

- 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** 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 existing implementation. But first decide if you need a 'Collection', or a 'Map'.

Arrays

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 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 ArrayList<Integer>();
ageList.add( 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](#).

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.

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** `add( E o );`
- **boolean** `addAll( Collection c );`
- **boolean** `remove( Object o );`
- **boolean** `removeAll( Collection c );`
- **boolean** `retainAll( Collection c );`

The methods above return **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 `LinkedList`.

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 `LinkedList`.

```
import java.util.Collection
import java.util.Iterator
...
public String addAll( Collection coll )
{
    int sizeBefore = _linkList.size();
    Iterator iter = coll.iterator();
    while( iter.hasNext() )
    {
        _linkList.add( iter.next() );
    }
    if ( sizeBefore > _linkList.size();
    {
```

```

        return true;
    }
    else
    {
        return 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 java.util.Collection
import java.util.Iterator
import java.yourcompany.Customer
...
public String printCustomerNames( Collection customerColl )
{
    StringBuffer buf = new StringBuffer();

    Iterator iter = customerColl.iterator();
    while( iter.hasNext() )
    {
        Customer cust = (Customer) iter.next();
        buf.append( cust.getName() );
        buf.append( "\n" );
    }
    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.

Map

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>`

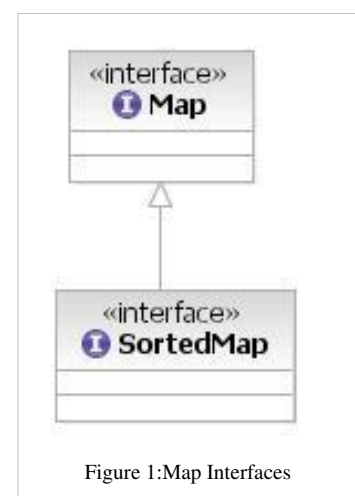


Figure 1:Map Interfaces