**9.3 Collection Classes :-**From Wikibooks, open books for an open world

|  |
| --- |
| Collections are a group of objects bound together by a common characteristic. Java has various built-in classes to support the collection of objects, either of the same type or general. The most basic construct to work with is the *array* of which you will come to know of in a [section](http://en.wikibooks.org/wiki/Java_Programming/Arrays) later in this chapter.  java collections framework is a set of classes and interfaces that are used to implement reusable collection data structures.  Code given below shows that how to use various methods of Collections class. For example- add element, copy elements, reverse data structure etc. |

The most basic collection interface is called Collection. This interface gives the user a generic usage of a collection.

[**import**](http://en.wikibooks.org/wiki/Java_Programming/Keywords/import) java.util.Collection; // Interface

[**import**](http://en.wikibooks.org/wiki/Java_Programming/Keywords/import) java.util.ArrayList; // Implementation

[**import**](http://en.wikibooks.org/wiki/Java_Programming/Keywords/import) java.util.HashSet; // Implementation

...

Collection coll1 = [**new**](http://en.wikibooks.org/wiki/Java_Programming/Keywords/new) ArrayList();

Collection coll2 = [**new**](http://en.wikibooks.org/wiki/Java_Programming/Keywords/new) HashSet();

...

< Use coll1 & coll2 >

In the above there are two collections. The usage of the collections are the same, the implementations are different. If the existing collection implementations do not meet your needs, you can write your version of the implementation. Your version of the implementation just needs to implement the same java.util.Collection interface, then you can switch to using your implementation and the code that is using the collection does not need to be changed.

[**import**](http://en.wikibooks.org/wiki/Java_Programming/Keywords/import) java.util.Collection;

[**import**](http://en.wikibooks.org/wiki/Java_Programming/Keywords/import) com.yourcomp.util.YourCollectionImpl;

...

Collection coll1 = new YourCollectionImpl();

Collection coll2 = new YourCollectionImpl();

...

< Use coll1 & coll2 >

The Java JDK collection implementations are quite powerful and good, so it is unlikely that you will need to write your own.

All collections contain object references. Because of that, if the object is changed after it was put in the collection, the object that is 'in' the collection also 'changes'. The object is not really in the collection, only the object reference is. It is not guaranteed that the objects 'inside' the collections won't change. This is an issue only if you put an actively used object in the collection. In that case when you are adding an object that could change any time you need to make a copy or clone of the object. A new object will be created and its reference will be put in the collection. In that case there will be no object references outside of the collection, so the objects 'inside' the collection can only be changed if we take out an object reference from the collection.

Aside from the java.util.Collection interface, the Java JDK has the java.util.Map interface as well. This defines key value mappings. Implementations of the Map interface do not contain collections of objects. Instead they contain collections of key->value mappings.

[**import**](http://en.wikibooks.org/wiki/Java_Programming/Keywords/import) java.util.Map;

[**import**](http://en.wikibooks.org/wiki/Java_Programming/Keywords/import) java.util.Hashtable;

...

Map map = new Hashtable();

...

map.put( key, value );

All collections need to have the same basic operations. Those are:

* Adding element(s) to the collection
* Removing element(s) from the collection
* Obtaining the number of elements in the collection
* Listing the contents of the collection, (Iterating through the collection)

Before selecting a particular collection implementation, ask the following question:

* Can my collection contain the same elements, i.e. are duplicates allowed?
* Can my collection contain the [**null**](http://en.wikibooks.org/wiki/Java_Programming/Literals/null) element?
* Should the collection maintain the order of the elements? Is the order important in any way?
* How do you want to access an element? By index, key or just with an iterator?
* Does the collection need to be synchronized?
* From a performance perspective, which one needs to be faster, updates or reads?
* From a usage perspective, which operation will be more frequent, updates or reads?

Once you know your needs, you can select an exsisting implementation. But first decide if you need a 'Collection', or a 'Map'.

**[**[**edit**](http://en.wikibooks.org/w/index.php?title=Java_Programming/Collection_Classes&action=edit&section=2)**] Generics**

Since JDK version 1.5 an enhancement to the type system of the Java language has been added. It is called *Generics*. Most often *Generics* will be used with the collection classes.

parameterized type <E>

All collection implementations since 1.5 now have one 'parameterized type <E>' added. The 'E' refers to an *Element* type. When a collection is created the actual 'Element type' will replace the E.

Objects put into a collection are upcasted to [Object](http://en.wikibooks.org/wiki/Java_Programming/API/java.lang.Object) class. It means that you need to cast the object reference back when you get an element out from the collection. It also means that **you need to know** the type of the object when you taking it out. For this reason we usually add objects of the same type to a collection. If a collection contains different types of objects, we will have difficulty finding out the type of the objects obtained from a collection at run time.

With the use of the 'parameterized type <E>', the 'Element-type' that can be put into the collection can be specified when the collection object is created.

Collection<Integer> ageList = [**new**](http://en.wikibooks.org/wiki/Java_Programming/Keywords/new) ArrayList<Integer>();

ageList.add( [**new**](http://en.wikibooks.org/wiki/Java_Programming/Keywords/new) Integer(46) ); // --- Integer can be added

ageList.add( "50" ); // --- Compilation error, ageList can have only Integers inside

ageList is a collection that can contain only Integer objects as elements. No casting is required when we take out an element.

Integer age = ageList.get(0);

For more information about Generics, please go to [Java Programming/Generics](http://en.wikibooks.org/wiki/Java_Programming/Generics).

**[**[**edit**](http://en.wikibooks.org/w/index.php?title=Java_Programming/Collection_Classes&action=edit&section=3)**] Collection or Map**

The Java JDK contains two high level Interfaces:

* java.util.Collection
* java.util.Map

Implementations for those interfaces are not interchangeable. Collections are used for collecting Java objects. Maps are used for mapping key/value pairs.

**[**[**edit**](http://en.wikibooks.org/w/index.php?title=Java_Programming/Collection_Classes&action=edit&section=4)**] Collection**

Use the Collection interface if you need to keep related (usually the same type of) objects together in a collection where you can:

* Search for a particular element
* List the elements
* Maintain and/or change the order of the elements by using the collection basic operations (Add, Remove, Update,..)
* Access the elements by an index number

The advantages of using the Collection interface are:

* Gives a generic usage, as we talked about above, it is easy to switch implementation
* It makes it easy to convert one type of collection to an other.

The Collection interface defines the following basic operations:

* [**boolean**](http://en.wikibooks.org/wiki/Java_Programming/Keywords/boolean) add( E o );
* [**boolean**](http://en.wikibooks.org/wiki/Java_Programming/Keywords/boolean) addAll( Collection c );
* [**boolean**](http://en.wikibooks.org/wiki/Java_Programming/Keywords/boolean) remove( [Object](http://en.wikibooks.org/wiki/Java_Programming/API/java.lang.Object) o );
* [**boolean**](http://en.wikibooks.org/wiki/Java_Programming/Keywords/boolean) removeAll( Collection c );
* [**boolean**](http://en.wikibooks.org/wiki/Java_Programming/Keywords/boolean) retainAll( Collection c );

The methods above return [**true**](http://en.wikibooks.org/wiki/Java_Programming/Literals/Boolean_Literals/true) if the collection has changed due to the operation. Note that in addAll() we can add any type of collection. This is the beauty of using the Collection interface. You can have a LinkedList and just call the addAll(list) method, passing in a list. You can pass in a Vector, an ArrayList, a HashSet, a TreeSet, a YourImpOfCollection, ... All those different type of collections will be **magically** converted to a LinkList.

Lets have a closer look at this *magic*. The conversion is easy because the Collection interface defines a standard way of looping through the elements. The following code is a possible implementation of addAll() method of the LinkList.

[**import**](http://en.wikibooks.org/wiki/Java_Programming/Keywords/import) java.util.Collection

[**import**](http://en.wikibooks.org/wiki/Java_Programming/Keywords/import) java.util.Iterator

...

[**public**](http://en.wikibooks.org/wiki/Java_Programming/Keywords/public) [String](http://en.wikibooks.org/wiki/Java_Programming/API/java.lang.String) addAll( Collection coll )

{

int sizeBefore = \_linkList.size();

**Iterator iter = coll.iterator();**

while( iter.hasNext() )

{

**\_linkList.add( iter.next() );**

}

if ( sizeBefore > \_linkList.size() )

{

[**return**](http://en.wikibooks.org/wiki/Java_Programming/Keywords/return) [**true**](http://en.wikibooks.org/wiki/Java_Programming/Literals/Boolean_Literals/true);

}

else

{

[**return**](http://en.wikibooks.org/wiki/Java_Programming/Keywords/return) [**false**](http://en.wikibooks.org/wiki/Java_Programming/Literals/Boolean_Literals/false);

}

}

The above code just iterates through the passed in collection and adds the elements to the link list. You do not have to do that, since that is already defined. What you might need to code for is to loop through a 'Customer' collection:

[**import**](http://en.wikibooks.org/wiki/Java_Programming/Keywords/import) java.util.Collection

[**import**](http://en.wikibooks.org/wiki/Java_Programming/Keywords/import) java.util.Iterator

[**import**](http://en.wikibooks.org/wiki/Java_Programming/Keywords/import) java.yourcompany.Customer

...

[**public**](http://en.wikibooks.org/wiki/Java_Programming/Keywords/public) [String](http://en.wikibooks.org/wiki/Java_Programming/API/java.lang.String) printCustomerNames( Collection customerColl )

{

StringBuffer buf = [**new**](http://en.wikibooks.org/wiki/Java_Programming/Keywords/new) StringBuffer();

**Iterator iter = customerColl.iterator();**

while( iter.hasNext() )

{

**Customer cust = (Customer) iter.next();**

buf.append( cust.getName() );

buf.append( "\n" );

}

[**return**](http://en.wikibooks.org/wiki/Java_Programming/Keywords/return) buf.toString();

}

Notice two things:

* The above code will work for all type of collections.
* We have to know the type of objects inside the collection, because we call a method on it.

**[**[**edit**](http://en.wikibooks.org/w/index.php?title=Java_Programming/Collection_Classes&action=edit&section=5)**] Map**

[](http://en.wikibooks.org/wiki/File:Java_map_interfaces.svg)

[http://bits.wikimedia.org/skins-1.17/common/images/magnify-clip.png](http://en.wikibooks.org/wiki/File:Java_map_interfaces.svg)

Figure 1:Map Interfaces

A map, sometimes also called an *Associated Array* or a *Dictionary*, can be thought of as an array where the index need not be an integer.

Use the Map interface if you need to keep related objects together in a Map where you can:

* Access an element by a key object
* Map one object to other

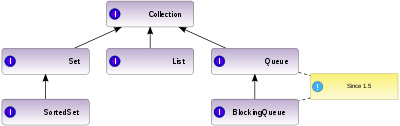
java.util.Map<K,V>

maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value. The Map interface provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The key is usually a non-mutable object. The value object however can be a mutable object.

java.util.SortedMap<K,V>

same as the Map interface, plus the keys in the Map are sorted.

**[**[**edit**](http://en.wikibooks.org/w/index.php?title=Java_Programming/Collection_Classes&action=edit&section=6)**] Set or List or Queue**

[](http://en.wikibooks.org/wiki/File:Java_collection_interfaces.svg)

[http://bits.wikimedia.org/skins-1.17/common/images/magnify-clip.png](http://en.wikibooks.org/wiki/File:Java_collection_interfaces.svg)

Figure 2:

There is no direct implementation for the java.util.Collection interface. The Collection interface has five sub interfaces. Those are:

java.util.Set<E>

contains unique elements, so duplicates not allowed. It is similar to a mathematical Set.

java.util.List<E>

elements are put in the list in a certain order, and can be accessed by an index. Duplicates are allowed, the same element can be added to a list.

java.util.SortedSet<E>

same as the Set interface; plus the elements in the SortedSet are sorted

java.util.Queue<E>

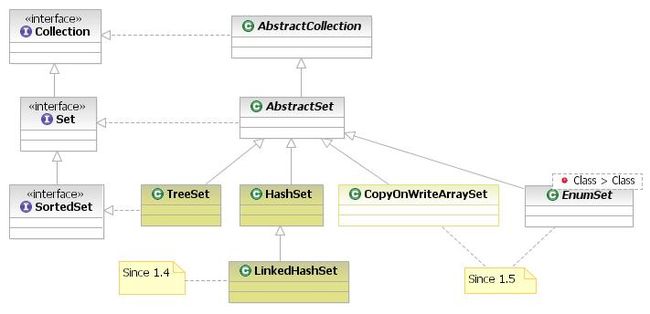
queues provide additional insertion, extraction, and inspection operations. There are FIFO (first in, first out) and LIFO (last in, first out) queues.

java.util.BlockingQueue<E>

waits for the queue to become non-empty when retrieving an element, and waits for space to become available in the queue when storing an element. Best used for producer-consumer queues.

**[**[**edit**](http://en.wikibooks.org/w/index.php?title=Java_Programming/Collection_Classes&action=edit&section=7)**] Set**

The basic implementation of the Set interface is the HashSet.

[](http://en.wikibooks.org/wiki/File:Java_collection_set_implementations.jpg)

[http://bits.wikimedia.org/skins-1.17/common/images/magnify-clip.png](http://en.wikibooks.org/wiki/File:Java_collection_set_implementations.jpg)

java.util.TreeSet<E>

Elements are sorted, not synchronized. [**null**](http://en.wikibooks.org/wiki/Java_Programming/Literals/null) not allowed

java.util.HashSet<E>

Not synchronized. Allows the [**null**](http://en.wikibooks.org/wiki/Java_Programming/Literals/null) elements

java.util.CopyOnWriteArraySet<E>

Thread safe, a fresh copy is created during modification operation. Add, update, delete are expensive.

java.util.EnumSet<E extends Enum<E>>

All of the elements in an enum set must come from a single enum type that is specified, explicitly or implicitly, when the set is created. Enum sets are represented internally as bit vectors.

java.util.LinkedHashSet<E>

Same as HashSet, plus defines the iteration ordering, which is the order in which elements were inserted into the set.

**[**[**edit**](http://en.wikibooks.org/w/index.php?title=Java_Programming/Collection_Classes&action=edit&section=8)**] Detecting duplicate objects in Sets**

Set cannot have duplicates in it. You may wonder how duplicates are detected when we are adding an object to the Set. We have to see if that object exists in the Set or not. It is not enough to check the object references, the objects' values have to be checked as well.

To do that, fortunately, each java object has the [boolean equal(Object obj);](http://en.wikibooks.org/wiki/Java_Programming/API/java.lang.Object#equals.28.29_Method), method available inherited from [Object](http://en.wikibooks.org/wiki/Java_Programming/API/java.lang.Object). You need to override it. That method will be called by the Set implementation to compare the two objects to see if they are equal or not.

There is a problem, though. What if I put two different type of objects to the Set. I put an Apple and an Orange. They can not be compared. Calling the [equal()](http://en.wikibooks.org/wiki/Java_Programming/API/java.lang.Object) method would cause a ClassCastException. There are two solutions to this:

* **Solution one** : Override the [int hashCode()](http://en.wikibooks.org/wiki/Java_Programming/API/java.lang.Object#hashCode.28.29_Method) method and return the same values for the same type of objects and return different values for different type of objects. The [equal()](http://en.wikibooks.org/wiki/Java_Programming/API/java.lang.Object) method is used to compare objects only with the same value of hashCode. So before an object is added, the Set implementation needs to:
  + find all the objects in the Set that has the same hashCode as the candidate object hashCode
  + and for those, call the [equal()](http://en.wikibooks.org/wiki/Java_Programming/API/java.lang.Object) methods passing in the candidate object
  + if any of them returns true, the object is not added to the Set.
* **Solution two** : Create a super class for the Apple and Orange, let's call it Fruit class. Put Fruits in the Set. You need to do the following:
  + Do not override the equals() and hashCode() methods in the Apple and Orange classes
  + Create appleEquals() method in the Apple class, and create orangeEquals() method in the Orange class
  + Override the hashCode() method in the Fruit class and return the same value, so the equals() is called by the Set implementation
  + Override the equals() method in the Fruit class for something like this.

|  |  |
| --- | --- |
| Computer code | public boolean equals( Object obj ) {     boolean ret = false;     if ( this instanceof Apple &&           obj instanceof Apple )     {         ret = this.appleEquals(obj);     }     else if ( this instanceof Orange &&               obj  instanceof Orange )     {         ret = this.orangeEquals(obj);       }     else     {         // --- Can not compare Orange to Apple ---        ret = false;     }     return ret; } |

**Note:**

* only the objects that have the same hashCode will be compared.
* you are responsible to override the equal() and hashCode() methods. The default implementations in Object won't work.
* Only override the [hashCode()](http://en.wikibooks.org/wiki/Java_Programming/API/java.lang.Object#hashCode.28.29_Method) method if you want to eliminate value duplicates.
* Do not override the [hashCode()](http://en.wikibooks.org/wiki/Java_Programming/API/java.lang.Object#hashCode.28.29_Method) method if you know that the values of your objects are different, or if you only want to prevent adding the exactly same object.
* Beware that the [hashCode()](http://en.wikibooks.org/wiki/Java_Programming/API/java.lang.Object#hashCode.28.29_Method) may be used in other collection implementaions, like in a Hashtable to find an object fast. Overriding the default hashCode() method may affect performance there.
* the default hashCodes are unique for each object created, so if you decide not to override the hashCode() method, there is no point overriding the equal() method, as it won't be called.

**[**[**edit**](http://en.wikibooks.org/w/index.php?title=Java_Programming/Collection_Classes&action=edit&section=9)**] SortedSet**

The SortedSet interface extends the Set Interface. All elements in the SortedSet must implement the Comparable Interface, futher more all elements must be mutually comparable.

Note that the ordering maintained by a sorted set must be consistent with equals if the sorted set is to correctly implement the Set interface. This is so because the Set interface is defined in terms of the equals operation, but a sorted set performs all element comparisons using its compareTo (or compare) method, so two elements that are deemed equal by this method are, from the standpoint of the sorted set, equal.

The SortedSet interface has additional methods due to the sorted nature of the 'Set'. Those are:

* E first(); -- returns the first element
* E last(); -- returns the last element
* SortedSet headSet(E toElement); -- returns from the first, to the exclusive toElement
* SortedSet tailSet(E fromElement); -- returns from the inclusive fromElement to the end