The Java Collections Framework

The Java Collections Framework

- An array is a very useful type in Java but it has its restrictions:
- once an array is created it must be sized, and this size is fixed;
- it contains no useful pre-defined methods.
- Java comes with a group of generic collection classes that grow as more elements are added to them, and these classes provide lots of useful methods.
- This group of collection classes are referred to as the **Java Collections Framework**.
- The classes in the JCF are all found in the `java.util` package.
- Three important interfaces from this group are:
  - `List;`
  - `Set;`
  - `Map.`

Collections Hierarchy

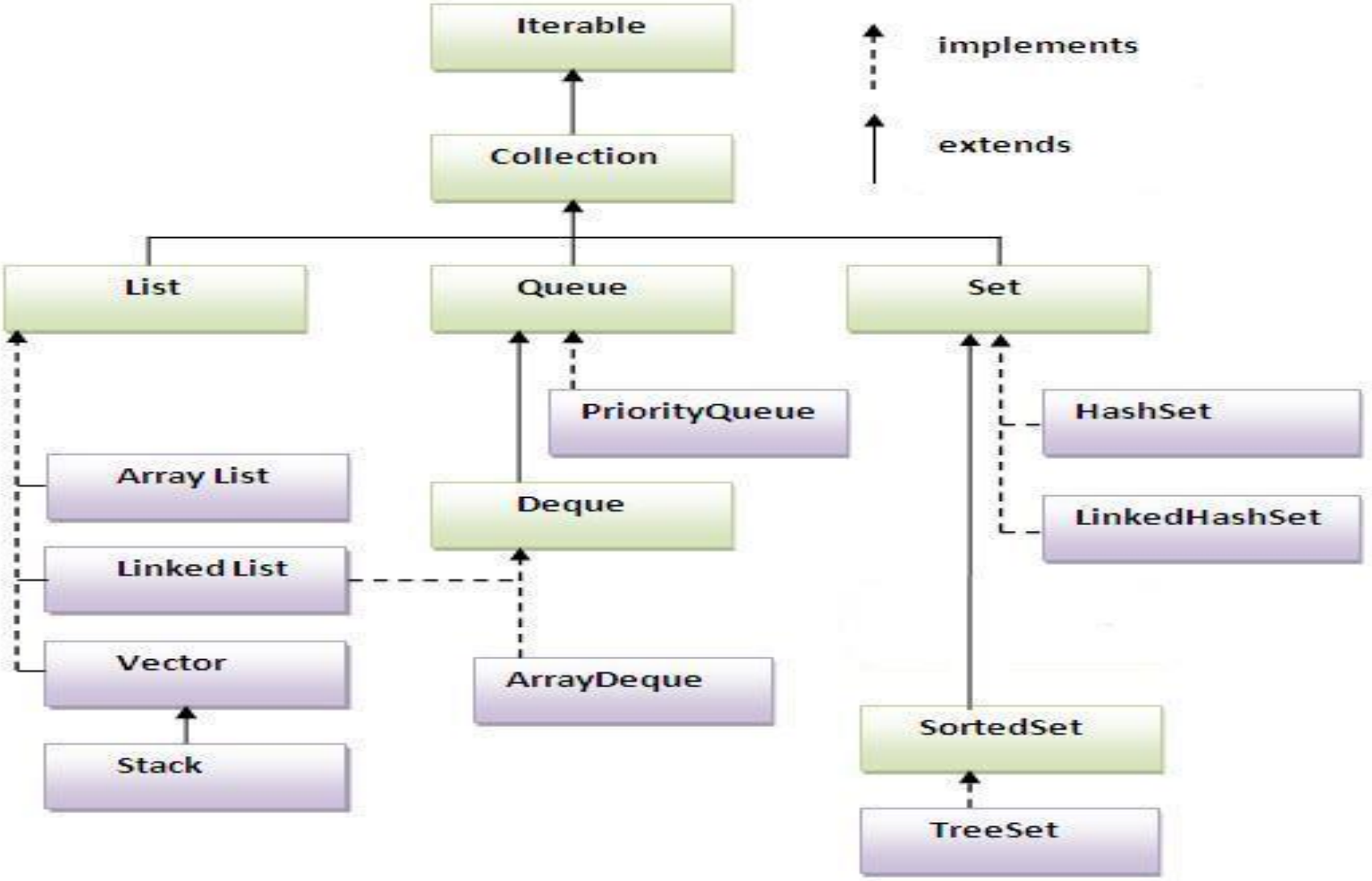

Image Source: http://www.javatpoint.com/

The *List* interface

- The List interface specifies the methods required to process an *ordered list* of objects.
- Such a list may contain duplicates.

Examples of a list of objects:

- jobs waiting for a printer,
- emergency calls waiting for an ambulance
- the names of players that have won the Wimbledon tennis tournament over the last 10 years.
- we often think of such a collection as a *sequence* of objects.
- there are two implementations provided for the List interface in the JCF.
- they are ArrayList and LinkedList.
- here we will look at the ArrayList class

Using an *ArrayList* to store a queue of jobs waiting for a printer

- We will represent these jobs by a series of Job ID Strings.
- The ArrayList constructor creates an empty list:

```
// creates an ArrayList object -'printQ'
ArrayList<String> printQ = new ArrayList<String>();
```

- The stuff in angled brackets allows us to *fix* the type of objects stored in a particular collection object.
- Remember **Generic Classes**?

Using the interface type instead of the implementation type

- It is considered good programming practice to declare collection objects to be the type of the *interface* rather than the type of the *class* that implements this collection.
- So this would be a better way to create our printQ object:

```
// the type is given as 'List' not 'ArrayList'
List<String> printQ = new ArrayList<String>();
```

- A method that receives a printQ object, would now be declared as follows:

```
// this method recieves a List<String> object
public void someMethod (List<String> printQIn)
{
   // some code here
}
```

- The advantage of this approach is that we can change our choice of implementation in the future without having to change the type of the object.

List methods - *add*

The List interface defines two add methods for inserting into a list

- one inserts the item at the end of the list;
- the other inserts the item at a specified position in the list.

We wish to use the first add method

This add method requires one parameter, the object to be added into the list:

```
printQ.add("myLetter.doc");
printQ.add("myFoto.jpg");
printQ.add("results.xls");
printQ.add("chapter.doc");
```

List methods - *toString*

All the Java collection types have a `toString` method defined

So we can display the entire list to the screen:

```
System.out.println(printQ); /* implicitly calling the
                               toString method */
```

Lists are displayed as follows.

[myLetter.doc, myFoto.jpg, results.xls, chapter.doc]

List methods – *add* revisited

The add method is overloaded to allow an item to be inserted into the list at a particular position.

When the item is inserted into that position, the item previously at that particular position and all items behind it shuffle along by one place.

This add method requires two parameters, the position into which the object should be inserted, and the object itself.

```
printQ.add(0, "importantMemo.doc");
                        // inserts into front of the queue
```

List methods – *set*

- If we wish to overwrite an item in the list, rather than insert a new item into the list, we can use the set method.
- The set method requires two parameters, the index of the item being overwritten and the new object to be inserted at that position.
- Let us change the name of the last job from "chapter.doc", to "newChapter.doc".

```
printQ.set(4, "newChapter.doc"); // fifth item at index 4
```

List methods – *size*

- Lists provide a size method to return the number of items in the list
- So we could have renamed the last job in the queue in the following way also:

```
printQ.set(printQ.size()-1, "newChapter.doc");
                            // last position is size-1
```

List methods – *indexOf*

The indexOf method returns the index of the first occurrence of a given object within the list.

It returns -1 if the object is not in the list.

Example: finding "myFoto.jpg"

```
int index = printQ.indexOf("myFoto.jpg");  // check index of job

if (index != -1) // check object is in list
{
   System.out.println("myFoto.jpg is at index position: " + index);
}
else // when job is not in list
{
   System.out.println("myFoto.jpg not in list");
}
```

List methods – *remove*

Items can be removed either by specifying an index or an object.

When an item is removed, items behind this item shuffle to the left

Example : removing "myFoto.jpg"

If we used its index, the following is required

```
printQ.remove(2);
```

Alternatively, we could have removed the item by referring to it directly rather than its index:

```
printQ.remove(“myFoto.jpg”);
```

List methods – *get*

The get method allows a particular item to be retrieved from the list via its index position.

The following displays the job at the head of the queue:

```
// the first item is at position 0
System.out.println("First job is " + printQ.get(0));
```

This would display the following:

First job is importantMemo.doc

List methods – *contains*

The contains method can be used to check whether or not a particular item is present in the list:

```
if (printQ.contains("poem.doc"))
                              // check if value is in list
{
   System.out.println("poem.doc is in the list");
}
else
{
   System.out.println("poem.doc is not in the list");
}
```

List methods – *isEmpty*

- The isEmpty method reports on whether or not the list contains any items
  Using the enhanced 'for' loop with collection classes
- The enhanced **for** loop can be used with the List (and Set) implementations provided in the JCF.
- For example, here an enhanced **for** loop is used to iterate through the printQ list to find and display those jobs that end with a ".doc" extension:

```
for (String item: printQ) /* iterate through all
                             items in the 'printQ' list */
{
    if (item.endsWith(".doc")) // check the extension of the job ID
    {
          System.out.println(item); // display this item
    }
}
```

Collections Hierarchy

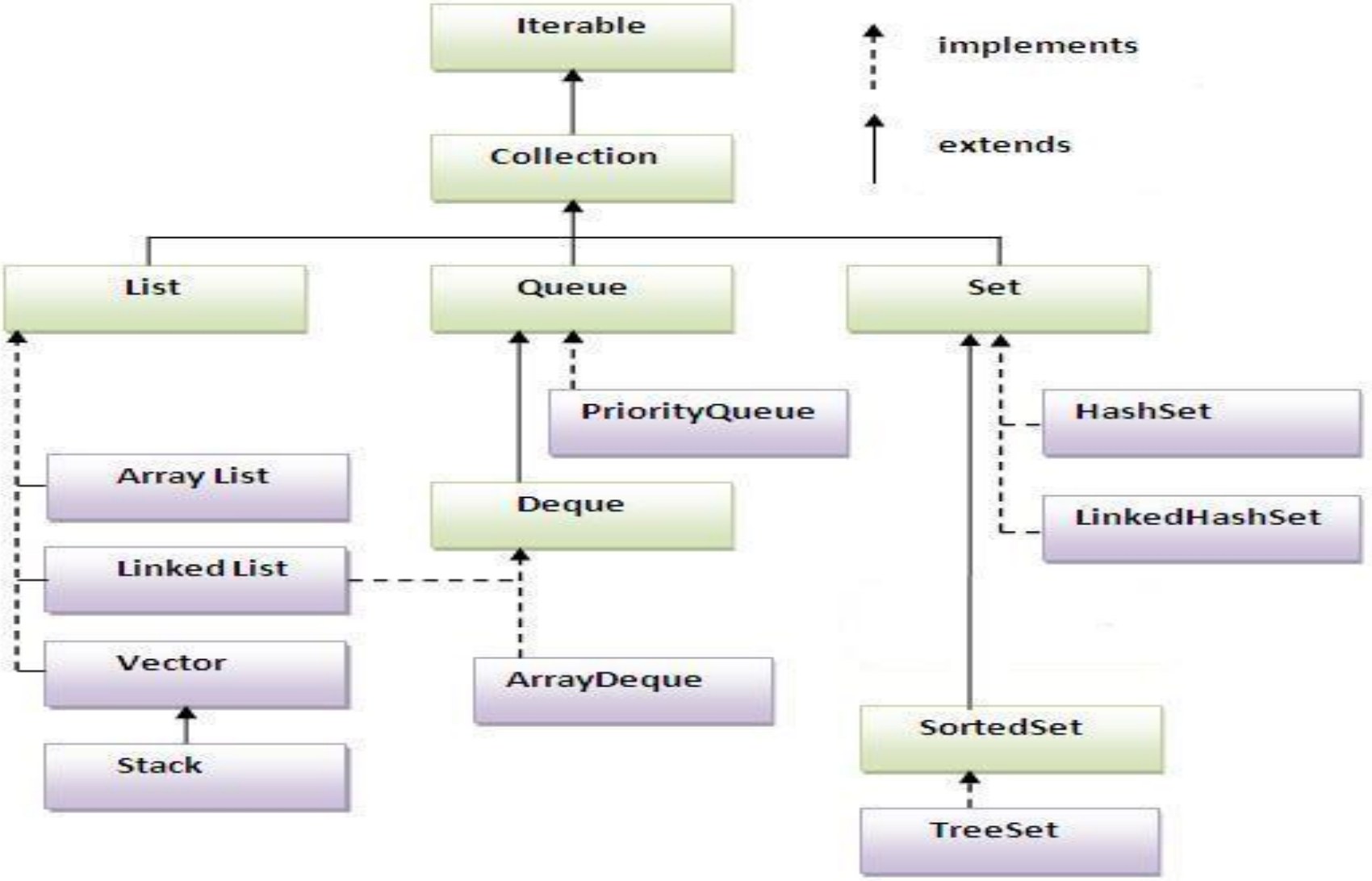

Image Source: http://www.javatpoint.com/

ArrayList: Properties

- Java ArrayList class uses a dynamic array for storing the elements.
- Extends AbstractList class and implements List interface.
- Can contain duplicate elements.
- Maintains insertion order.
- Is non synchronized- Will cover this later
- Allows random access because array works at the index basis.
- Manipulation is slow. Example, a lot of shifting needs to be done if any element is removed from the array list.

LinkedList: Properties

- Java LinkedList class uses doubly linked list to store the elements.
- Extends the AbstractList class and implements List and Deque interfaces.
- Can contain duplicate elements.
- Maintains insertion order.
- Is non synchronized.
- Manipulation is fast because no shifting needed.

The *Set* interface

- The `Set` interface defines the methods required to process a collection of objects in which there is no repetition, and ordering is unimportant.
- Which of these are sets?
  - a queue of people waiting to see a doctor;
  - a list of specific records for each of the 52 weeks of a particular year;
  - car registration numbers allocated parking permits.
- Only the collection of car registration numbers can be considered a set as there will be no duplicates and ordering is unimportant.
- There are two implementations provided for the Set interface in the JCF.
- They are `HashSet` and `TreeSet`.
- Here we will look at the `HashSet` class.

Using a *HashSet* to store a collection of vehicle registration numbers

The constructor creates an empty set:

```
// creates an empty set of String objects
Set<String> regNums = new HashSet<String>();
```

Again, notice that

- we have used the generics mechanism to indicate that this is a set of String objects, and
- we have given the type of this object as the interface Set<String>.

Set methods - add

The add method allows us to insert objects into the set

```
regNums.add("V53PLS");
regNums.add("X85ADZ");
regNums.add("L22SBG");
regNums.add("W79TRV");
```

Set methods - *toString*

We can display the entire set as follows::

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

The set is displayed in the same format as a list:

[W79TRV, X85ADZ, V53PLS, L22SBG]

Set methods - *size*

As with a list, the size method returns the number of items in the set

```
System.out.println("Number of items in set: " + regNums.size() );
```

Set methods - *remove*

The `remove` method deletes an item from the set if it is present.

```
regNums.remove("X85ADZ");
```

regNums.**remove**("X85ADZ");
If we now display the set, the given registration will have been removed:
[W79TRV, V53PLS, L22SBG]
The `Set` interface also includes contains and `isEmpty` methods that work in exactly the same way as their `List` counterparts

Using the enhanced ‘for’ loop to iterate through a set

The following enhanced **for** loop will allow us to iterate through the collection of registration numbers and display all registrations after ‘T’.

```
for (String item: regNums) // iterate through all items in 'regNums'
{
      if (item.charAt(0)> 'T') // check first letter of registration
      {
         System.out.println(item); // display this registration
      }
 }
```

Assuming we have the following set of registration numbers:

[W79TRV, V53PLS, L22SBG]

The enhanced **for** loop above would produce the following result:

W79TRV

V53PLS

Iterator objects

- An Iterator object allows the items in a collection to be retrieved by providing three methods defined in the Iterator interface:

Methods of the *Iterator* interface			
Method	**Description**	**Inputs**	**Outputs**
`hasNext`	Returns **`true`** if there are more elements in the collection to retrieve and **`false`** otherwise.	None	An item of type **`boolean`**.
`next`	Retrieves one element from the collection.	None	An item of the given element type.
`remove`	Removes from the collection the element that is currently retrieved	None	None

- To obtain an Iterator object from a set, the iterator method is called.

Using an *Iterator* object with a 'while' loop

The following removes all registration prior to and including "T"

```
/* an Iterator object can be used with a 'while' loop if we wish to
   iterate over a set and modify its contents */

// first create an Iterator object as discussed before
Iterator<String> elements = regNums.iterator();

// repeatedly retrieve items as long as there are items to be retrieved

while (elements.hasNext())
{
   String item = elements.next(); // retrieve next element from set
  if (item.charAt(0)<= 'T') // check first letter of registration
   {
         elements.remove(); /* call Iterator method to
                               remove registration */
   }
}
```

STL vs Java Iterator

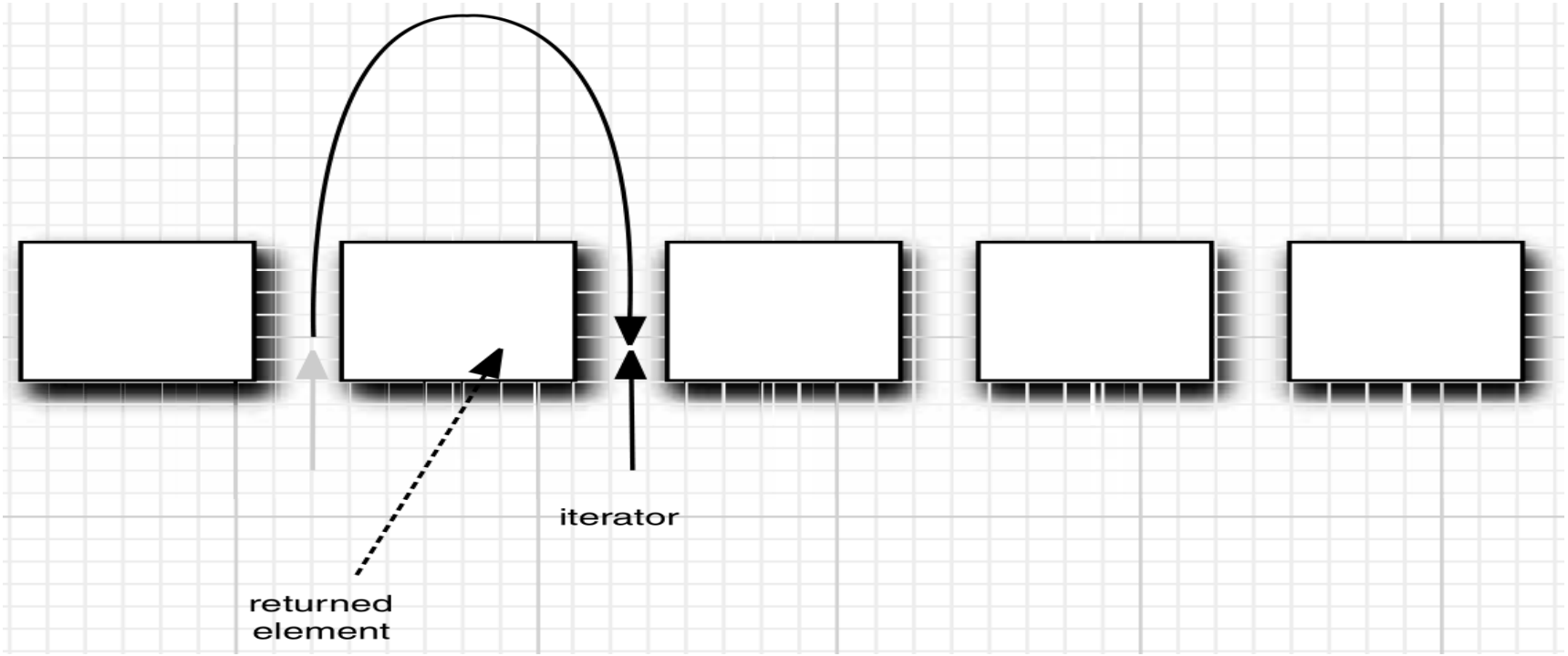

Image Source: Core Java

Removing Elements

- `Iterator<String> it = c.iterator();`
- `it.next(); // skip over the first element`
- `it.remove(); // now remove it`

- `it.remove();`
- `it.remove(); // Error!`

- `it.remove();`
- `it.next();`
- `it.remove(); // Ok`

Queues

- A queue lets you efficiently add elements at the tail and remove elements from the head.
- Queue is an interface
  - You need to instantiate a concrete implementation of the interface in order to use it.
  - Choose between the following Queue implementations in the Java Collections API:
    - java.util.LinkedList:

      ```
      Queue queueA = new LinkedList();
      ```

    - java.util.PriorityQueue:

      ```
      Queue queueB = new PriorityQueue();
      ```

Priority Queue

- Retrieves elements in sorted order after they were inserted in arbitrary order.
  - You get the smallest element currently in the priority queue on calling the remove method

- Makes use of heap data-structure

- Priority queue can either hold elements of a class that implements
  - Comparable interface or
  - Comparator object you supply in the constructor

Priority Queue

```
public static void main(String[] args)
{
   PriorityQueue<GregorianCalendar> pq = new PriorityQueue<>();
   pq.add(new GregorianCalendar(1906, Calendar.DECEMBER, 9));
   pq.add(new GregorianCalendar(1815, Calendar.DECEMBER, 10));
   pq.add(new GregorianCalendar(1903, Calendar.DECEMBER, 3));
   pq.add(new GregorianCalendar(1910, Calendar.JUNE, 22));

   System.out.println("Iterating over elements...");
   for (GregorianCalendar date : pq)
         System.out.println(date.get(Calendar.YEAR));

   System.out.println("Removing elements...");
   while (!pq.isEmpty())
         System.out.println(pq.remove().get(Calendar.YEAR));
}
```

Output?

Source: Core Java, Volume 1

Priority Queue

Output

```
Iterating over elements...
1815
1906
1903
1910
Removing elements...
1815
1903
1906
1910
```

Priority Queue

- The priority queue does not sort all its elements

The *Map* interface

- The Map interface defines the methods required to process a collection consisting of *pairs* of objects.
- Rather than looking up an item via an index value, the first object of the pair is used.
- The first object in the pair is considered a **key**.
- The second object in the pair is its associated **value**.
- Ordering is unimportant in maps, and keys are unique.
- It is often useful to think of a map as a *look-up* table, with the key object the item used to look up (access) an associated value in the table.
- There are two implementations provided for the Map interface.
- They are HashMap and TreeMap.
- Here we will look at the HashMap class.

Using a *HashMap* to store a collection of user names and passwords

- The constructor creates the empty map:

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

- As before the type of the collection is given as the interface: Map.
- To use the generics mechanism to fix the types used in a Map object, we must provide *two* types in the angled brackets.
- The first type will be the type of the key and the second the type of its associated value. In this case, *both* are String objects, but in general each may be of any object type.

Map methods - *put*

- To add a user's name and password to this map we use the put method
- The put method requires two parameters, the key object and the value object:

```
users.put("lauraHaliwell", "popcorn");
```

- The put method treats the first parameter as a key item and the second parameter as its associated value.
- The put method overrides the value associated with a key if that key is already present in the map.

Map methods - *containsKey*

- The containsKey method accepts an object and returns **true** if the object is a key in the map and **false** otherwise:
- There is also a containsValue method to check for the presence of a value in a map.

Map methods - get

- The get method accepts an object and searches for that object among the keys of the map.
- If it is found, the associated value object is returned.
- If it is not found the **null** value is returned:

Map methods - toString

- Maps are displayed in the following output:

```
{lauraHaliwell=popcorn,sunaGuven=television, bobbyMann=elephant}
```

Iterating over the elements of a map

In order to scan the items in the map, the keySet method can be used to return the set of keys.

```
/* the keySet method returns the keys of the map as a
   set object */
Set<String> theKeys = users.keySet();
```

The set of keys can then be processed in the ways discussed previously for sets.

Map methods - *remove*

- The remove method accepts a key value and, if the key is present in the map, both the key and value pair are removed:

```
// this removes the given key and its associated value
users.remove("lauraHaliwell");
```

- Displaying the map now shows the user's ID and password have been removed:

```
{sunaGuven=television, bobbyMann=elephant }
```

- The remove method returns the value of the object that has been removed, or **null** if they key was not present.
- The map collection also provides `size` and `isEmpty` methods that behave in exactly the same way as the `size` and `isEmpty` methods for sets and lists.

Using your own classes with Java's collection classes

- Consider an application to store a collection of books that a person may own.
- The constructor creates an empty list:

```
// create empty list to contain Book objects
List<Book> books = new ArrayList<Book>();
```

- To indicate this list will hold `Book` objects, the `Book` type is given in angled brackets.
- In general, when storing user-defined objects, such as `Book` objects, in any of the JCF collections, such objects should have three specific methods defined:
  - `toString`
  - `equals`
  - `hashCode`

Defining a *toString* method

Here is one possible toString method we could provide for our Book class:

```
public String toString()
{
    return "(" + isbn +", "+ author + ", " + title +")\n";
}
```

Defining an *equals* method

- One possible interpretation of two books being equal is simply that their ISBNs are
- equal, so the following equals method could be added to the Book class:

```
public boolean equals (Object objIn) /* equals method
                                        must have this header*/
{
   Book bookIn = (Book) objIn; // type cast to a Book
   return   isbn.equals(bookIn.isbn); // check isbn
}
```

The *hashCode* method

- The `hashCode` method returns an integer value from an object.
- This integer value determines where in the `HashMap` and `HashSet` collection the given object is stored.
- Rather than searching the whole collection for an object, just those with identical hash codes are checked.
- Objects that are equal (as determined by the object's equals method) should produce identical `hashCode` numbers and, ideally, objects that are not equal should return different `hashCode` numbers.
- We need to define our own `hashCode` method for the Book class so that objects of this class can be used effectively with the `HashSet` and `HashMap` classes.

Defining your own *hashCode* method

- All of Java's predefined classes (such as String) have a meaningful *hashCode* method defined.
- So one way of defining the *hashCode* number for an object of your class would be to add together the *hashCode* numbers generated by all the attributes to determine object equality.
- For Book equality we checked the ISBN only.
- This ISBN is a String, so all we need to do is to return the *hashCode* number of this String:

```
// this is a suitable hashCode method for our Book class
public int hashCode()
{
   // derive hash code by returning hash code of ISBN string
    return isbn.hashCode();
}
```