The Java Collections Framework 15

Outcomes:

By the end of this chapter you should be able to:

- *use the* `ArrayList` *class to store a* ***list*** *of objects*;
- *use the* `HashSet` *class to store a* ***set*** *of objects*;
- *use the* `HashMap` *class to store objects in a* ***map***;
- *use the enhanced* **`for`** *loop and a* **`forEach`** *loop to scan through a collection*;
- *use an* `Iterator` *object to scan through a collection*;
- *create objects of your own classes, and use them in conjunction with Java's collection classes*;
- *sort elements in a collection using the* `Comparable<T>` *and* `Comparator<T>` *interfaces.*

15.1 Introduction

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.

Think back to the `SomeUsefulArrayMethods` program of Chap. 6. As this program used an array to hold a collection of elements we had to put an upper limit to the size of this collection. Sometimes, however, an upper limit is not known. Just creating a very big array is very wasteful of memory—and what happens if even this very big array proves to be too small? Another problem, as seen in our `SomeUsefulArrayMethods` program, was that to carry out any interesting processing (like searching the array) required us to write complex algorithms.

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_15

One solution to this problem might be to develop our own collection class that hid the array from the user and contained the code to create a reasonable sized array and, when the array is full, copy this array into a slightly bigger array and use this new array and continue doing this every time the array gets full. This collection class could also include a large group of useful methods to process the array (such as searching and deleting). In order to make this collection class as useful as possible we could incorporate generics (that we met in Chap. 13) to allow this collection class to hold a collection of any type. This collection class could then be used in place of an array to hold and process a collection of objects.

Luckily we do not need to go to such lengths; the Java developers have already done this for us. They have developed 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** (**JCF**). These collection classes are organized around several collection interfaces that define different types of collections that we might be interested in using. Three important interfaces from this group are:

- `List`;
- `Set`;
- `Map`.

The `List` and `Set` interfaces are themselves specialised types of the super interface named `Collection`, and so share many common methods, whereas the `Map` interface is distinct from `Collection` group of interfaces and so has several methods that are unique to it.

As well as providing a wide group of collection interfaces, the Java Collections Framework also contains many classes that implement these interfaces. In this chapter we will focus on the three key interfaces named above (`List`, `Set` and `Map`) and some of the classes provided in the JCF which implement these interfaces. To find out about all the interfaces and classes in the JCF you can refer to the Oracle™ site.

15.2 The *List* Interface and the *ArrayList* Class

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 include jobs waiting for a printer, emergency calls waiting for an ambulance and the names of players that have won the Wimbledon tennis tournament over the last 10 years. In each case ordering is important, and repetition may also be required. 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`. We have already introduced you to the `ArrayList` in the `Bank` application in Chap. 8, that stored a collection of `BankAccount` objects, so let's take a closer look at the `ArrayList` class now.

15.2.1 Creating an *ArrayList* Collection Object

All classes in the JCF are in the `java.util` package, so to use the `ArrayList` class we require the following **`import`** statement:

```
import java.util.ArrayList;
```

Like all the classes in JCF the `ArrayList` class is a *generic* collection class. This means it can be used to store objects of *any* type.[1] Let's use an `ArrayList` to store a queue of jobs waiting for a printer, and let us 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<> ();
```

Notice the use of generics to set the type of objects stored in our list. In the case of the `printQ` object, we want each element within this collection to be of type `String`. As mentioned in Chap. 7, type inference means that we do not need to include the `String` type in the angle brackets in the call to the constructor on the right, as the type is inferred to be `String` in this case.

Of course, the generics mechanism can be used to fix *any* object type for the elements within a collection. In Chap. 8, for example, we used an `ArrayList` to store a collection of `BankAccount` objects in our `Bank` class. As another example, if you wished create a list of `Oblong` objects you could do so as follows:

```
// this will make 'someOblongs' a list of Oblong objects
ArrayList<Oblong> someOblongs = new ArrayList<> ();
```

You should be aware that the type of any object created using this generics mechanism is now the class name *plus* the contained type brackets. So, for example, if we were to write a method that received a list of strings, we would specify it as follows:

[1] As we mentioned in Chap. 7, generic collections cannot store primitive types like **`int`** and **`double`**. If primitive types are required then objects of the appropriate wrapper class (`Integer`, `Double` and so on) must be used. However, as discussed in Chap. 9, *autoboxing* and *unboxing* automate the process of moving from a primitive type to its associated wrapper.

```
// this method receives an ArrayList<String> object
public void someMethod (ArrayList<String> printQIn)
{
    // some code here
}
```

15.2.2 The Interface Type Versus the Implementation Type

In order to create the object `printQ`, we have used the `ArrayList` class to implement the `List` interface. What if, at some point, we decide to change to the `LinkedList` implementation? Or some other implementation that might be available in the future? If we did this, then all references to the type of this object (such as in the method header of the previous section) would have to be modified to give the new class name.

There is an alternative approach. It is actually considered better programming practice to declare collection objects to be the type of the interface (`List` in this case) 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<> ();
```

Notice that the interface type still needs to be marked as being a list of `String` objects using the generics mechanism. A method that receives a `printQ` object would now be declared as follows:

```
// this method receives 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 (maybe by using `LinkedList` or some other class that might be available that implements the `List` interface), without having to change the type of the object (which will always remain as `List`). This is the approach that we will take.

Remember, all classes in the JCF reside in the `util` package, so to use the `List` interface in your code you now need to add an additional **`import`** statement as follows:

```
import java.util.ArrayList;
import java.util.List;
```

Now, let us look at some `List` methods.

15.2.3 *List* Methods

The `List` interface defines two `add` methods for inserting into a list, one that inserts the item at the end of the list and one that inserts the item at a specified position in the list. Like arrays, `ArrayList` positions begin at zero. We wish to use the `add` method that adds items to the end of the list as we are modelling a queue here. 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");
```

Notice that, since we have marked this list as containing `String` objects only, an attempt to add an object of any other type will result in a compiler error:

```
// will not compile as 'printQ' can hold Strings only!
printQ.add(new Oblong(10, 20));
```

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
```

When the list is displayed, the values in the list are separated by commas and enclosed in square brackets. So this `println` instruction would display the following list:

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

As you can see, the items in the list are kept in the order in which they were inserted using the `add` method above.

As we said earlier, 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 (that is they have their indices incremented by one). This `add` method requires two parameters, the position into which the object should be inserted, and the object itself. For example, let's insert a high priority job at the start of the queue:

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

Notice that the index is provided first, then the object. The index must be a valid index within the current list or it may be the index of the back of the queue. An invalid index throws an unchecked `IndexOutOfBoundsException`.

Displaying this list confirms that the given job ("`importantMemo.doc`") has been added to the front of the queue, and all other items shuffled by one place:

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

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`". This is the fifth item in the list so its index is 4:

```
printQ.set(4, "newChapter.doc");
```

If the index used in the `set` method is invalid an `IndexOutOfBoundsException` is thrown once again. Displaying the new list now gives us the following:

`[importantMemo.doc, myLetter.doc, myFoto.jpg, results.xls, newChapter.doc]`

`List` provides 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
```

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. For example, the following checks the index position of the job "`myFoto.jpg`" in the list:

```
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");
}
```

This would display the following from our list:

```
myFoto.jpg is at index position: 2
```

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 (i.e. they have their indices decremented by one). As an example, let us remove the "`myFoto.jpg`" job. If we used its index, the following is required:

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

Once again, an `IndexOutOfBoundsException` would be thrown if this was not a valid index. This method returns the object that has been removed, which could be checked if necessary. Displaying the list would confirm the item has indeed been removed:

```
[importantMemo.doc, myLetter.doc, results.xls, newChapter.
doc]
```

Alternatively, we could have removed the item by referring to it directly rather than by its index[2]:

```
printQ.remove("myFoto.jpg");
```

This method returns a **`boolean`** value to confirm that the item was in the list initially, which again could be checked if necessary.

Behind the scenes you can guess how this `remove` method works. It looks in the list for the given `String` object ("`myFoto.jpg`"). It uses the `equals` method of the `String` class to identify a match. Once it finds such a match it shuffles items along so there are no gaps in the list.

Of course, as we have already said, these collection classes can be used to store objects of *any* type—not just `String`s. For methods like `remove` to work properly, the contained object must have a properly defined `equals` method. We will return to this later in the chapter, when we look at how to use objects of our own classes in conjunction with the classes in the JCF.

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));
```

[2]If there were more than one occurrence of the object, the first occurrence would be deleted.

This would display the following:

First job is importantMemo.doc

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");
}
```

Finally, the `isEmpty` method reports on whether or not the list contains any items:

```
if (printQ.isEmpty()) // returns true when list is empty
{
    System.out.println("Print queue is empty");
}
else
{
    System.out.println("Print queue is not empty");
}
```

15.3 The Enhanced *for* Loop and Java Collections

In Chap. 6 we showed you how the enhanced **`for`** loop can be used to iterate through an entire array. The use of this loop is not restricted to arrays, it can also 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 extension of the job ID
   {
       System.out.println(item); // display this item
   }
}
```

Notice that the type of each item in the `printQ` list is `String`. Within the loop we use the `String` method `endsWith` to check if the given job ID ends with `String` ".doc". Assuming we had the following `printQ`:

[importantMemo.doc, myLetter.doc, results.xls, newChapter.doc]

the enhanced **for** loop above would produce the following output:

```
importantMemo.doc
myLetter.doc
newChapter.doc
```

If we do not wish to iterate through the *entire* list, or if we wish to *modify* the items within a list as we iterate through them, then (as we have said before) the enhanced **for** loop should not be used.

For example, if we wished to display the items in the `printQ` that are behind the head of the queue, the enhanced **for** loop is not appropriate as we are not processing the *entire* `printQ`. Instead the following standard **for** loop could be used:

```
// remember second item in list is at index 1!
for (int pos = 1; pos < printQ.size(); pos++)
{
     System.out.println(printQ.get(pos)); // retrieve item in printQ
}
```

Notice how the `size` method is used to determine the last index in the loop header. Within the loop, the `get` method is used to look up an item at the given index.

Again, if we assume we have the following `printQ`:

```
[importantMemo.doc, myLetter.doc, results.xls, newChapter.doc]
```

the **for** loop above would produce the following output:

```
myLetter.doc
results.xls
newChapter.doc
```

15.4 The *forEach* Loop

There is a very similar loop to the enhanced **for** loop known as the **forEach** loop. The **forEach** loop makes use of the `forEach` method, which its to be found in classes implementing the `Collection` interface (such as lists and sets). The `forEach` method requires an implementation of a `Consumer` interface, which can be provided via a lambda expression. Table 15.1 provides a reminder of the `Consumer` interface that we discussed in Chap. 13.

Table 15.1 Reminder of the `Consumer` interface

Functional interface	Abstract method name	Parameter types	Return type
`Consumer<T>`	`accept`	`T`	`void`

This implementation is then applied to each element in the underlying collection. For example, lets revisit the code in Sect. 15.3 for searching through our `printQ` list to display document names that end with ".doc". This time we will use a lambda expression and a **`forEach`** loop:

```
// using a forEach loop to iterate through a list
printQ.forEach(item ->
        {
            if(item.endsWith(".doc")) // check extension of the job ID
            {
                System.out.println(item); // display this item
            }
        });
```

As you can see, we use a **`forEach`** loop by calling the `forEach` method on the list object `printQ`. The parameter to the lambda expression is a name we give to one value from this collection, `item` in this case, and to the right of the lambda arrow we define how we process this value (using the same code we used in Sect. 15.3). The **`forEach`** loop will retrieve all the items within the collection to be processed in this way. You should note that, as with the enhanced **`for`** loop, the **`forEach`** loop should not be used to modify the underlying collection.

We will see further examples of the use of this **`forEach`** loop in this chapter. However, the **`forEach`** loop is primarily designed to be used with **streams**, a mechanism for processing Java collections that we will cover in Chap. 22.

15.5 The *Set* Interface and the *HashSet* Class

Like `List`, the `Set` interface is a specialised version of the general `Collection` interface. The `Set` interface defines the methods required to process a collection of objects in which there is no repetition, and ordering is unimportant. Let's consider the following collections and consider if they are suitable for a set:

- a queue of people waiting to see a doctor;
- a list of number-one records for each of the 52 weeks of a particular year;
- car registration numbers allocated parking permits.

The queue of people waiting to see a doctor cannot be considered to be a set as ordering is important here. Order may also be important when recording the list of number-one records in a year. It would also be necessary to allow for repetition—as a record may be number-one for more than one week. So a set is not a good choice

for this collection. The collection of car registration numbers can be considered a set, however, as there will be no duplicates and ordering is unimportant.

Java provides three implementations of this `Set` interface: `HashSet`, `TreeSet` and `LinkedHashSet`. Here we will look at the `HashSet` class. The constructor creates an empty set. We will use the set to store a collection of vehicle registration numbers (as `Strings`):

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

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>`. Now let us look at some `Set` methods.

15.5.1 *Set* Methods

Once a set is created the methods specified in the `Set` interface can be used. The `add` method allows us to insert objects into the set, so let us add a few registration numbers:

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

We can display the entire set as follows:

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

The set is displayed in the same format as a list, in square brackets and separated by commas:

```
[W79TRV, X85ADZ, V53PLS, L22SBG]
```

Notice that, unlike lists, the order in which the items are displayed is not determined by the order in which the items were added. Instead, the set is displayed in the order in which the items are stored internally (and over which we have no control). This will not be a problem as ordering is unimportant in a set.

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() );
```

This would print the following onto the screen:

```
Number of items in set: 4
```

If we try to add an item that is already in the set, the set remains unchanged. Let us assume the four items above have been added into the set and we now try and add a registration number that is already in the set:

```
regNums.add("X85ADZ"); // this number is already in the set
System.out.println(regNums);
```

When this set is displayed, "X85ADZ" appears only once:

```
[W79TRV, X85ADZ, V53PLS, L22SBG]
```

The add method returns a **boolean** value to indicate whether or not the given item was successfully added. This value can be checked if required.

```
boolean ok = regNums.add("X85ADZ"); // store boolean return value
if (!ok) //check if add method returned a value of false
{
       System.out.println("item already in the set!");
}
```

The remove method deletes an item from the set if it is present. Again, assuming that the four items given above are in the set, we can delete one item as follows:

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

If we now display the set, the given registration will have been removed:

```
[W79TRV, V53PLS, L22SBG]
```

As with the add method, the remove method returns a **boolean** value of **false** if the given item to remove was not actually in the set. The Set interface also includes contains and isEmpty methods that work in exactly the same way as their List counterparts.

15.5.2 Iterating Through the Elements of a Set

Both the enhanced **for** loop and the **forEach** loop can be used to iterate through all the elements of a set. Let us look at an example.

In the UK, the first letter of the registration number was at one time used to determine the time period when the vehicle came to market. A registration beginning with 'S', for example, denoted a vehicle that came to market between August 1998 and February 1999, while a registration beginning with 'T' denoted a vehicle that came to market between March 1999 and July 1999.

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
      }
}
```

Here is the equivalent **forEach** loop:

```
regNums.forEach ( item ->
{
      if (item.charAt(0) > 'T') // check first letter of registration

      {
            System.out.println(item); // display this registration
      }
});
```

Again, notice that the type of every element within our `regNums` set is `String`. Within the loops we use the `String` method `charAt` to check the first letter of registration. Assuming we have the following set of registration numbers:

[W79TRV, V53PLS, L22SBG]

the loops above would both produce the following result:

W79TRV
V53PLS

Let us consider a slightly different scenario now. Instead of simply displaying registration numbers after 'T', we now wish to modify the original `regNums` set so that registrations prior or equal to 'T' are removed. We could try using an enhanced **for** loop again:

```
// this will compile but is not safe!
for (String item: regNums) // iterate through all items in 'regNums'
{
     if (item.charAt(0) <= 'T') // check first letter of registration
     {
        regNums.remove(item); // remove this registration
     }
}
```

Here we are once again iterating over the elements of the `regNums` set. But this time, within the loop, we are attempting to *remove* the given element from the set. As we said, both the enhanced **`for`** loop and the **`forEach`** loop both should *not* be used to modify or remove elements from the original collection. To do so would not give a compiler error but may lead to run-time exceptions. If we cannot use these loops here, how else can we iterate over the elements in a set? Unlike an array, values in the set cannot be retrieved by an index value. Instead, to access the items in a set, the `iterator` method can be used to obtain an **iterator** object.

15.5.3 *Iterator* Objects

An `Iterator` object allows the items in a collection to be retrieved by providing three methods defined in the `Iterator` interface (see Table 15.2).

To obtain an `Iterator` object from the `regNums` set, the `iterator` method could be called as follows:

```
// the 'iterator' method retrieves an Iterator object from a set
Iterator<String> elements = regNums.iterator();
```

Here we have called the `iterator` method and stored the item returned by this method in a variable we have called elements. The generics mechanism has been used here to indicate that the `Iterator` object will be used to iterate over `String` objects only.

Table 15.2 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	None	None
`forEachRemaining`	Performs the given action for each remaining element until all elements have been processed or the action throws an exception	An implementation for the `accept` method of the `Consumer <T>` interface	None

Once an `Iterator` object has been created, a **`while`** loop can be used to iterate through the collection, with the `hasNext` method the test of the loop. The body of the loop can then retrieve items with the `next` method and, if required, delete items with the `remove` method. Let us see how this would work with the `regNums` set.

```
/* 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
    }
}
```

Within the loop we call the `next` method of our `Iterator` object to retrieve the next item within the collection. Since we have specified the `Iterator` object to retrieve `String` objects, we know that this method will return a `String`. We have stored this `String` object in a variable we have called `item`:

```
// the String returned from the Iterator object is stored in a variable
String item = elements.next();
```

It is always a good idea to store the object returned by the `next` method in a variable, as the `next` method should be called only *once* within the loop. Storing the result in a variable allows us to refer to this object as many times as we like. In this case we refer to the object only once, in the test of the **`if`** statement:

```
if (item.charAt(0)<= 'T') // check the first character in this String
```

If the registration number needs to be removed from the set, we may do so now safely—by calling the `remove` method of our `Iterator` object:

```
elements.remove(); // call Iterator method to remove current item
```

The `IteratorDemo` class gathers this code into a complete program with a few example registration numbers.

IteratorDemo

```
import java.util.Set;
import java.util.HashSet;
import java.util.Iterator;

public class IteratorDemo
{
    public static void main(String[] args)
    {
        Set<String> regNums = new HashSet<>();
        regNums.add("V53PLS");
        regNums.add("X85ADZ");
        regNums.add("L22SBG");
        regNums.add("W79TRV");
        regNums.add("E16UEL");
        System.out.println("items before removing: " + regNums);
        // create an Iterator object
        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
            }
        }
        System.out.println("items after removing: " + regNums);
    }
}
```

The program produces the expected output demonstrating the removal of one of the registration numbers that start with 'T' or earlier:

```
items before removing: [L22SBG, V53PLS, W79TRV, E16UEL, X85ADZ]
items after removing: [V53PLS, W79TRV, X85ADZ]
```

An alternative approach we could have taken is to use the `forEachRemaining` method from Table 15.2. This allows us to specify an action to be carried out over the remaining items in an `Iterator` via a lambda expression. As noted in Table 15.2, we provide an implementation for the `Consumer` interface to code this action. The `ForEachRemainingDemo` program rewrites the `IteratorDemo` program to illustrate the use of this method:

ForEachRemainingDemo

```
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class ForEachRemainingDemo
{
    public static void main(String[] args)
    {
        Set<String> regNums = new HashSet<>();
        regNums.add("V53PLS");
        regNums.add("X85ADZ");
        regNums.add("L22SBG");
        regNums.add("W79TRV");
        regNums.add("E16UEL");
        System.out.println("items before removing: " + regNums);
        Iterator<String> elements = regNums.iterator();
        // the lambda expression is applied to each remaining item in the collection
        elements.forEachRemaining ( item ->
                                    {
                                        if(item.charAt(0) <= 'T')
                                        {
                                            elements.remove();
                                        }
                                    } );
        System.out.println(regNums);
    }
}
```

Effectively, behind the scenes, the `forEachRemaining` method repeatedly calls the `hasNext` method on remaining items (in this case that is all items) and applies the given lambda expression to an item retrieved by a `next` method. The output will be the same as the output of the `IteratorDemo` before.

15.6 The *Map* Interface and the *HashMap* Class

The `Map` interface is not one of the interfaces that inherit from the general `Collections` interface. The Map interface is separate from this group and 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**, and the second 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. For example, the password of users of a network can be looked up by entering their username. Table 15.3 gives an example of such a look-up table.

We can look up the password of a user by looking up their user name in Table 15.2. The password of *lauraHaliwell*, for example, is *unicorn*, whereas the password of *wendyHarris* is *bumble*. Notice that it is important we make usernames the key of the look-up table and *not* passwords. This is because usernames are unique—no two users can have the same username. However, passwords are not unique. Two or more users may have the same password. Indeed, in Table 15.2, two users (*lauraHaliwell* and *lucyLane*) do have the same password (*unicorn*).

Let us implement this kind of look-up table using a `Map`. There are three implementations provided for the `Map` interface: `HashMap`, `TreeMap` and `LinkedHashMap`. Here we will look at the `HashMap` class.

The constructor creates the empty map:

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

Table 15.3 A look-up table for users of a network

Username	Password
lauraHaliwell	unicorn
wendyHarris	bumble
bobbyMann	elephant
lucyLane	unicorn
kabirMohan	magic

As before the type of the collection is given as the interface: `Map`. Notice that to use the generics mechanism to fix the types used in a `Map` object, we must provide *two* types in the angle 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.

15.6.1 *Map* Methods

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

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

Note that the `put` method treats the first parameter as a key item and the second parameter as its associated value. Really, we should be a bit more careful before we add user IDs and passwords into this map—only user IDs that are not already taken should be added. If we did not check this, we would end up overwriting a previous user's password. The `containsKey` method allows us to check this. This method accepts an object and returns **`true`** if the object is a key in the map and **`false`** otherwise:

```
if (users.containsKey("lauraHaliwell")) // check if ID taken
{
      System.out.println("user ID already taken");
}
else // ok to use this ID
{
      users.put("lauraHaliwell", "unicorn");
}
```

Notice we do not need to check that the password is unique as multiple users can have the same password. If we did require unique passwords the `containsValue` method could be used in the same way we used the `containsKey` method above.

Later a user might wish to change his or her password. The `put` method overrides the value associated with a key if that key is already present in the map. The following changes the password associated with "`lauraHaliwell`" to "`popcorn`":

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

The `put` method returns the value that was overwritten, or **`null`** if there was no value before, and this can be checked if necessary.

Later, a user might be asked to enter his or her ID and password before being able to access company resources. The `get` method can be used to check whether or not the correct password has been entered. 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:

```
System.out.print("enter user ID ");
String idIn = EasyScanner.nextString(); // requires EasyScanner class
System.out.print("enter password ");
String passwordIn = EasyScanner.nextString();// requires EasyScanner class
// retrieve the actual password for this user
String password = users.get(idIn);
// password will be 'null' if the user name was invalid
if (password != null)
{
       if(passwordIn.equals(password))// check password is correct
       {
          // allow access to company resources here
       }
       else // invalid password
       {
          System.out.println ("INVALID PASSWORD!");
       }
}
else // no such user
{
       System.out.println ("INVALID USERNAME!");
}
```

As you can see, once the user has entered what they believe to be their username and password, the actual password for the given user is retrieved using the `get` method.

We know the password retrieved will be of type `String` as we created our `Map` by specifying that both the keys and values of the `Map` object would be `String`s. We can then check whether this password equals the password entered by calling the `equals` method of the `String` class:

```
if(passwordIn.equals(password))
```

We have to be careful when we use the `equals` method to compare two objects in the way that we have done here. In this case, we are comparing the password entered by the user with the password obtained by the `get` method. However, the `get` method might have returned a **`null`** value instead of a password (if the key entered was invalid). The `equals` method of the `String` class does not return **`false`** when comparing a `String` with a **`null`** value, instead it throws a `NullPointerException`. So, to avoid this exception, we must check the value returned by the `get` method is not **`null`** *before* we use the `equals` method:

```
// check password returned is not 'null' before calling 'equals' method
if (password!= null)
{
   if(passwordIn.equals(password))// now it is safe to call 'equals'
   {
      // allow access to company resources
   }
   else
   {
      System.out.println ("INVALID PASSWORD!")
   }
}
```

Note that the **null** value is always checked with primitive comparison operators (== for equality and != for inequality).

Like all the other Java collection classes, the HashMap class provides a toString method so that the items in the map can be displayed:

```
System.out.print(users); // implicitly calls 'toString' method
```

Key and value pairs are displayed in braces. Let us assume we have added two more users: "bobbyMann" and "wendyHarris", with passwords "elephant" and "bumble" respectively. Displaying the map would produce the following output:

{lauraHaliwell=popcorn, wendyHarris=bumble, bobbyMann=elephant}

As with a set, the order in which the items are displayed is not determined by the order in which they were added but upon how they have been stored internally.

As with the other collections, a map provides a remove method. In the case of map, a key value is given to this method. 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:

{wendyHarris=bumble, bobbyMann=elephant}

The remove method returns the value of the object that has been removed, or **null** if the key was not present. This value can be checked if necessary to confirm that the given key was in the map.

Finally, the map collection provides size and isEmpty methods that behave in exactly the same way as the size and isEmpty methods for sets and lists.

15.6.2 Iterating Through the Elements of a Map

In order to scan the items in the map, we can use any of the methods discussed so far (the enhanced **for** loop, Iterators or the **forEach** loop).

An enhanced **for** loop cannot directly be used with a map as it is designed for collections of single values such as arrays, lists and sets. A map, however, consists of pairs of values. To use an enhanced **for** loop with a map we can extract the set of keys, using the keySet method:

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

Table 15.4 Reminder of the `BiConsumer` interface

Functional interface	Abstract method name	Parameter types	Return type
`BiConsumer<T, U>`	`accept`	`T, U`	`void`

Again notice that we know this set of keys returned by the `keySet` method will be a set of `String` objects, so we mark the type of this set accordingly.

An enhanced **`for`** loop can now be used to iterate through the keys of the map; within the loop we can look up the associated password using the `get` method. For example, we might wish to display the contents of the map in our own way, rather than the format given to us by the `toString` method:

```
for(String username: theKeys)// iterate through the set of keys
{
     String password = users.get(item); // retrieve password value
     System.out.println(username + "\t" + password); // format output
}
```

This would display the map in the following table format:

```
lauraHaliwell       popcorn
wendyHarris         bumble
bobbyMann           elephant
```

While the enhanced **`for`** loop is restricted to iterating over single values, the-**`forEach`** loop can be used with pairs of values as well as a single value, as the `forEach` method is overloaded to take a `BiConsumer` (see Table 15.4 for a reminder) as well as a `Consumer` implementation.

Here is the `forEach` implementation for displaying items in our map:

```
// the forEach loop can take two parameters, representing the key and value item of a map pair
users.forEach((username, password) -> System.out.println(username + "\t" + password));
```

You can see that the two parameters given to this `forEach` method represent the key and its associated value respectively. In this case the key is the username and its associated value is a password.

15.7 Using Your Own Classes with Java's Collection Classes

In the examples above, we stuck to the pre-defined `String` type for the type of objects used in the collection classes of Java. However, objects of any class can be used inside these collections—including objects of your own classes. Care needs to be taken, however, when using your own classes. As an example, let us consider an application to store a collection of books that a person may own.

15.7.1 The *Book* Class

Figure 15.1 gives the UML design for a `Book` class.

This class consists of an ISBN number, a title and an author. An ISBN number is a unique International Standard Book Number that is allocated to each new book title. Before we deal with a *collection* of books, here is the initial implementation of the `Book` class.

The initial Book class

```
public class Book
{
    private String isbn;
    private String title;
    private String author;

    public Book(String isbnIn, String titleIn, String authorIn)
    {
        isbn = isbnIn;
        title = titleIn;
        author = authorIn;
    }

    public String getISBN()
    {
        return isbn;
    }

    public String getTitle()
    {
        return title;
    }

    public String getAuthor()
    {
        return author;
    }
}
```

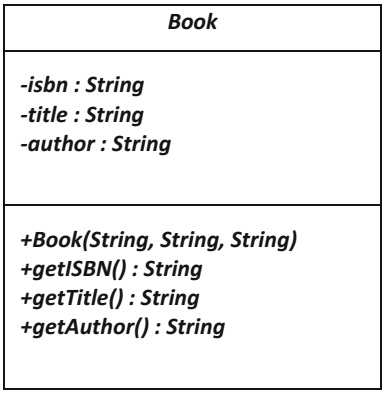


Fig. 15.1 Initial design of the *Book* class

There is nothing new here so let's move on to see how objects of this class can be stored in the Java collection classes. To begin with, let's create a list to contain `Book` objects:

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

Note once again the use of the interface type, `List`, and generics to fix the collection type to `Book`. Now let's add some book objects and display the list:

```
// create two Book objects
Book b1 = new Book("9999999999", "Clowning Around", "Joe King");
Book b2 = new Book("2222222222", "Travel With Me", "Sandy Beach");
// add Book objects to list
books.add(b1);
books.add(b2);
System.out.println(books); // implicitly call toString method of List
```

As things stand this will display something like the following list on the screen:

```
[Book@15db9742, Book@6d06d69c]
```

This isn't exactly what we require! When Java collections are displayed on the screen by calling their `toString` method, each object in the collection is displayed by calling its own `toString` method. As we said explained in Chap. 9, this will call the default `toString` method inherited from the `Object` class, and all this does is display information on the memory address of the object.

If you wish to use objects of your own classes as types in the Java collection classes, in Java's `toString` method a meaningful `toString` method should be defined in your class if you intend to make use of the toString method of the collection class.

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

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

We create a single `String` by joining the ISBN, title and author `Strings`. To improve the look of this `String` we have separated the attributes by commas and have enclosed it in round brackets. Notice the **`@Override`** annotation here, as we are overriding the `toString` method from the superclass (`Object`).

Now if we print out the list we get the following:

```
[(9999999999, Clowning Around, Joe King), (2222222222, Travel With Me, Sandy Beach)]
```

While you might wish to improve this `String` representation of a book further (maybe by adding more formatting), this is acceptable for testing purposes.

15.7.2 Defining an *equals* Method

Another issue arises if we wish to use methods such as `contains` and `remove`. Methods such as these call the `equals` method of the contained object to determine whether or not a given object is in the collection. The `Book` class does not define its own `equals` method so again the inherited `equals` method of the `Object` class is called. This method is inadequate as it simply compares the memory address of two objects rather than the attributes of two objects. For example, the following would display **`false`** on the screen when we would want it to display **`true`**:

```
// check a book that is in the list
boolean check = books.contains(new Book("9999999999", "Clowning Around", "Joe King"));
// display result
System.out.println(check);
```

Here, while the `Book` parameter has the same attribute data as one of the books in the list, it is a new `Book` object stored in a different memory location and so the `contains` method will return **`false`**.

To use objects of your own classes effectively, a meaningful `equals` method should be defined. 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:

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

Again, we have added an **`@Override`** annotation. Notice that the `equals` method must accept an item of type `Object`. The body of the method then needs to type cast this item back to the required type (`Book` in this case).

If you are using lists, these are the only two additional methods you need to provide in your class. Also, if you are using objects of your own classes only as *values* of a map, then again these are the only two additional methods you need to provide. If, however, you are using objects of your classes as keys of a map or as the items of a set, then you need to include an additional method in your classes: `hashCode`.

15.7.3 Defining a *hashCode* Method

To understand how to define your own `hashCode` method you need to understand how the `HashSet` and `HashMap` implementations work. Both of these Java classes have been implemented using an array. Unlike the `ArrayList` class

however (which has also been implemented using an array), the position of the items in the `HashSet` and `HashMap` arrays is not determined by the order in which they were added. Instead, the position into which items are added into these arrays is determined by their `hashCode` method.

The `hashCode` method returns an integer value from an object. This integer value determines where in the array the given object is stored. Objects that are equal (as determined by the `equals` method) should produce identical `hashCode` numbers and, ideally, objects that are not equal should return different `hashCode` numbers.

The reason for using `hashCode` numbers is that they considerably reduce the time it takes to search a given array for a given item. If items were stored consecutively, then a search of the array would require *every* item in the array being checked using the `equals` method, until a match was found. This would become very inefficient when the collection becomes very large. So, instead, the `HashSet` and `HashMap` classes make use of the `hashCode` method, so that when a search is required for an item, say `x`, that `hashCode` number is computed and this value is used to look up other items in the array. Then, only objects with the same `hashCode` number are compared to `x` using their `equals` method.

If you are using objects of your own classes, and you have not provided a `hashCode` method, the inherited `hashCode` method from the `Object` class is called. This does not behave in the way we would wish. It generates the `hashCode` number from the memory address of the object, so two "equal" objects could have different `hashCode` values. The `TestHashCode` program demonstrates this.

TestHashCode

```
public class TestHashCode
{
   public static void main (String[] args)
   {
      // create two "equal" books
      Book b1 = new Book("9999999999", "Clowning Around", "Joe King");
      Book b2 = new Book("9999999999", "Clowning Around", "Joe King");
      // check their hashCode numbers
      System.out.println(b1.hashCode());
      System.out.println(b2.hashCode());
   }
}
```

When we run this program it produces the following `hashCode` numbers for the two "equal" books:

```
1044036744
1826771953
```

As you can see, the `hashCode` numbers do not match—even though the objects are "equal". This means that if we were searching for the given book in a `HashSet` or in the keys of a `HashMap`, the book would not be found, as only objects with identical `hashCode` values are checked. Also, we cannot ensure that objects in the `HashSet` or the keys of the `HashMap` will be unique, as two (or more) identical books (with different `hashCode` values) would both be stored in the underlying array at different array positions.

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.

Luckily, all of Java's predefined classes (such as `String`) have a meaningful `hashCode` method defined. These `hashCode` methods return equal `hashCode` numbers for "equal" objects.

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`:

```
@Override
// 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();
}
```

If you have more than one attribute that plays a role in determining object equality, then add each such `hashCode` numbers (assuming the attribute is not of primitive type) to determine your `hashCode` number.

If primitive attributes also play a part in object equality, they too can simply be added into your `hashCode` formula by generating an integer value from each primitive attribute. Here is a simple set of guidelines for generating an integer value from the primitive types (although much more sophisticated algorithms exist than these!):

- **`byte`**, **`short`**, **`int`**, **`long`**: leave as they are;
- **`float`**, **`double`**: type cast to an integer;
- **`char`**: use its Unicode value;
- **`boolean`**: use an **`if...else`** statement to allocate 1 if the attribute is **`true`** and 0 if it is **`false`**.

15.7.4 The Updated *Book* Class

The complete code for the updated `Book` class is now presented below with `toString`, `equals` and `hashCode` methods included:

The updated Book class

```
public class Book
{
    private String isbn;
    private String title;
    private String author;

    public Book(String isbnIn, String titleIn, String authorIn)
    {
       isbn = isbnIn;
       title = titleIn;
       author = authorIn;
    }

    public String getISBN()
    {
       return isbn;
    }

    public String getTitle()
    {
       return title;
    }

    public String getAuthor()
    {
       return author;
    }

    @Override
    public String toString()
    {
       return "(" + isbn +", " + title + ", " + author +  ")";
    }

    @Override
    public boolean equals (Object objIn)
    {
       Book bookIn = (Book) objIn; // type cast to a Book
       // check isbn
       return  isbn.equals(bookIn.isbn);
    }

    @Override
    public int hashCode()
    {
       // derive hash code by returning hash code of ISBN string
       return isbn.hashCode();
    }
}
```

15.8 Developing a Collection Class for *Book* Objects

In the previous section we amended the `Book` class by supplying it with a `toString` method, an `equals` method and a `hashCode` method. If you look at the documentation of any class in the Java API you will see that it also possesses these three methods. When developing classes professionally you should always include these methods. That way, objects of these classes can be used with any collection type.

Now we have a suitable `Book` class we can store objects of this class in one of the Java collection classes. Which collection shall we use? There is no ordering required on the collection of books so we do not really need a list here. We could store these books in a set, but since each book has a unique ISBN it makes more

sense to use a map and have ISBNs as the keys to the map, with `Book` objects themselves as the values of the map.

We will use this collection to help us develop our own class, `Library`, which keeps track of the books that a person may own. Figure 15.2 gives its UML design.

Notice that the keys of the map are specified to be `String` objects (to represent ISBNs), and the values of the map are specified as `Book` objects.

Here is the implementation of the `Library` class. Take a look at it and then we will discuss it.

Library

```
import java.util.Map;
import java.util.HashMap;
import java.util.Set;
import java.util.HashSet;


public class Library
{
   Map <String, Book> books; // declare map collection
   // create empty map
   public Library()
   {
      books = new HashMap<> ();
   }

   // add the given book into the collection
   public boolean addBook(Book bookIn)
   {
      String keyIn = bookIn.getISBN(); // isbn will be key of map
      if (books.containsKey(keyIn)) // check if isbn already in use
      {
         return false; // indicate error
      }
      else // ok to add this book
      {
         books.put(keyIn, bookIn); // add key and book pair into map
         return true;
      }
   }

   // remove the book with the given isbn
   public boolean removeBook(String isbnIn)
   {
      if (books.remove(isbnIn)!= null) // check if item was removed
      {
         return true;
      }
      else // when item is not removed
      {
         return false;
      }
   }

   // return the number of books in the collection
   public int getTotalNumberOfBooks()
   {
      return books.size();
   }

   // return the book with the given isbn or null if no such book
   public Book getBook (String isbnIn)
   {
      return books.get(isbnIn);
   }

   // return the set of books in the collection
   public Set<Book> getAllBooks ()
   {
      Set<Book> bookSet = new HashSet<>(); // to store the set of books
      books.forEach((key,book) -> bookSet.add(book)); // use forEach loop to add books
      return bookSet; // return the set of books
   }
}
```

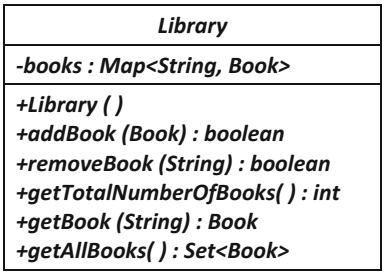


Fig. 15.2 UML design for the *Library* class

Most of this class should be self-explanatory. Just consider how complicated some of these methods would have been if we had used an array instead of a map!

We just draw your attention to the `getAllBooks` method. The UML diagram indicates that this method should return a *`set`* of books. An alternative approach could have been to return the map itself, but it would be more useful for this method to return a set of objects, as a set is easier to scan than a map. It is fine to use a set as there is no repetition of books.

```
public Set<Book> getAllBooks ()
{
  // code to create this set goes here
}
```

There is no map method that returns the set of values in the map directly,[3] so a suitable set needs to be created in this method. We begin by creating an empty set:

```
Set<Book> bookSet = new HashSet<>();
```

This set is empty initially and we must fill it with book objects. In order to access these values we use a **`forEach`** loop and simply add the values of the map into this set.

```
books.forEach((key,book) -> bookSet.add(book)); // use forEach loop to add books
```

Finally, we return this set of books:

```
return bookSet;
```

That completes our discussion of the `Library` class. We leave the task of creating a tester for this class to the end of chapter exercises.

[3]There are several approaches to indirectly create a set of values in a map. Here we look at just one approach.

15.9 Sorting Objects in a Collection

We have seen how we can use implementations of the `List` interface (such as `ArrayList`) to create an ordered collection in our programs. Previously we have seen how arrays can also be used to store an ordered collection. There may be times when you wish to sort these collections so they are presented in a different order. For example, you may have an ordered collection of students and wish to re-order these students based on their student ID, or their final mark.

15.9.1 The *Collections.sort* and *Arrays.sort* Methods

Sorting a collection can involve complex algorithms. Luckily Java provides two classes that contain class methods to sort your ordered collections. The `Collections` class and the `Arrays` class both have a number of utility methods for processing collections in the JCF and arrays respectively. In particular, both classes contain a `sort` method to sort a given list or array. Both classes are in the `java.util` package. The `StringSortDemo` program below demonstrates how to use these methods with an ordered collection of `String` objects. It introduces a few new additional concepts. Take a look at it and then we will discuss it:

StringSortDemo

```
import java.util.Collections;
import java.util.Arrays;
import java.util.List;

public class StringSortDemo
{
    public static void main(String[] args)
    {
        // create array of strings
        String[] citysArray = {"London", "Birmingham", "Manchester", "Liverpool"};
        // display array using Arrays.toString
        System.out.println("Original Array " + Arrays.toString(citysArray));
        // convert array to List using Arrays.asList
        List<String> citysList = Arrays.asList(citysArray);
        // display List
        System.out.println("Original List " + citysList);
        // sort array
        Arrays.sort(citysArray);
        // display array
        System.out.println("Sorted Array " + Arrays.toString(citysArray));
        // sort List
        Collections.sort(citysList);
        // display List
        System.out.println("Sorted List " + citysList);
     }
}
```

First you can see we have gone back to using an array type to store a collection `String` objects:

```
// create array of strings
String[] citysArray = {"London", "Birmingham", "Manchester", "Liverpool"};
```

The `Arrays` utility class has a `toString` class method that allows us to convert an array into a `String` which may then be displayed on the screen:

```
// display array using toString
System.out.println("Original Array " + Arrays.toString(citysArray));
```

Next we use the `asList` method of `Arrays` that returns an equivalent `List` object containing the same items in the collection, in the same order—we store this in a new `List` object, which we then display:

```
// convert array to List using asList
List<String> citysList = Arrays.asList(citysArray);
// display List
System.out.println("Original List " + citysList);
```

Now the really useful `sort` methods that re-order the given collection of strings so they are sorted in ascending alphabetical order:

```
// sort array
Arrays.sort(citysArray);
// display array using toString
System.out.println("Sorted Array " + Arrays.toString(citysArray));
// sort List
Collections.sort(citysList);
// display List using toString
System.out.println("Sorted List " + citysList);
```

You can see that in order to sort an array we use the `Arrays.sort` method and to sort a `List` we use the `Collections.sort` method. Here is the output from this program:

```
Original Array [London, Birmingham, Manchester, Liverpool]
Original List [London, Birmingham, Manchester, Liverpool]
Sorted Array [Birmingham, Liverpool, London, Manchester]
Sorted List [Birmingham, Liverpool, London, Manchester]
```

As expected the `sort` methods re-order the items in the collection so that they are now in ascending lexicographical order. Note that the `Arrays.toString` method returns a `String` representation of the array in the same format as a `List` (i.e. with items separated by commas and enclosed in square brackets).

For these `sort` methods to work the contained object needs to be derived from a class that implements the generic `Comparable<T>` interface.

15.9.2 The *Comparable<T>* Interface

The generic `Comparable<T>` interface consist of a single `compareTo` method that accepts an object of type `T` and returns an integer (see Table 15.5).

A `compareTo` method should return a positive integer when the first object is greater than the second, a negative integer when it is less than the second and zero when the two objects are equal. As we saw in Chap. 7, the `String` class does have such a method defined and so implements the `Comparable<T>` interface.

But what happens if we try to sort a collection of objects that do not have a `compareTo` method defined? Well, the `Collections.sort` method would result in a compiler error, whereas the `Arrays.sort` method would not give a compiler error but would throw an exception at runtime.

To sort ordered collections of objects derived from classes that you have defined you will need to ensure their associated class implements the `Comparable<T>` interface.

As an example, let's return to our `Book` class. We would be unable to use the `sort` methods we have met to sort lists or arrays of `Book` objects as the `Book` class does not implement the `Comparable<T>` interface. To implement this interface we need to define a suitable `compareTo` method. One way of comparing two `Book` objects might be just to compare their ISBN numbers. Since ISBN numbers have been stored as `Strings`, we just need to use the `compareTo` method of `String` and use that result as our `Book` `compareTo` result. Here is the code:

A sortable Book class

```
// this class can be used in conjunction with Collections.sort and Arrays.sort

public class Book implements Comparable <Book> // notice generic type fixed to Book
{
    //attributes and methods as before


   // add a compareTo method

   @Override
    public int compareTo(Book bIn)
    {
         return isbn.compareTo(bIn.isbn); // compare ISBN numbers
    }
}
```

Notice how we use the generics mechanism to indicate the type of object the `Comparable` interface will be fixed to. In this case it is set to `Book` as we wish to compare two books.

Table 15.5 The `Comparable<T>` interface

Functional interface	Abstract method name	Parameter types	Return type
`Comparable<T>`	`compareTo`	`T`	`int`

The BookSortDemo1 program below makes use our updated Book class to sort a list of Book objects.

BookSortDemo1

```
import java.util.Collections;
import java.util.List;
import java.util.ArrayList;

public class BookSortDemo1
{
    public static void main(String[] args)
    {
       // create three Book objects
       Book b1 = new Book("9999999999", "Clowning Around", "Joe King");
       Book b2 = new Book("2222222222", "Travel With Me", "Sandy Beach");
       Book b3 = new Book("4444444444", "Interior Design", "Anita Room");
       // create an empty list of books
       List<Book> bookList = new ArrayList<>();
       // add the three Book ojects
       bookList.add(b1);
       bookList.add(b2);
       bookList.add(b3);
       // display list before and after sorting
       System.out.println("***COMPARABLE DEMO***");
       System.out.println("\nBefore Sort\n"+bookList);
       Collections.sort(bookList); // uses the Book compareTo method
       System.out.println("\nAfter sort\n"+ bookList);
    }
}
```

Here is the output from this program:

```
***COMPARABLE DEMO***

Before Sort
[(9999999999, Clowning Around, Joe King), (2222222222,
Travel With Me, Sandy Beach), (4444444444, Interior Design,
Anita Room)]

After sort
[(2222222222, Travel With Me, Sandy Beach), (4444444444,
Interior Design, Anita Room), (9999999999, Clowning Around,
Joe King)]
```

As you can see the Book objects have been sorted on ISBN numbers.

15.9.3 The *Comparator<T>* Interface

The previous section demonstrated one technique for comparing objects for sorting purposes; implement the Comparable<T> interface. This technique is fine if we always wish to use the same criteria for sorting objects; in our case if we always wish to sort Book objects by their ISBN number.

But what if you wish to sort your collection by some other criteria, or multiple criteria? For example, we may wish to sort our list of Book objects via their title, at other times we might wish to sort via the author name. The Comparable<T>

Table 15.6 The `Comparator` interface

Functional interface	Abstract method name	Parameter types	Return type
`Comparator<T>`	`compare`	`T, T`	`int`

interface only allows us to define a single criteria for sorting purposes. But since Java 8 a more flexible way of doing this with the `Comparator<T>` interface has been introduced.

The `Comparator<T>` interface requires us to provide an implementation for a `compare` method (see Table 15.6).

The `compare` method accepts two objects of a given type and (as with the `compareTo` method) returns a positive integer indicating the first object is greater, or a negative integer if the first object is less than the second object, or zero if both objects are equal.

The `List` interface contains a **`default`** method[4] called `sort` that takes a single parameter of type `Comparator`. Since this is a **`default`** method of `List` it is available to `ArrayList` objects. `BookSortDemo2` makes use of this `sort` method to sort the books based on author names. Take a look at it and then we will discuss it.

BookSortDemo2

```
import java.util.List;
import java.util.ArrayList;

public class BookSortDemo2
{
    public static void main(String[] args)
    {
       // create three Book objects
       Book b1 = new Book("9999999999", "Clowning Around", "Joe King");
       Book b2 = new Book("2222222222", "Travel With Me", "Sandy Beach");
       Book b3 = new Book("4444444444", "Interior Design", "Anita Room");
       // create an empty list of books
       List<Book> bookList = new ArrayList<>();
       // add the three Book ojects
       bookList.add(b1);
       bookList.add(b2);
       bookList.add(b3);
       // display list before and after sorting
       System.out.println("***COMPARATOR DEMO***");
       System.out.println("\nBefore Sort\n"+bookList);
       // sort list using sort method of List and a Comparator implementation using lambda
       bookList.sort((book1,book2)  ->  {return  book1.getAuthor().compareTo(book2.getAuthor());});
       System.out.println("\nAfter Author sort\n"+ bookList);
    }
}
```

The key instruction here is call to `bookList.sort()`. You can see that this accepts an implementation of a `compare` method. The `compare` method takes two parameters representing two books that need to be compared. The given

[4]Remember from Chap. 13, **`default`** methods were added in Java 8 and are fully implemented methods within an interface.

lambda expression uses the author field of the two `Book` objects in order to compare them (using the `compareTo` method of `String`):

```
// note the lambda expression to provide a Comparator implementation
bookList.sort((book1,book2) -> {return book1.getAuthor().compareTo(book2.getAuthor());});
```

When we run this program we get the expected output:

```
***COMPARATOR DEMO***

Before Sort
[(9999999999, Clowning Around, Joe King), (2222222222,
Travel With Me, Sandy Beach), (4444444444, Interior Design,
Anita Room)]

After Author sort
[(4444444444, Interior Design, Anita Room), (9999999999,
Clowning Around, Joe King), (2222222222, Travel With Me,
Sandy Beach)]
```

If we wished to use another attribute for comparison, such as the title, we could replace the call to `getAuthor` with a call to `getTitle` (for example). In each case we could use the `compareTo` method to compare the appropriate attribute which acts as the key to our sort.

This pattern of picking a sort key (ISBN, author, title etc.) and then comparing two keys using `compareTo` is a very common approach to implementing a `Comparator` and a **`static`** `comparing` method of `Comparator` is provided in order to simplify this task. As mentioned in Chap. 13, since Java 8 interfaces can contain **`static`** methods.

The `comparing` method is provided with an appropriate sort key via a suitable method (`getAuthor` for the author key, `getTitle` for the title key and so on). Behind the scenes the `comparing` method then compares two keys for us using their `compareTo` method. The appropriate key can be sent to the `comparing` method via the method reference (double colon) notation we discussed in Chap. 13. `BookSortDemo3` illustrates how we can sort via all three attribute keys of a book (ISBN, author and title) using this technique:

BookSortDemo3

```
import java.util.List;
import java.util.ArrayList;
import java.util.Comparator;

public class BookSortDemo3
{
    public static void main(String[] args)
    {
       // create three Book objects
       Book b1 = new Book("9999999999", "Clowning Around", "Joe King");
       Book b2 = new Book("2222222222", "Travel With Me", "Sandy Beach");
       Book b3 = new Book("4444444444", "Interior Design", "Anita Room");
       // create an empty list of books
       List<Book> bookList = new ArrayList<>();
       // add the three Book ojects
       bookList.add(b1);
       bookList.add(b2);
       bookList.add(b3);
       // display list before and after sorting
       System.out.println("***COMPARATOR DEMO***");
       System.out.println("\nBefore Sort\n"+bookList);

       // sort list using getISBN method reference
       bookList.sort(Comparator.comparing(Book::getISBN));
       System.out.println("\nAfter ISBN sort\n"+ bookList);
       // sort list using getAuthor method reference
       bookList.sort(Comparator.comparing(Book::getAuthor));
       System.out.println("\nAfter Author sort\n"+ bookList);
       // sort list using the getTitle method reference
       bookList.sort(Comparator.comparing(Book::getTitle));
       System.out.println("\nAfter Title sort\n"+ bookList);
    }
}
```

Here is the expected output:

```
***COMPARATOR DEMO***

Before Sort
[(9999999999, Clowning Around, Joe King), (2222222222,
Travel With Me, Sandy Beach), (4444444444, Interior Design,
Anita Room)]

After ISBN sort
[(2222222222, Travel With Me, Sandy Beach), (4444444444,
Interior Design, Anita Room), (9999999999, Clowning Around,
Joe King)]

After Author sort
[(4444444444, Interior Design, Anita Room), (9999999999,
Clowning Around, Joe King), (2222222222, Travel With Me,
Sandy Beach)]

After Title sort
[(9999999999, Clowning Around, Joe King), (4444444444,
Interior Design, Anita Room), (2222222222, Travel With Me,
Sandy Beach)]
```

The comparing method will work on numeric attributes too (such as **double**s, **int**s and **long**s) via the compareTo method provided in the equivalent wrapper classes. However, to reduce the overhead of boxing and unboxing primitive values you can use specific comparing methods for these types (comparingDouble, comparingInt, comparingLong etc.). You will see an example of this in Chap. 22.

It is worth pointing out that you can always revert to the natural ordering key defined in your own compareTo method by using the **static** naturalOrder method of Comparator.

We used ISBNs as our built-in comparison key when writing the compareTo method of Book, so instead of the following line from BookDemo4:

```
bookList.sort(Comparator.comparing(Book::getISBN));
```

we could have used the naturalOrder method instead as follows:

```
bookList.sort(Comparator.naturalOrder());
```

Note that if the underlying list contains strings or numeric values only, naturalOrder will sort the given list in alphabetical or numerical order respectively.

To find out more about the Collections.sort, Arrays.sort and the sort method of List, as well as more about the Comparable<T> and Comparator<T> interfaces, you can refer to the Oracle™ site.

15.10 Self-test Questions

1. Distinguish between the following types of collection in the Java Collections Framework:
   - List;
   - Set;
   - Map.
2. Identify, with reasons, the most appropriate Java collection type to implement the following collections:
   (a) An ordered collection of patient names waiting for a doctor;
   (b) An unordered collection of patient names registered to a doctor;
   (c) An unordered collection of BankAccount objects.

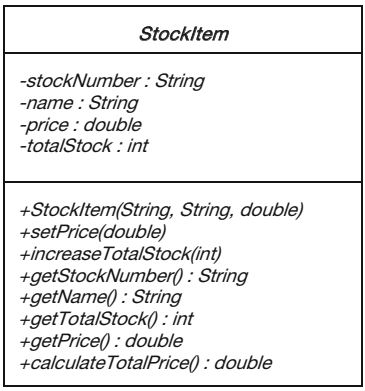
3. Consider the following instruction:

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

   (a) Why is the type of this object given as `Map` and not `HashMap`?
   (b) Assuming the object `javaStudents` has been created as above, why would the following line cause a compiler error?

```
javaStudents.put("u0012345", "Jeet");
```

4. Consider again the `StockItem` class from the programming exercises of Chap. 8. Here is the UML diagram:



   (a) Identify the purpose of a `toString` method and define an appropriate `toString` method for this class.
   (b) Identify the purpose of an `equals` method and define an appropriate `equals` method for this class.
   (c) Identify the purpose of a `hashCode` method and define an appropriate `hashCode` method for this class.

5. In Sect. 15.5 a set called `regNums` was created to store a collection of car registration numbers.

   (a) Write a fragment of code that makes use of an enhanced **`for`** loop to display all registrations ending in 'S'.
   (b) Write a fragment of code that makes use of a **`forEach`** loop to display all registrations ending in 'S'.
   (c) Write a fragment of code, which makes use of the `iterator` method, to remove all registration numbers ending in 'S'.

6. In Chap. 8 we introduced a `BankAccount` class and a collection class to hold bank accounts called `Bank`. The `Bank` class was implemented using a `List` in the form of an `ArrayList` class.

   (a) If we were to change from a `List` to a `Map`, what would be the key type of the `Map` and what would be the value type?
   (b) Write a fragment of code that declares a `Map` to hold `BankAccount` objects and add two `BankAccount` objects into this map.
   (c) Write a fragment of code that uses a **`forEach`** loop to display all `BankAccount` objects in the map that have a balance over 100.

7. Consider the `BankAccount` class from Chaps. 7 and 8 once again. Assume we have the following array to store a collection of `BankAccount` objects:

```
BankAccount[] accountList = new BankAccount[5];
```

   Now assume five `BankAccount` objects have been added into this list.

   (a) What method of the `Arrays` class would allow you to display this array?
   (b) What method of the `Arrays` class would you allow you to create an equivalent `List` from this array?
   (c) Assuming the array has been converted to a list, describe how the `Comparable<T>` and `Comparator<T>` interfaces can be used to help sort the list, via the account number, by making use of the `Collections.sort` method and the `sort` method of `List`.

15.11 Programming Exercises

1. Copy, from the accompanying website, the `Book` and `Library` classes and then implement a tester for the `Library` class.
2. Make the changes to the `StockItem` class you considered in self-test question 4 above, then:

   (a) Write a tester that adds five `StockItem` objects into a set.
   (b) Modify the tester so that the set is displayed using the `toString` method of `Set`.
   (c) Use the `contains` method of `Set` method to search for one of the stock items.

3. Copy, from the accompanying website, the `Book` and `BookSortDemo2` classes and then:

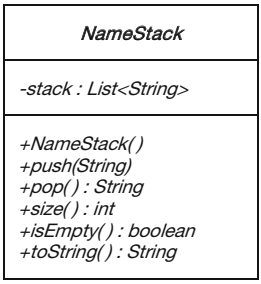
   (a) Rewrite the `compareTo` method (that implements the `Comparable<Book>` interface) in the `Book` class so the length of the title is used to compare two books. Run the `BookSortDemo4` program to test this.
   (b) Modify the `BookSortDemo4` program to sort the list of books based on length of title by using both the `comparing` method and `naturalOrdering` methods of `Comparator`.

4. Write a tester program to test your answers to self-test question 7 above.

5. In this chapter we looked at an example of a printer queue. A *queue* is a collection where the first item added into the queue is the first item removed from the queue. Consequently, a queue is often referred to as a *first in first out* (FIFO) collection.

   A *stack*, on the other hand, is a *last in first out* (LIFO) collection—much like a stack of plates, where the last plate added to the stack is the first plate removed from the stack. The method to add an item onto a stack is often called *push*. The method to remove an item from a stack is often called *pop*.

   Below, we give the UML design for a `NameStack` class, which stores a stack of names.



   A description of each `NameStack` method is given below:

   `+NameStack()`
   Initializes the stack of names to be empty.

   `+push(String)`
   Adds the given name onto the top of the stack.

   `+pop() : String`
   Removes and returns the name at the top of the stack. Throws a `NameStackException` (which will also need to be implemented) if an attempt is made to pop an item from an empty stack.

+size() : int
Returns the number of names in the stack.

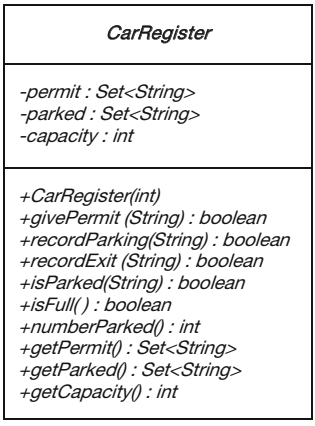
+isEmpty() : boolean
Returns **true** if the stack is empty and **false** otherwise.

+toString() : String
Returns a `String` representation of the stack of names.

(a) Implement the `NameStack` class. You will also need to implement a `NameStackException` class.

(b) Test the `NameStack` class with a suitable tester.

6. Consider an application that keeps track of the registration numbers of all cars that have a permit to use a company car park. It also keeps track of the registration numbers of the cars actually in the car park at any one time. While there is no limit to the number of cars that can have permits to park in the car park, the capacity of the car park is limited. Below we give the UML design for the `CarRegister` class:



A description of each `CarRegister` method is given below:

+CarRegister(int)
Initializes the `permit` and `parked` sets to be empty and sets the capacity of the car park with the given parameter. Throws a `CarRegisterException` (which will also need to be implemented) if the given parameter is negative.

+givePermit(String) : boolean
Records the registration of a car given a permit to park. Returns **false** if the car has already been given a permit and **true** otherwise.

+recordParking(String) : boolean
Records the registration of a car entering the car park. Returns **false** if the car park is full, or the car has no permit to enter the car park, and **true** otherwise.

+recordExit (String) : boolean
Records the registration of a car leaving the car park. Returns **false** if the car was not initially registered as being parked and **true** otherwise.

+isParked(String) : boolean
Returns **true** if the car with the given registration is recorded as being parked in the car park and **false** otherwise.

+isFull() : boolean
Returns **true** if the car park is full and **false** otherwise.

+numberParked() : int
Returns the number of cars currently in the car park.

+getPermit() : Set<String>
Returns the set of car registrations allocated permits.

getParked() : Set<String>
Returns the set of registration numbers of cars in the car park.

+getCapacity() : int
Returns the maximum capacity of the car park.

(a) Implement the `CarRegister` class. You will also need to implement a `CarRegisterException` class.
(b) Test the `CarRegister` class with a suitable tester.

7. Re-write the `Bank` class of Chap. 8 so that it makes use of a `Map` instead of a `List`, as discussed in self-test question 6 above.