softech solutions

CORE JAVA - COLLECTIONS

Lecture Notes

A COMPLETE GUIDE FOR BEGINNERS

Softech Solutions Inc.

www.softechsolutionsgroup.com

saney.alam@softechsolutionsgroup.com

Core Java - Collections

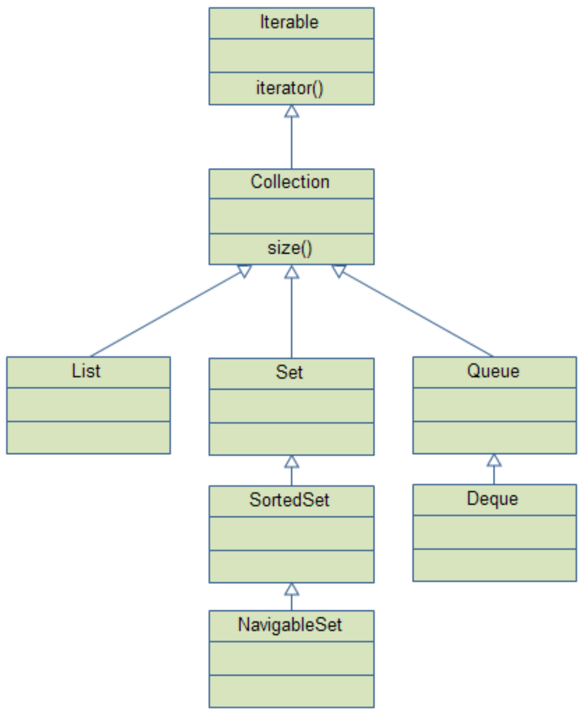
Softech Solution Inc.
www.softechsolutionsgroup.com
info@softechsolutionsgroup.com

OVERVIEW

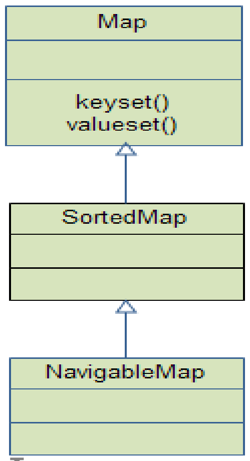
In order to understand and use the Java Collections API effectively it is useful to have an overview of the interfaces it contains. So, that is what I will provide here.

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:



ITERABLE

The Iterable interface (*java.lang.Iterable*) is one of the root interfaces of the Java collection classes. The *Collection* interface extends *Iterable*, so all subtypes of *Collection* also implement the Iterableinterface.

A class that implements the Iterable can be used with the new for-loop. Here is such an example:

```
List list = new ArrayList();

for(Object o : list){
    //do something o;
}
```

The Iterable interface has only one method:

```
public interface Iterable<T> {
  public Iterator<T> iterator();
}
```

COLLECTION

The Collection interface (*java.util.Collection*) is one of the root interfaces of the Java collection classes. Though you do not instantiate a Collection directly, but rather a subtype of Collection, you may often treat these subtypes uniformly as a Collection.

COLLECTION SUBTYPES

The following interfaces (collection types) extend the Collection interface:

- List
- Set
- SortedSet
- NavigableSet
- Queue
- Deque

Java does not come with a usable implementation of the Collection interface, so you will have to use one of the listed subtypes. The Collection interface just defines a set of methods (behavior) that each of these Collection subtypes share. This makes it possible ignore what specific type of Collection you are using, and just treat it as a Collection. This is standard inheritance, so there is nothing magical about, but it can still be a nice feature from time to time. Later sections in this text will describe the most used of these common operations.

Here is a method that operates on a Collection:

```
public class MyCollectionUtil{

  public static void doSomething(Collection collection) {

    Iterator iterator = collection.iterator();
    while(iterator.hasNext()){
      Object object = iterator.next();

      //do something to object here...
    }
  }
}
```

And here are a few ways to call this method with different Collection subtypes:

```
Set  set  = new HashSet();
List list = new ArrayList();

MyCollectionUtil.doSomething(set);
MyCollectionUtil.doSomething(list);
```

ADDING AND REMOVING ELEMENTS

Regardless of what Collection subtype you are using there are a few standard methods to add and remove elements from a Collection. Adding and removing single elements is done like this:

```
String     anElement  = "an element";
Collection collection = new HashSet();

boolean didCollectionChange = collection.add(anElement);
boolean wasElementRemoved   = collection.remove(anElement);
```

`add()` adds the given element to the collection, and returns true if the Collection changed as a result of calling the `add()` method. A Set for instance may not have changed. If the Set already contained that element, it is not added again. On the other hand, if you called `add()` on a List and the List already contained that element, the element would then exist twice in the List.

`remove()` removes the given element and returns true if the removed element was present in the Collection, and was removed. If the element was not present, the `remove()` method returns false.

You can also add and remove collections of objects. Here are a few examples:

```
Set  aSet  = ... // get Set  with elements from somewhere
List aList = ... // get List with elements from somewhere

Collection collection = new HashSet();

collection.addAll(aSet);    //returns boolean too, but ignored here.
collection.addAll(aList);   //returns boolean too, but ignored here.

collection.removeAll(aList);   //returns boolean too...
collection.retainAll(aSet);    //returns boolean too...
```

`addAll()` adds all elements found in the Collection passed as parameter to the method. The Collection object itself is not added. Only its elements. If you had called `add()` with the Collection as parameter instead, the Collection object itself would have been added, not its elements.

Exactly how that `addAll()` method behaves depends on the Collection subtype. Some Collection subtypes allow the same element to be added more than once, and others don't.

`removeAll()` removes all elements found the Collection passed as parameter to the method. If the Collection parameter contains any elements not found the target collection, these are just ignored.

`retainAll()` does the opposite of `removeAll()`. Instead of removing all the elements found in the parameter Collection, it keeps all these elements, and removes all other elements. Keep in mind, that only if the elements were already contained in the target collection, are they retained. Any new elements found in the parameter Collection which are not in the target collection, are not automatically added. They are just ignored.

Let's see an example using some pseudo-language:

```
Collection colA = [A,B,C]
Collection colB = [1,2,3]

Collection target = [];

target.addAll(colA);     //target now contains [A,B,C]
target.addAll(colB);     //target now contains [A,B,C,1,2,3]

target.retainAll(colB);  //target now contains [1,2,3]

target.removeAll(colA);  //nothing happens - already removed
target.removeAll(colB);  //target is now empty.
```

CHECKING IF A COLLECTION CONTAINS A CERTAIN ELEMENT

The Collection interface has two methods to check if a Collection contains one or more certain elements. These are the `contains()` and `containsAll()` methods. They are illustrated here:

```
Collection collection   = new HashSet();
boolean containsElement = collection.contains("an element");

Collection elements     = new HashSet();
boolean containsAll     = collection.containsAll(elements);
```

`contains()` returns true if the collection contains the element, and false if not.

`containsAll()` returns true if the collection contains all the elements in the parameter collection, and false if not.

COLLECTION SIZE

You can check the size of a collection using the size() method. By "size" is meant the number of elements in the collection. Here is an example:

```
int numberOfElements = collection.size();
```

ITERATING A COLLECTION

You can iterate all elements of a collection. This is done by obtaining an Iterator from the collection, and iterate through that. Here is how it looks:

```
Collection collection = new HashSet();
//... add elements to the collection

Iterator iterator = collection.iterator();
while(iterator.hasNext()){
    Object object = iterator.next();
    //do something to object;
}
```

You can also use the new for-loop:

```
Collection collection = new HashSet();
//... add elements to the collection

for(Object object : collection) {
  //do something to object;
}
```

LIST

The *`java.util.List`* interface is a subtype of the *`java.util.Collection`* interface. It represents an ordered list of objects, meaning you can access the elements of a *`List`* in a specific order, and by an index too. You can also add the same element more than once to a *`List`*.

LIST IMPLEMENTATIONS

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

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

- java.util.ArrayList
- java.util.LinkedList
- java.util.Vector
- java.util.Stack

There are also `List` implementations in the `java.util.concurrent` package, but lets leave the concurrency utilities out of this section.

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();
```

ADDING AND ACCESSING ELEMENTS

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

```
List listA = new ArrayList();

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

listA.add(0, "element 0");
```

The first three `add()` calls add a String instance to the end of the list. The last `add()` call adds a String at index 0, meaning at the beginning of the list.

The order in which the elements are added to the List is stored, so you can access the elements in the same order. You can do so using either the `get(int index)` method, or via the `Iterator` returned by the `iterator()` method. Here is how:

```
List 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);
//access via Iterator
Iterator iterator = listA.iterator();
while(iterator.hasNext(){
  String element = (String) iterator.next();
}


//access via new for-loop
for(Object object : listA) {
    String element = (String) object;
}
```

When iterating the list via its `Iterator` or via the for-loop (which also uses the Iterator behind the scene), the elements are iterated in the same sequence they are stored in the list.

REMOVING ELEMENTS

You can remove elements in two ways:

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.

`remove(int index)` 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.

CLEARING A LIST

The Java List interface contains a *`clear()`* method which removes all elements from the list when called. Here is simple example of clearing a *`List`* with *`clear():`*

```
List list = new ArrayList();

list.add("object 1");
list.add("object 2");
//etc.
```

```
list.clear();
```

LIST SIZE

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

```
List list = new ArrayList();

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

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

GENERIC LISTS

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

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

This `List` 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:

```
MyObject myObject = list.get(0);

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

ITERATING A LIST

You can iterate a Java List in several different ways. We will cover the three most common mechanisms here.

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

```
List list = new ArrayList();

//add elements to list

Iterator iterator = list.iterator();
while(iterator.hasNext()) {
    Object 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 `next()` method of the Iterator interface to obtain the next element pointed to by the `Iterator`.

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

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

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

Another 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();

//add elements to list

for(Object obj : list) {

}
```

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 obj : list) {

}
```

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

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

```
List list = new ArrayList();

//add elements to list

for(int i=0; i < list.size(); i++) {
    Object obj = 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.

SET

The `java.util.Set` interface is a subtype of the `java.util.Collection` interface. It represents set of objects, meaning each element can only exist once in a Set.

JAVA SET EXAMPLE

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

```
Set setA = new HashSet();
```

```
String element = "element 1";

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.

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 should 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, let's leave the concurrency utilities out of this section.

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

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

ADDING AND ACCESSING ELEMENTS

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 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.

When iterating the elements in the Set the order of the elements depends on what `Set` implementation you use, as mentioned earlier. Here is an iteration example:

```
Set setA = new HashSet();

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

//access via Iterator
Iterator iterator = setA.iterator();
while(iterator.hasNext(){
  String element = (String) iterator.next();
}


//access via new for-loop
for(Object object : setA) {
    String element = (String) object;
}
```

REMOVING ELEMENTS

You remove elements by calling the `remove(Object o)` method. 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.

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...
}
```

MAP

The `java.util.Map` interface represents a mapping between a key and a value. The `Map` interface is not a subtype of the `Collection` interface. Therefore it behaves a bit different from the rest of the collection types.

MAP IMPLEMENTATIONS

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

- java.util.HashMap
- java.util.Hashtable
- java.util.EnumMap

- java.util.IdentityHashMap
- java.util.LinkedHashMap
- java.util.Properties
- java.util.TreeMap
- java.util.WeakHashMap

In experience, the most commonly used `Map` implementations are `HashMap` and `TreeMap`.

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

`HashMap` maps a key and a value. It does not guarantee any order of the elements stored internally in the map.

`TreeMap` also maps a key and a value. Furthermore it guarantees the order in which keys or values are iterated - which is the sort order of the keys or values. Check out the JavaDoc for more details.

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

```
Map mapA = new HashMap();
Map mapB = new TreeMap();
```

ADDING AND ACCESSING ELEMENTS

To add elements to a `Map` you call its `put()` method. Here are a few examples:

```
Map mapA = new HashMap();

mapA.put("key1", "element 1");
mapA.put("key2", "element 2");
mapA.put("key3", "element 3");
```

The three `put()` calls maps a string value to a string key. You can then obtain the value using the key. To do that you use the `get()` method like this:

```
String element1 = (String) mapA.get("key1");
```

You can iterate either the keys or the values of a `Map`. Here is how you do that:

```
// key iterator
Iterator iterator = mapA.keySet().iterator();

// value iterator
Iterator iterator = mapA.values();
```

Most often you iterate the keys of the `Map` and then get the corresponding values during the iteration. Here is how it looks:

```
Iterator iterator = mapA.keySet().iterator();
while(iterator.hasNext(){
  Object key   = iterator.next();
  Object value = mapA.get(key);
}

//access via new for-loop
for(Object key : mapA.keySet()) {
    Object value = mapA.get(key);
}
```

REMOVING ELEMENTS

You remove elements by calling the `remove(Object key)` method. You thus remove the (key, value) pair matching the key.

GENERIC MAPS

By default you can put any `Object` into a `Map`, but from Java 5, Java Generics makes it possible to limit the types of object you can use for both keys and values in a `Map`. Here is an example:

```
Map<String, MyObject> map = new HashMap<String, MyObject>();
```

This `Map` can now only accept String objects for keys, and `MyObject` instances for values. You can then access and iterate keys and values without casting them. Here is how it looks:

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

for(String key : map.keySet()){
   MyObject value = map.get(key);
   //do something to value
}
```

SORTING

You can sort `List` collections using the `java.util.Collections.sort()` method. You can sort these two types of List's.

1. List
2. LinkedList

SORTING OBJECTS BY THEIR NATURAL ORDER

To sort a `List` you do this:

```
List list = new ArrayList();

//add elements to the list

Collections.sort(list);
```

When sorting a list like this the elements are ordered according to their "natural order". For objects to have a natural order they must implement the interface `java.lang.Comparable`. In other words, the objects must be comparable to determine their order. Here is how the `Comparable` interface looks:

```
public interface Comparable<T> {
  int compareTo(T o);
}
```

The `compareTo()` method should compare this object to another object, return an `int` value. Here are the rules for that `int` value:

Return a negative value if this object is smaller than the other object

Return 0 (zero) if this object is equal to the other object.

Return a positive value if this object is larger than the other object.

There are a few more specific rules to obey in the implementation, but the above is the primary requirements. Check out the JavaDoc for the details.

Let's say you are sorting a List of String element. To sort them, each string is compared to the others according to some sorting algorithm (not interesting here). Each string compares itself to another string by alphabetic comparison. So, if a string is less than another string by alphabetic comparison it will return a negative number from the `compareTo()` method.

When you implement the `compareTo()` method in your own classes you will have to decide how these objects should be compared to each other. For instance, Employee objects can be compared by their first name, last name, salary, start year or whatever else you think makes sense.

SORTING OBJECTS USING A COMPARATOR

Sometimes you may want to sort a list according to another order than their natural order. Perhaps the objects you are sorting do not even have a natural order. In that case you can use a `Comparator` instead. Here is how you sort a list using a `Comparator`:

```
List list = new ArrayList();

//add elements to the list

Comparator comparator = new SomeComparator();

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

Notice how the `Collections.sort()` method now takes a `java.util.Comparator` as parameter in addition to the `List`. This Comparator compares the elements in the list two by two. Here is how the `Comparator` interface looks:

```
public interface Comparator<T> {
    int compare(T object1, T object2);
}
```

The `compare()` method compares two objects to each other and should:

Return a negative value if object1 is smaller than object2

Return 0 (zero) if objec1 is equal to object2.

Return a positive value if object1 is larger than object2.

There are a few more requirements to the implementation of the `compare()` method, but these are the primary requirements. Check out the JavaDoc for more specific details.

Here is an example `Comparator` that compares two fictive Employee objects:

```
public class MyComparator<Employee> implements Comparator<Employee> {

    public int compare(Employee emp1, Employee emp2){
       if(emp1.getSalary() <  emp2.getSalary()) return -1;
       if(emp1.getSalary() == emp2.getSalary()) return 0;
       return 1;
    }
}
```

A shorter way to write the comparison would be like this:

```
public class MyComparator<Employee> implements Comparator<Employee> {

    public int compare(Employee emp1, Employee emp2){
       return emp1.getSalary() - emp2.getSalary();
    }
}
```

By subtracting one salary from the other, the resulting value is automatically negative, 0 or positive. Smart, right?

If you want to compare objects by more than one factor, start by comparing by the first factor (e.g. first name). Then, if the first factors are equal, compare by the second factor (e.g. last name, or salary) etc.