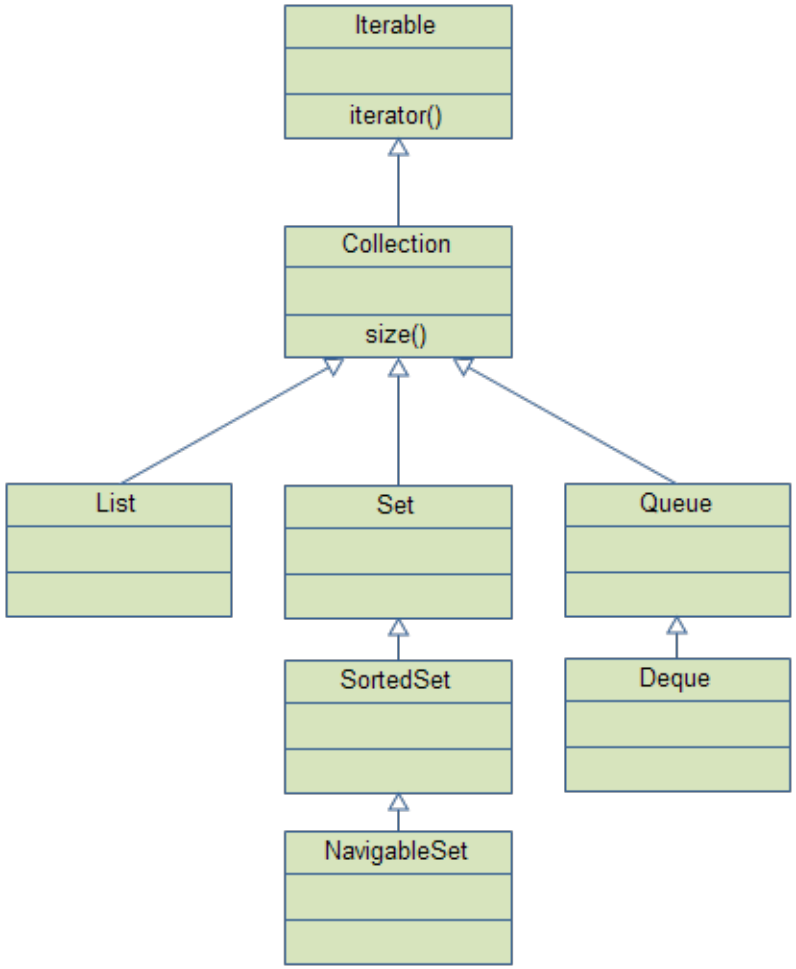
Rather than having to write your own collection classes, Java provides these ready-to-use collection classes for you. This tutorial will look closer at the Java Collections, as they are also sometimes referred to, and more specifically the Java Collections available in Java 8 and later.

The Java Collection interface represents the operations possible on a generic collection, like on a List, Set, Stack, Queue and Deque. For instance, methods to access the elements based on their index are available in the Java Collection interface. The Java Collection interface is explained in more detail in the Java Collection interface tutorial.

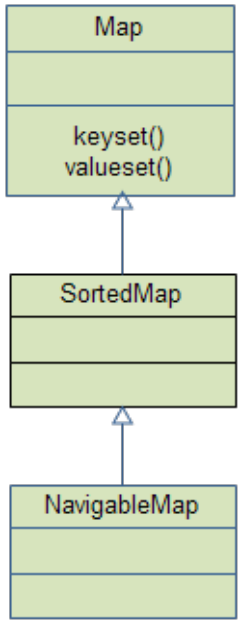
In order to understand and use the Java Collections API effectively it is useful to have an overview of the interfaces it contains. So, that is what I will provide here.

There are two "groups" of interfaces: `Collection`'s and `Map`'s.

Here is a graphical overview of the `Collection` interface hierarchy:



And here is a graphical overview of the `Map` interface hierarchy:



You can find links to explanations of most (if not all) of these interfaces and implementations in the sub-menu at the top right of this page. That top-menu exists on all pages in this trail.

2. Java Iterator

The Java `Iterator` interface represents an object capable of iterating through a collection of Java objects, one object at a time. The `Iterator` interface is one of the oldest mechanisms in Java for iterating collections of objects (although not the oldest - `Enumerator` predated `Iterator`).

2.1. Java Iterator Core Methods

The Java Iterator interface is reasonably simple. The core methods of the Iterator interface are:

Method	Description
hasNext()	Returns `true` if the Iterator has more elements, and `false` if not.
next()	Return the next element from the Iterator
remove()	Removes the latest element returned from next() from the Collection the Iterator is iterating over.
forEachRemaining()	Iterates over all remaining elements in the Iterator and calls a Java Lambda Expression passing each remaining element as parameter to the lambda expression.

Each of these methods will be covered in the following sections.

3. Java Iterable

The *Java Iterable* interface represents a collection of objects which is *iterable* - meaning which can be iterated. This means, that a class that implements the Java `Iterable` interface can have its elements iterated. You can iterate the objects of a Java Iterable in three ways: Via the , by obtaining a Java Iterator from the Iterable, or by calling the Java Iterable `forEach()` method. You will see examples of all three iteration methods later in this Java Iterable tutorial.

3.1. Iterate an Iterable With the for-each Loop

The first way to iterate the elements of a Java Iterable is via the Java for-each loop loop. Below is an example showing how to iterate the elements of a Java List via the Java for-each loop. Since the Java `List` interface extends the `Collection` interface, and the `Collection` interface extends the `Iterable` interface, a `List` object can be used with the for-each loop.

```
List<String> list = new ArrayList><();

list.add("one");
list.add("two");
list.add("three");

for( String element : list ){
   System.out.println( element.toString() );
}
```

This example first creates a new `List` and adds 3 elements to it. Then it uses a for-each loop to iterate the elements of the `List`, and print out the `toString()` value of each element.

3.2. Implementing the Iterable Interface

How you implement this `Iterable` interface so that you can use it with the for-each loop, is explained in the text Implementing the Iterable Interface, in my Java Generics tutorial.

Here I will just show you a simple `Iterable` implementation example though:

```
public class Persons implements Iterable {
  private List<Person> persons = new ArrayList<Person>();

  public Iterator<Person> iterator() {
    return this.persons.iterator();
  }
}
```

An instance of `Persons` can be used with the Java for-each loop like this:

```
Persons persons = ... //obtain Persons instance with Person objects inside.

for(Person person : persons) {
  // do something with person object.
}
```

4. Java List

The *Java List* interface, `java.util.List`, represents an ordered sequence of objects. The elements contained in a Java `List` can be inserted, accessed, iterated and removed according to the order in which they appear internally in the Java `List`. The ordering of the elements is why this data structure is called a *List*.

Each element in a Java `List` has an index. The first element in the `List` has index 0, the second element has index 1 etc. The index means "how many elements away from the beginning of the list". The first element is thus 0 elements away from the beginning of the list - because it is at the beginning of the list.

You can add any Java object to a `List`. If the `List` is not typed, using Java Generics, then you can even mix objects of different types (classes) in the same `List`. Mixing objects of different types in the same `List` is not often done in practice, however.

The Java `List` interface is a standard Java interface, and it is a subtype of the Java Collection interface, meaning `List` inherits from `Collection`.

4.1. Create a List

You create a List instance by creating an instance of one of the classes that implements the List interface. Here are a few examples of how to create a `List` instance:

```
List listA = new ArrayList();
List listB = new LinkedList();
List listC = new Vector();
List listD = new Stack();
```

4.2. Insert Elements in a Java List

You insert elements (objects) into a Java `List` using its `add()` method. Here is an example of adding elements to a Java `List` using the `add()` method:

```
List<String> listA = new ArrayList<>();

listA.add("element 1");
listA.add("element 2");
listA.add("element 3");
```

It is possible to insert an element into a Java `List` at a specific index. The `List` interface has a version of the `add()` method that takes an index as first parameter, and the element to insert as the second parameter. Here is an example of inserting an element at index 0 into a Java `List`:

```
list.add(0, "element 4");
```

4.3. Get Elements From a Java List

You can get the elements from a Java `List` using the index of the elements. You can do so using either the `get(int index)` method. Here is an example of accessing the elements of a Java `List` using the element indexes:

```
List<String> listA = new ArrayList<>();

listA.add("element 0");
listA.add("element 1");
listA.add("element 2");

//access via index
String element0 = listA.get(0);
String element1 = listA.get(1);
String element3 = listA.get(2);
```

It is also possible to iterate the elements of a Java `List` in the order they are stored in internally. I will show you how to do that later in this Java List tutorial.

4.4. Find Elements in a List

You can find elements in a Java `List` using one of these two methods:

- indexOf()
- lastIndexOf()

The `indexOf()` method finds the index of the first occurrence in the `List` of the given element. Here is an example finding the index of two elements in a Java `List`:

```
List<String> list = new ArrayList<>();

String element1 = "element 1";
String element2 = "element 2";

list.add(element1);
list.add(element2);

int index1 = list.indexOf(element1);
int index2 = list.indexOf(element2);

System.out.println("index1 = " + index1);
System.out.println("index2 = " + index2);
```

Running this code will result in this output:

```
index1 = 0
```

index2 = 1

4.5. Checking if List Contains Element

You can check if a Java `List` contains a given element using the `List` `contains()` method. Here is an example of checking if a Java `List` contains an element using the `contains()` method:

```
List<String> list = new ArrayList<>();

String element1 = "element 1";

list.add(element1);

boolean containsElement =
    list.contains("element 1");

System.out.println(containsElement);
```

The output from running this Java `List` example will be:

```
true
```

4.6. Remove Elements From a Java List

You can remove elements from a Java `List` via these two methods:

1. remove(Object element)
2. remove(int index)

`remove(Object element)` removes that element in the list, if it is present. All subsequent elements in the list are then moved up in the list. Their index thus decreases by 1. Here is an example of removing an element from a Java `List` based on the element itself:

```
List<String> list = new ArrayList<>();

String element = "first element";
list.add(element);

list.remove(element);
```

The `List` `remove(int index)` method removes the element at the given index. All subsequent elements in the list are then moved up in the list. Their index thus decreases by 1. Here is an example of removing an element from a Java `List` by its index:

```
List<String> list = new ArrayList<>();

list.add("element 0");
list.add("element 1");
list.add("element 2");

list.remove(0);
```

4.7. List Size

You can obtain the number of elements in the `List` by calling the `size()` method. Here is an example:

```
List<String> list = new ArrayList<>();

list.add("object 1");
list.add("object 2");

int size = list.size();
```

4.8. Sublist of List

The Java `List` interface has a method called `subList()` which can create a new `List` with a subset of the elements from the original `List`.

The `subList()` method takes 2 parameters: A start index and and end index. The start index is the index of the first element from the original `List` to include in the sublist. The end index is the last index of the sublist, but the element at the last index is not included in the sublist. This is similar to how the Java String substring method works.

Here is a Java example of creating a sublist of elements from another `List` using the `subList()` method:

```
List<String> list     = new ArrayList<>();

list.add("element 1");
list.add("element 2");
list.add("element 3");
list.add("element 4");

List<String> sublist = list.subList(1, 3);
```

4.9. Convert List to Array

You can convert a Java `List` to a **Java Array** using the `List` `toArray()` method. Here is an example of converting a Java `List` to a Java array:

```
List<String> list     = new ArrayList<>();

list.add("element 1");
list.add("element 2");
list.add("element 3");
list.add("element 3");

Object[] objects = list.toArray();
```

It is also possible to convert a `List` to an array of a specific type. Here is an example of converting a Java `List` to an array of a specific type:

```
List<String> list     = new ArrayList<>();

list.add("element 1");
list.add("element 2");
list.add("element 3");
list.add("element 3");

String[] objects1 = list.toArray(new String[0]);
```

Note that even if we pass a String array of size 0 to the `toArray()`, the array returned will have all the elements in the `List` in it. It will have the same number of elements as the `List`.

4.10. Convert Array to List

It is also possible to convert a Java `List` to an array. Here is an example of converting a Java array to a `List`:

```
String[] values = new String[]{ "one", "two", "three" };

List<String> list = (List<String>) Arrays.asList(values);
```

It is the `Arrays.asList()` method that converts the array to a `List`.

4.11. Sort List

You can sort a Java `List` using the `Collections` `sort()` method.

4.12. Iterate List

You can iterate a Java `List` in several different ways. The three most common ways are:

- Using an `Iterator`
- Using a for-each loop
- Using a for loop
- Using the Java Stream API

I will explain each of these methods of iterating a Java `List` in the following sections.

4.12.1. Iterate List Using Iterator

The first way to iterate a `List` is to use a Java Iterator. Here is an example of iterating a `List` with an `Iterator`:

```
List<String> list = new ArrayList<>();

list.add("first");
list.add("second");
list.add("third");

Iterator<String> iterator = list.iterator();
while(iterator.hasNext()) {
    String next = iterator.next();
}
```

You obtain an `Iterator` by calling the `iterator()` method of the `List` interface.

Once you have obtained an `Iterator` you can keep calling its `hasNext()` method until it returns `false`. Calling `hasNext()` is done inside a `while` loop as you can see.

Inside the `while` loop you call the `Iterator` `next()` method of the `Iterator` interface to obtain the next element pointed to by the `Iterator`.

If the `List` is typed using Java Generics you can save some object casting inside the `while` loop. Here is an example:

```
List<String> list = new ArrayList<>();

list.add("first");
list.add("second");
list.add("third");

Iterator<String> iterator = list.iterator();
while(iterator.hasNext()){
   String obj = iterator.next();
}
```

4.12.2. Iterate List Using For-Each Loop

The second way to iterate a `List` is to use the `for` loop added in Java 5 (also called a "for each" loop). Here is an example of iterating a `List` using the `for` loop:

```
List list = new ArrayList();

list.add("first");
list.add("second");
list.add("third");

for(Object element : list) {
   System.out.println(element);
}
```

The `for` loop is executed once per element in the `List`. Inside the `for` loop each element is in turn bound to the `obj` variable.

If the list is typed (a *generic* `List`) you can change the type of the variable inside the `for` loop. Here is typed `List` iteration example:

```
List<String> list = new ArrayList<String>();

//add elements to list

for(String element : list) {
   System.out.println(element);
}
```

Notice how the `List` is typed to `String`. Therefore you can set the type of the variable inside the `for` loop to `String`.

4.12.3. Iterate List Using For Loop

The third way to iterate a `List` is to use a standard `for` loop like this:

```
List list = new ArrayList();

list.add("first");
list.add("second");
list.add("third");

for(int i=0; i < list.size(); i++) {
   Object element = list.get(i);
}
```

The `for` loop creates an `int` variable and initializes it to 0. Then it loops as long as the int variable `i` is less than the size of the list. For each iteration the variable `i` is incremented.

Inside the `for` loop the example accesses the elements in the `List` via its `get()` method, passing the incrementing variable `i` as parameter.

Again, if the `List` is typed using Java Generics to e.g. to a `String`, then you can use the generic type of the `List` as type for the local variable that is assigned each element in the `List` during iteration. An example will make this more clear:

```
List<String> list = new ArrayList<String>();

list.add("first");
list.add("second");
list.add("third");

for(int i=0; i < list.size(); i++) {
   String element = list.get(i);
}
```

Notice the type of the local variable inside the `for` loop is now `String`. Because the `List` is generically typed to `String`, it can only contain `String` objects. Hence, the compiler knows that only a `String` can be returned from the `get()` method. Therefore you do not need to cast the element returned by `get()` to `String`.

4.12.4. Iterate List Using Java Stream API

The fourth way to iterate a Java `List` is via the Java Stream API. To iterate a Java `List` you must first obtain a `Stream` from the `List`. Obtaining a `Stream` from a `List` in Java is done by calling the `List` `stream()` method. Here is an example of obtaining a Java `Stream` from a Java `List`:

```
List<String> stringList = new ArrayList<String>();

stringList.add("abc");
stringList.add("def");

Stream<String> stream = stringList.stream();
```

It is the last line of this example that calls the `List` `stream()` method to obtain the `Stream` representing the elements in the `List`.

Once you have obtained a `Stream` from a `List` you can iterate the `Stream` by calling its `forEach()` method. Here is an example of iterating the elements of a `List` using the `Stream` `forEach()` method:

```
List<String> stringList = new ArrayList<String>();

stringList.add("one");
stringList.add("two");
stringList.add("three");

Stream<String> stream = stringList.stream();
stream
   .forEach( element -> { System.out.println(element); });
```

Calling the `forEach()` method will make the `Stream` iterate all the element of the `Stream` internally, and call the `Consumer` passed as parameter to the `forEach()` method for each element in the `Stream`.

5. Java Set

The *Java Set* interface, `java.util.Set`, represents a collection of objects where each object in the Java *Set* is unique. In other words, the same object cannot occur more than once in a Java *Set*. The Java `Set` interface is a standard Java interface, and it is a subtype of the Java Collection interface, meaning `Set` inherits from `Collection`.

You can add any Java object to a Java `Set`. If the `Set` is not typed, using Java Generics, then you can even mix objects of different types (classes) in the same `Set`. Mixing objects of different types in the same `Set` is not often done in reality, however.

5.1. Java Set vs. List

The Java `Set` and Java List interfaces are quite similar to each other. Bot interfaces represents a collection of elements. However, there are some significant differences. These differences are reflected in the methods the `Set` and `List` interfaces contain.

The first difference between the Java `Set` and `List` interface is, that the same element cannot occur more than once in a Java `Set`. This is different from a Java `List` where each element can occur more than once.

The second difference between a Java `Set` and Java `List` interfaces is, that the elements in a `Set` has no guaranteed internal order. The elements in a `List` has an internal order, and the elements can be iterated in that order.

5.2. Java Set Example

Here is first a simple Java `Set` example to give you a feel for how sets work:

```
package com.jenkov.collections;

import java.util.HashSet;

public class SetExample {

    public static void main(String[] args) {

        Set setA = new HashSet();

        setA.add(element);

        System.out.println( setA.contains(element) );
    }
}
```

This example creates a `HashSet` which is one of the classes in the Java APIs that implement the `Set` interface. Then it adds a string object to the set, and finally it checks if the set contains the element just added.

5.3. Set Implementations

Being a `Collection` subtype all methods in the `Collection` interface are also available in the `Set` interface.

Since `Set` is an interface you need to instantiate a concrete implementation of the interface in order to use it. You can choose between the following `Set` implementations in the Java Collections API:

- java.util.EnumSet
- java.util.HashSet
- java.util.LinkedHashSet
- java.util.TreeSet

Each of these `Set` implementations behaves a little differently with respect to the order of the elements when iterating the `Set`, and the time (big O notation) it takes to insert and access elements in the sets.

`HashSet` is backed by a `HashMap`. It makes no guarantees about the sequence of the elements when you iterate them.

`LinkedHashSet` differs from `HashSet` by guaranteeing that the order of the elements during iteration is the same as the order they were inserted into the `LinkedHashSet`. Reinserting an element that is already in the `LinkedHashSet` does not change this order.

`TreeSet` also guarantees the order of the elements when iterated, but the order is the sorting order of the elements. In other words, the order in which the elements whould be sorted if you used a `Collections.sort()` on a `List` or array containing these elements. This order is determined either by their natural order (if they implement `Comparable`), or by a specific `Comparator` implementation.

There are also `Set` implementations in the `java.util.concurrent` package, but I will leave the concurrency utilities out of this tutorial.

5.4. Create a Set

Here are a few examples of how to create a `Set` instance:

```
package com.test.collections;

import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.Set;
import java.util.TreeSet;

public class SetExample {

    public static void main(String[] args) {

        Set setA = new HashSet();
        Set setB = new LinkedHashSet();
        Set setC = new TreeSet();

    }
}
```

5.5. Generic Sets

By default you can put any `Object` into a `Set`, but from Java 5, Java Generics makes it possible to limit the types of object you can insert into a `Set`. Here is an example:

```
Set<MyObject> set = new HashSet<MyObject>();
```

This `Set` can now only have `MyObject` instances inserted into it. You can then access and iterate its elements without casting them. Here is how it looks:

```
for(MyObject anObject : set){
  //do someting to anObject...
}
```

It is considered good practice to alway specify a generic type for a Java Set whenever you know it. Most of the examples in this tutorial uses generic types.

For more information about Java Generics, see the **Java Generics Tutorial**.

5.6. Add Element to Set

To add elements to a `Set` you call its `add()` method. This method is inherited from the `Collection` interface. Here are a few examples:

```
Set<String> setA = new HashSet<>();

setA.add("element 1");
setA.add("element 2");
setA.add("element 3");
```

The three `add()` calls add a `String` instance to the set.

5.7. Iterate Set Elements

There are two ways to iterate the elements of a Java `Set`:

- Using an `Iterator` obtained from the `Set`.
- Using the for-each loop.

Both of these options are covered in the following sections.

When iterating the elements in the `Set` the order of the elements depends on what `Set` implementation you use, as mentioned earlier.

5.8. Iterate Set Using Iterator

To iterate the elements of a `Set` using an Java Iterator, you must first obtain an `Iterator` from the `Set`. You obtain an `Iterator` from a `Set` by calling the `iterator()` method. Here is an example of obtaining an `Iterator` from a `Set`:

```
Set<String> setA = new HashSet<>();

setA.add("element 1");
setA.add("element 2");
setA.add("element 3");

Iterator<String> iterator = set.iterator();

while(iterator.hasNext(){
  String element = iterator.next();
}
```

5.9. Iterate Set Using For-Each Loop

The second way to iterate the elements of a `Set` is by using a for-each loop. Here is how iterating the elements of a `Set` using a for-each loop looks:

```
Set set = new HashSet();

for(Object object : set) {
   String element = (String) object;
}
```

The `Set` interface implements the Java Iterable interface. That is why you can iterate the elements of a `Set` using the for-each loop.

If the set has a generic type specified, you can use that type as the variable type inside the for-each loop. Here is how that looks:

```
Set<String> set = new HashSet<>();

for(String str : set) {
   System.out.println(str);
}
```

This is the preferred way to use the for-each loop - in combination with the generic type specified for the Collection the for-each loop is used on.

5.10. Iterate Set Using the Java Stream API

The third way to iterate a Java `Set` is via the **Java Stream API** . To itereate a Java `Set` using the Java Stream API you must create a `Stream` from the `Set` . Here is an example of creating a Java `Stream` from a `Set` and iterate the `Stream` :

```
Set<String> set = new HashSet<>();

set.add("one");
set.add("two");
set.add("three");

Stream<String> stream = set.stream();

stream.forEach((element) -> { System.out.println(element); });
```

You can read more about what options you have available in the Java Stream API in my **Java Stream API tutorial**.

5.11. Remove Elements From Set

You remove elements from a Java `Set` by calling the `remove(Object o)` method. Here is an example of removing an element from a Java `Set`:

```
set.remove("object-to-remove");
```

There is no way to remove an object based on index in a `Set`, since the order of the elements depends on the `Set` implementation.

5.12. Set Size

You can check the *size* of a Java `Set` using the `size()` method. The size of a `Set` is the number of elements contained in the `Set`. Here is an example of reading the size of a Java `Set`:

```
Set<String> set = new HashSet<>();

set.add("123");
set.add("456");
set.add("789");

int size = set.size();
```

After executing this Java code the `size` variable will have the value 3, because the `Set` created in the example had 3 elements added to it.

5.13. Check if Set is Empty

You can check if a Java `Set` is empty, meaning it contains no elements, by calling the `isEmpty()` method on the `Set`. Here is an example of checking if a Java `Set` is empty:

```
Set<String> set = new HashSet<>();

boolean isEmpty = set.isEmpty();
```

After running this Java code the `isEmpty` variable will contain the value `true`, because the `Set` is empty (has no elements in it).

You can also check if a `Set` is empty by comparing the value returned by the `size()` method with 0. Here is an example that shows how:

```
Set<String> set = new HashSet<>();

boolean isEmpty = (set.size() == 0);
```

After running this Java code the `isEmpty` variable will contain the value `true`, because the `Set` `size()` method returns 0 - because the `Set` in the example contains no elements.

5.14. Check if Set Contains Element

You can check if a Java `Set` contains a given element (object) by calling the `contains()` method. Here is an example of checking if a Java `Set` contains a given element:

```
Set<String> set = new HashSet<>();

set.add("123");
set.add("456");

boolean contains123 = set.contains("123");
```

After running this Java code the `contains123` variable will contain the value `true` because the `Set` actually contains the String `123`.

To determine if the `Set` contains the element, the `Set` will internally iterate its elements and compare each element to the object passed as parameter. The comparison uses the **Java equals** method of the element to check if the element is equal to the parameter.

Since it is possible to add `null` values to a `Set`, it is also possible to check if the `Set` contains a `null` value. Here is how you check if a `Set` contains a `null` value:

```
set.add(null);

containsElement = set.contains(null);

System.out.println(containsElement);
```

Obviously, if the input parameter to `contains()` is `null`, the `contains()` method will not use the `equals()` method to compare against each element, but rather use the `==` operator.

5.15. Convert Java Set to List

You can convert a Java `Set` to a Java `List` by creating a `List` and calling its `addAll()` method, passing the `Set` as parameter to the `addAll()` method. Here is an example of converting a Java `Set` to a `List`:

```
Set<String> set = new HashSet<>();
set.add("123");
set.add("456");

List<String> list = new ArrayList<>();
list.addAll(set);
```

After running this Java example, the `List` will contain the String elements `123` and `456` - since these were all the elements present in the `Set` when `List` `addAll(set)` was called.

6. Comparator

6.1. Sort List of Comparable Objects

If the `List` contains objects that implement the `Comparable` interface (`java.lang.Comparable`), then the objects can compare themselves to each other. In that case you can sort the `List` like this:

```
List<String> list = new ArrayList<>();

list.add("c");
list.add("b");
list.add("a");

Collections.sort(list);
```

The Java `String` class implements the `Comparable` interface, you can sort them in their natural order, using the `Collections` `sort()` method.

6.2. Sort List Using Comparator

If the objects in the Java `List` do not implement the `Comparable` interface, or if you want to sort the objects in another order than their `compare()` implementation, then you need to use a `Comparator` implementation (`java.util.Comparator`). Here is an example of sorting a list of `Car` objects using a `Comparator`. Here is first the `Car` class:

```
public class Car{
   public String brand;
   public String numberPlate;
   public int noOfDoors;

   public Car(String brand, String numberPlate, int noOfDoors) {
      this.brand = brand;
      this.numberPlate = numberPlate;
      this.noOfDoors = noOfDoors;
   }
}
```

Here is the code that sorts a Java `List` of the above `Car` objects:

```
List<Car> list = new ArrayList<>();

list.add(new Car("Volvo V40" , "XYZ 201845", 5));
list.add(new Car("Citroen C1", "ABC 164521", 4));
list.add(new Car("Dodge Ram" , "KLM 845990", 2));

Comparator<Car> carBrandComparator = new Comparator<Car>() {
   @Override
   public int compare(Car car1, Car car2) {
      return car1.brand.compareTo(car2.brand);
   }
};

Collections.sort(list, carBrandComparator);
```

Notice the `Comparator` implementation in the example above. This implementation only compares the `brand` field of the `Car` objects. It is possible to create another `Comparator` implementation which compares the number plates, or even the number of doors in the cars.

7. Java hashCode() and equals()

The methods `hashCode()` and `equals()` play a distinct role in the objects you insert into Java collections. The specific contract rules of these two methods are best described in the JavaDoc. Here I will just tell you what role they play. What they are used for, so you know why their implementations are important.

7.1. equals()

`equals()` is used in most collections to determine if a collection contains a given element. For instance:

```
List list = new ArrayList();
list.add("123");

boolean contains123 = list.contains("123");
```

The `ArrayList` iterates all its elements and execute `"123".equals(element)` to determine if the element is equal to the parameter object “123”. It is the `String.equals()` implementation that determines if two strings are equal.

The `equals()` method is also used when removing elements. For instance:

```
List list = new ArrayList();
list.add("123");

boolean removed = list.remove("123");
```

The `ArrayList` again iterates all its elements and execute `"123".equals(element)` to determine if the element is equal to the parameter object “123”. The first element it finds that is equal to the given parameter “123” is removed.

7.2. objects equal

Here is a simple example of such an `Employee` class:

```
public class Employee {
  protected long  employeeId;
  protected String firstName;
  protected String lastName;
}
```

You could decide that two `Employee` objects are equal to each other if just their `employeeId`'s are equal. Or, you could decide that all fields must be equal - both `employeeId`, `firstName` and `lastName`. Here are two example implementation of `equals()` matching these criterias:

Java hashCode() and equals()


```
public class Employee {
  ...
  public boolean equals(Object o){
    if(o == null)              return false;
    if(!(o instanceof) Employee) return false;

    Employee other = (Employee) o;
    return this.employeeId == other.employeeId;
  }
}
public class Employee {
  ...
  public boolean equals(Object o){
    if(o == null)              return false;
    if(!(o instanceof) Employee) return false;

    Employee other = (Employee) o;
    if(this.employeeId != other.employeeId)      return false;
    if(! this.firstName.equals(other.firstName)) return false;
    if(! this.lastName.equals(other.lastName))   return false;

    return true;
  }
}
```

7.3. hashCode()

a combination of the `hashCode()` and `equals()` methods are used when storing and when looking up objects in a hashtable.

Here are two rules that are good to know about implementing the `hashCode()` method in your own classes, if the hashtables in the Java Collections API are to work correctly:

1. If object1 and object2 are equal according to their `equals()` method, they must also have the same hash code.
2. If object1 and object2 have the same hash code, they do NOT have to be equal too.

In shorter words:

1. If equal, then same hash codes too.
2. Same hash codes no guarantee of being equal.

Here are two example implementation of the `hashCode()` method matching the `equals()` methods shown earlier:

```
public class Employee {
  protected long  employeeId;
  protected String firstName;
  protected String lastName;

  public int hashCode(){
    return (int) employeeId;
  }
}
public class Employee {
    protected long  employeeId;
    protected String firstName;
    protected String lastName;

  public int hashCode(){
    return (int) employeeId *
            firstName.hashCode() *
            lastName.hashCode();
  }
}
```

Notice, that if two `Employee` objects are equal, they will also have the same hash code. But, as is especially easy to see in the first example, two `Employee` objects can be not equal, and still have the same hash code.

Thank you

全国统一咨询热线：400-690-6115

北京|上海|广州|深圳|天津|成都|重庆|武汉|济南|青岛|杭州|西安