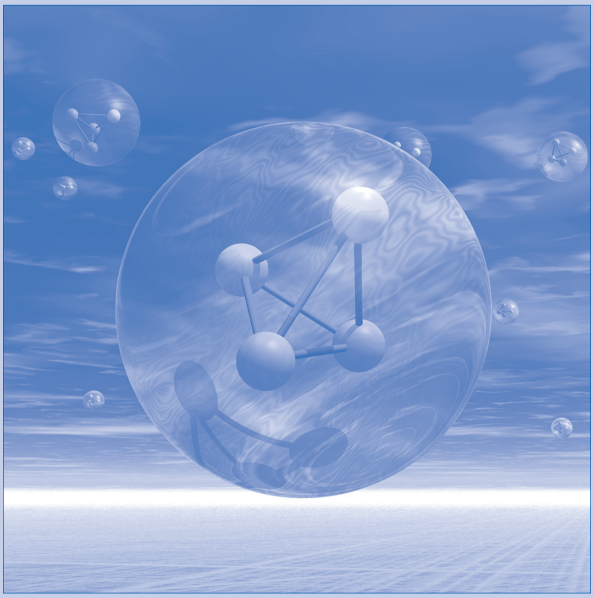
M255 Unit 11

UNDERGRADUATE COMPUTING

Object-oriented programming with Java



Ordered and sorted collections

Unit 11

This publication forms part of an Open University course M255 *Object-oriented programming with Java*. Details of this and other Open University courses can be obtained from the Student Registration and Enquiry Service, The Open University, PO Box 197, Milton Keynes, MK7 6BJ, United Kingdom: tel. +44 (0)870 333 4340, email general-enquiries@open.ac.uk

Alternatively, you may visit the Open University website at http://www.open.ac.uk where you can learn more about the wide range of courses and packs offered at all levels by The Open University.

To purchase a selection of Open University course materials visit http://www.ouw.co.uk, or contact Open University Worldwide, Michael Young Building, Walton Hall, Milton Keynes, MK7 6AA, United Kingdom for a brochure: tel. +44 (0)1908 858785; fax +44 (0)1908 858787; email ouwenq@open.ac.uk

The Open University
Walton Hall
Milton Keynes
MK7 6AA

First published 2006. Second edition 2008.

Edited and designed by The Open University.

Typeset by The Open University.

Printed and bound in the United Kingdom by The Charlesworth Group, Wakefield.

ISBN 978 0 7492 6792 6

2.1

CONTENTS

M255 COURSE TEAM

Affiliated to The Open University unless otherwise stated.

Rob Griffiths, Course Chair, Author and Academic Editor

Lindsey Court, Author

Marion Edwards, Author and Software Developer

Philip Gray, External Assessor, University of Glasgow

Simon Holland, Author

Mike Innes, Course Manager

Robin Laney, Author

Sarah Mattingly, Critical Reader

Percy Mett, Academic Editor

Barbara Segal, Author

Rita Tingle, Author

Richard Walker, Author and Critical Reader

Robin Walker, Critical Reader

Julia White, Course Manager

Ian Blackham, Editor

Phillip Howe, Compositor

John O'Dwyer, Media Project Manager

Andy Seddon, Media Project Manager

Andrew Whitehead, Graphic Artist

Thanks are due to the Desktop Publishing Unit, Faculty of Mathematics and Computing.

Introduction

In the first part of this unit we study an important new interface from the **Java Collections Framework**. The interface is `List` and Figure 1 shows how it relates to the interface hierarchy we met in *Unit 10*.

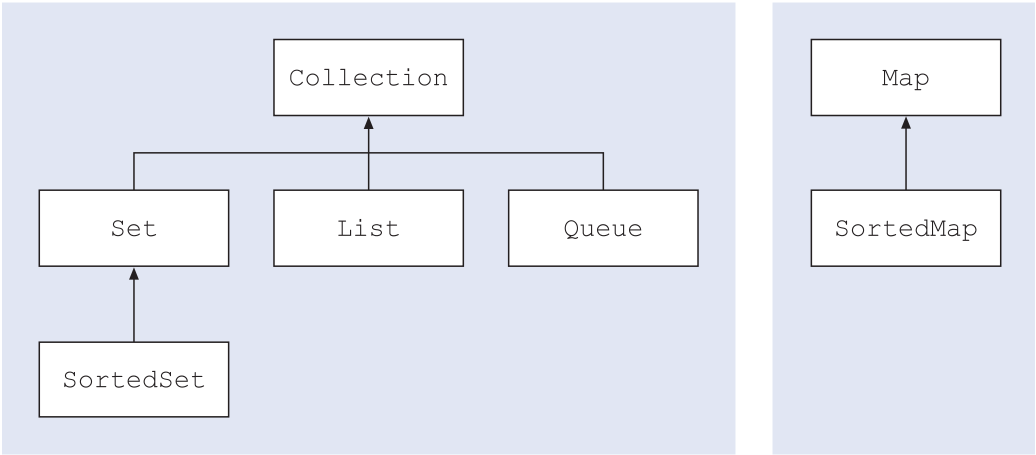


Figure 1 The principal interfaces of the Collections Framework

A list in the programming sense is roughly similar to a list in the everyday sense of the word. It holds a sequence of items, in a definite order, for example:

d, b, d, a, m, e, d, w.

The list above has a size – eight at the moment – and each item has a position in the list, starting with d at position 0. Items can be added (which means the size changes):

d, b, d, a, m, e, d, w, **q**.

They can be removed:

d, b, a, m, e, d, w, q.

And they can be inserted anywhere:

d, b, a, m, e, **p**, d, w, q.

Repetitions are allowed. The items in the list may have a **natural ordering** – alphabetical in this case – but the sequence followed by the items in the list does not have to follow it.

This is only an outline, of course, and the first business of the unit will be to examine the properties of lists in much more detail. We will then go on to look at a very versatile utility class called `Collections`, which is a treasure store of class (static) methods for doing things to collections, such as sorting or shuffling them, picking out the biggest or smallest item, and so on. Note the '`s`' in the name of the `Collections` class – it should not be confused with the `Collection` interface.

Sorting the items in a list is an important task that we as programmers will often need to do, and in Section 2 we investigate how the classes we write can be given a natural ordering, so instances of those classes can be sorted.

We shall see that there are **collection classes** which are always sorted – their contents are always held in natural order, and new items are automatically added at the right position to preserve this order. These classes implement the interfaces `SortedSet` and `SortedMap` shown in Figure 1.

Finally we look at what it means to say that two objects are equal. This is important because equality is what decides what counts as a duplicate in sets and maps.

1 List – an ordered and indexed collection

The set of criteria used for differentiating different kinds of collection, introduced at the beginning of *Unit 10*, provides a good way of gaining an initial impression of what lists can do and what they are 'about'.

- **Fixed size** versus **dynamic:** Lists are *dynamically sized.* A list will grow and shrink as elements are added or removed.
- **Ordering:** Lists are *ordered.* A list can be grown simply by tagging items onto the end, in which case the elements will be in the order they were added. Alternatively an item can be inserted at a specified position.
- **Homogeneity:** Like any Java collection, a list is homogeneous – all the elements must be of the same type.
- **Indexing:** Every element in a list has an integer index 0, 1, 2, 3, ... to indicate its position. The index provides a way of directly accessing the element.

  If an item is inserted in the middle of a list then all the elements beyond it will be moved up to make space for it, so their indexes will increase by 1. If an item is removed from a list, all the elements beyond it will have their indexes decreased by 1, so the gap closes up.
- **Duplicates:** Unlike sets, which do not allow duplicates, a list can contain any number of repeated elements.

`List` is a subinterface of `Collection`

In *Unit 10* we met the `Collection` interface and looked at the abstract methods it defined. For convenience we have repeated the details in Table 1 below. Note that we have given longer and more informative names to the formal arguments than you will find in the Java documentation.

Table 1 Some methods of the `Collection` interface

Method	Category	Description
`add(ElementType obj)`	adding	Adds the argument to the collection. Returns `true` if the operation is successful, `false` otherwise.
`addAll(Collection aCol)`	adding	Adds all the elements in the argument to the receiver. Returns `true` if the operation is successful, `false` otherwise.
`remove(Object obj)`	removing	Removes one occurrence of the argument from the collection. Returns `true` if the operation is successful, `false` otherwise.
`clear()`	removing	Removes all elements from the collection (if there are any).
`size()`	testing	Returns the number of elements in the collection.
`contains(Object obj)`	testing	Tests whether the argument is present in the collection. Returns `true` if the argument is an element of the collection, `false` otherwise.
`isEmpty()`	testing	Tests whether the collection is empty. Returns `true` if the collection is empty, `false` otherwise.

In Figure 1 we see that `List` is a subinterface of `Collection`, and so we can tell straight away that any list must implement all the methods in Table 1 and honour the promises made there. So we already know quite a lot about lists! This is an excellent example of how understanding the interfaces of the Collections Framework gives a good insight into what particular collection classes are going to be capable of, even before we have studied them in detail.

Of course a class that implements `List` may do *more* than Table 1 promises and, as we shall see, this is the case with the class `ArrayList`, which we introduce shortly.

1.1 Getting started with lists

In *Unit 10* we saw that to use a **collection class** we have to do three things:

- decide the interface type;
- choose an implementation class;
- declare an element type.

Here the interface type is `List` and, as we discussed in *Unit 10*, best practice is to use a variable declared as being of the interface type, because that gives us flexibility to change our mind later about what implementation class we will use.

A good general-purpose choice for a class that implements the `List` interface is `ArrayList` and this is the one to use unless we know of a particular reason for using another list class.

The element type is, as we have seen, the type of the elements that the collection will be allowed to hold, and is declared using angle brackets < >.

Putting this all together, if we wanted a list to hold instances of class `HoverFrog` we could use a statement such as the following:

```
List<HoverFrog> hoverFrogList = new ArrayList<HoverFrog>();
```

This declaration follows the same pattern as the declarations we made for sets in *Unit 10*.

1.2 Basic operations with lists

Getting the size of a list

As with other collections you have seen, one of the simplest things you can do with a list is to get its size. The size of a list is simply the number of elements it contains. You can get the size of a list by sending it the message `size()` in the same way as we saw `size()` being used with sets and maps in *Unit 10*. As you might expect, the size of newly created empty list is `0`. Our first example illustrates this.

Note that the elements of a list may include `null`; if so, it will be counted in the size.

```
List<String> herbList = new ArrayList<String>();
int size = herbList.size();
System.out.println("Size of newly created list is: " + size);
```

Notice that as in *Unit 10* we have declared the variable as being of an interface type, and we have to declare the element type of the collection.

Adding an element to a list

To do anything interesting with a list we need to add some elements to it. Elements can be added one at a time to a list using an `add()` message. When we add an element it is always put at the end of the list unless we specify a position.

The code below demonstrates the use of `add()`. It adds three string elements to an initially empty set, making the size `3`. The elements are stored in the order they are added, beginning with index `0`.

```
List<String> herbList = new ArrayList<String>();
herbList.add("Parsley");
herbList.add("Sage");
herbList.add("Rosemary");
```

Lists may contain duplicate elements

Unlike a set, a list can contain duplicate elements, as the code below demonstrates. Five elements are added to the list, but three of those elements are duplicates. The size of the list is nevertheless `5`.

```
List<String> herbList = new ArrayList<String>();
herbList.add("Parsley");
herbList.add("Sage");
herbList.add("Rosemary");
herbList.add("Rosemary");
herbList.add("Rosemary");
```

Inserting elements into a list

An element can be inserted at any index we please by using an `add()` message, which takes two arguments (the first being the index into the list, the second being the object to add at that index). This is demonstrated in the next code example, which places the filling between the slices of bread. When the filling is placed at index `1`, the top slice is moved up to position `2`.

```
List<String> sandwich = new ArrayList<String>();
sandwich.add("Bottom slice");
sandwich.add("Top slice");
sandwich.add(1, "Filling");
```

ACTIVITY 1

Open Unit11_Project_1. Open the editor on the class `ListExamples`, where you will find the methods `sizeDemo()`, `addDemo()`, `duplicatesDemo()` and `insertDemo()`, which demonstrate what you have learnt about lists so far.

Study the comments and code of these methods and then with pencil and paper jot down what output you would expect if each of the following statements were executed. As in *Unit 10* the methods are static and are therefore invoked on the class.

```
ListExamples.sizeDemo();
ListExamples.addDemo();
ListExamples.duplicatesDemo();
ListExamples.insertDemo();
```

Now open the OUWorkspace. Enter the statements into the Code Pane and execute them one by one. Compare the results shown in the Display Pane with your predictions.

DISCUSSION OF ACTIVITY 1

The following results should be displayed in the Display Pane:

From `sizeDemo()`:

```
Size of newly created list is: 0
```

From `addDemo()`:

```
Size after adding elements: 3
Parsley
Sage
Rosemary
```

From `duplicatesDemo()`:

```
Size after attempting to add duplicates: 5
Parsley
Sage
Rosemary
Rosemary
Rosemary
```

From `insertDemo()`:

```
Bottom slice
Filling
Top slice
```

Notice that when we iterate through a list we start from index `0`, which we would consider to be at the bottom of the list, and so when printed out the sandwich seems to be upside down!

ACTIVITY 2

When trying out the behaviour of a class it is often useful to investigate what happens at the boundaries. For example, what is the range of index values for which we are allowed to insert elements into a list?

Open Unit11_Project_1 and open the editor on the `ListExamples` class. Scroll down until you find the class method called `rangeDemo()`, which takes a single argument of type `int`.

If you examine this method, you will see that it creates an instance of `ArrayList` and populates it with four elements, which will have indexes `0`, `1`, `2` and `3` respectively. It then tries to insert a further element at the position specified by the argument. If the insertion is successful, a confirmation message is printed, followed by the elements in order.

In the OUWorkspace, invoke the method five times on `ListExamples`, with successive arguments `0`, `2`, `4`, `5` and `-1`.

Observe the results in the Display Pane.

DISCUSSION OF ACTIVITY 2

Adding the new element at position `0`, `2` or `4` succeeds. If the element goes in at `0` or `2` existing elements are moved up one to accommodate it. If the element goes in at `4`, the next unused index, it is added to the end of the list.

However, an attempt to add an element at position `5`, which is two steps beyond the end of the current list, will fail. An exception occurs:

```
java.lang.IndexOutOfBoundsException: Index: 5, Size: 4
```

Attempting to add an element at `-1` also generates an exception; a negative index is illegal.

ACTIVITY 3

Open Unit11_Project_1.

In the class `ListExamples` you will find an outline version of a method called `myDemo1()`. (If you have trouble locating it use the Find button on the toolbar to do a search.) The intention is that the method will store six string elements in an instance of `ArrayList` and then print them out to produce the well-known phrase 'To be or not to be'. The code that inserts the elements is missing at present and you are invited to supply it. However, there is a catch! To make the task more challenging you are required to add the strings to the list in the following sequence:

`"be"` first

`"To"` second

`"or"` third

`"be"` fourth

`"not"` fifth

`"to"` sixth

Therefore you cannot perform the task simply by adding elements to the end of the list. To get the strings into the necessary order, some the elements will have to be inserted at specific indexes.

When you have written your method, recompile the class and invoke the method on `ListExamples` from the OUWorkspace to check that it works as specified.

DISCUSSION OF ACTIVITY 3

Here is our solution; other solutions are possible.

```
public static void myDemo1()
{
    List<String> phrase = new ArrayList<String>();
    phrase.add("be");
    phrase.add(0, "To");
    phrase.add("or");
    phrase.add("be");
    phrase.add(3, "not");
    phrase.add(4, "to");
    for (String eachElement : phrase)
    {
        // Prints elements all on one line, separated by spaces
        System.out.print(eachElement + " ");
    }
}
```

The completed code for the `myDemo1()` *method has been added to the* `ListExamples` *class in Unit11_Project_2.*

Accessing the element at a given position

Accessing the element at a specified position in a list is straightforward. You simply use the message `get()`, as demonstrated in the following code:

```
List<String> herbList = new ArrayList<String>();
herbList.add("Sorrel");
herbList.add("Tansy");
herbList.add("Parsley");
herbList.add("Rosemary");
String selected = herbList.get(2);
```

In the above code, `herbList.get(2)` would return `"Parsley"`.

Removing the element at a given position

The element at a specific index can be removed using the message `remove()`.

Removing a specified element from a list is a little different from removing a specified element from a set, because a list can have duplicate elements. So it is possible to remove an element from a list, reducing the size of the list by one, while other instances of the element remain.

Removing an element does not leave an empty space; instead the elements beyond the one removed shuffle down one position to close the gap.

These points are demonstrated in the following code. Note that the string `"Rosemary"` is duplicated in the list, so we can demonstrate the removal of one occurrence while leaving the other in place.

```
List<String> herbList = new ArrayList<String>();
herbList.add("Sorrel");
herbList.add("Tansy");
herbList.add("Parsley");
herbList.add("Rosemary");
herbList.add("Rosemary");
String removed = herbList.remove(3);
```

In the above code, the message-send `herbList.remove(3)` would remove `"Rosemary"` from the list at index `3`, and the list would then contain the following elements, in the following order:

```
"Sorrel", "Tansy", "Parsley", "Rosemary".
```

Finding the index of a specified element, if it is present

If an element is present in a list we can use a message to find its index. Of course the element may be duplicated in the list!

The message `indexOf()` returns the integer position of the *first* occurrence of its argument in the list. The message `lastIndexOf()` returns the integer position of the *last* occurrence of its argument in the list. Either message returns `-1` if its argument is not an element in the list. These messages are explored in the following code.

```
List<String> herbList = new ArrayList<String>();
herbList.add("Oregano");
herbList.add("Oregano");
herbList.add("Basil");
herbList.add("Rosemary");
herbList.add("Rosemary");
herbList.add("Rosemary");
int position1 = herbList.indexOf("Oregano");
int position2 = herbList.indexOf("Sorrel");
int position3 = herbList.lastIndexOf("Rosemary");
int position4 = herbList.lastIndexOf("Parsley");
```

In the above code, `position1` is assigned the value `0`, `position2` is assigned the value `-1`, `position3` is assigned the value `5` and finally `position4` is assigned the value `-1`.

Replacing the element at a given index

The element at a given position can be *replaced* by a new one. This is not the same as inserting an element, which increases the size of the list. Here the size stays the same and one of the original elements is overwritten by the new one. This is demonstrated by the following code, which shows all the original contents of a list being overwritten:

```
List<String> herbList = new ArrayList<String>();
herbList.add("Parsley");
herbList.add("Sage");
herbList.add("Rosemary");
herbList.add("and");
herbList.add("Thyme");
herbList.set(0, "Are");
herbList.set(1, "you");
herbList.set(2, "going");
herbList.set(3, "to");
herbList.set(4, "Scarborough Fair");
```

In the above code, `herbList` ends up holding the following elements, in the following order:

```
"Are", "you", "going", "to", "Scarborough Fair".
```

Iterating through a list

Remember, from *Unit 9*, that `foreach` refers to a code construct not a keyword.

Of course, we have already been doing this in most of our demonstration methods! A `foreach` loop will iterate through a list, in index order. For example:

```
for (String herb : herbList)
{
   System.out.println(herb);
}
```

Note that this gives access to the elements but not to the index. If there were some reason why we needed access to the index we could use an ordinary `for` loop and iterate over the index itself, for example:

```
for (int herbIndex = 0; herbIndex < herbList.size(); herbIndex++)
{
   System.out.println(herbList.get(herbIndex) + " is at index "
                      + herbIndex);
}
```

This second type of iteration is only possible because lists have an integer index; it cannot be used with sets and maps.

ACTIVITY 4

Open Unit11_Project_2 and then open the editor on the class `ListExamples`, where you will find the class methods `getDemo()`, `removeDemo()`, `indexOfDemo()` and `setDemo()`.

Study the code of these methods and then with pencil and paper jot down what output you would expect if each of the following statements were executed.

```
ListExamples.getDemo(1);
ListExamples.getDemo(3);
ListExamples.getDemo(5);
ListExamples.removeDemo(2);
ListExamples.indexOfDemo("Basil");
ListExamples.indexOfDemo("Rosemary");
ListExamples.indexOfDemo("Wormwood");
ListExamples.setDemo();
```

Now enter the statements into the OUWorkspace and execute them one by one. Compare the results shown in the Display Pane with your predictions.

DISCUSSION OF ACTIVITY 4

The results should be as follows. We have broken them down by method and provided some commentary.

1 `getDemo()`

Executing the following two statements is straightforward:

```
ListExamples.getDemo(1);
ListExamples.getDemo(3);
```

And the Display Pane shows:

```
Element in position 1 is Tansy
Element in position 3 is Rosemary
```

Executing the statement

```
ListExamples.getDemo(5);
```

results in an exception:

```
Exception: java.lang.IndexOutOfBoundsException: Index: 5, Size: 4
```

Because index `5` is beyond the end of the list.

2 `removeDemo()`

Executing the statement

```
ListExamples.removeDemo(2);
```

results in the following being printed to the Display Pane:

```
Parsley has been removed. List is now:
Sorrel
Tansy
Rosemary
Rosemary
```

3 `indexOfDemo()`

Executing the statements

```
ListExamples.indexOfDemo("Basil");
ListExamples.indexOfDemo("Rosemary");
ListExamples.indexOfDemo("Wormwood");
```

results in the following being printed to the Display Pane:

```
Index of Basil is 2
Last index of Basil is 2
Index of Rosemary is 3
Last index of Rosemary is 5
Index of Wormwood is -1
Last index of Wormwood is -1
```

Basil was found first at index `2` and that was the only place in which it was found, so the last index is also `2`.

Rosemary appears more than once, the first place being `3` and the last `5`.

Wormwood does not appear anywhere, so the first index and last index are both `-1`.

4 `setDemo()`

Executing the statement

```
ListExamples.setDemo();
```

results in the following being printed to the Display Pane:

```
Original contents:
Parsley
Sage
Rosemary
and
Thyme
Contents after replacement:
Are
you
going
to
Scarborough Fair
```

The method has overwritten all five original elements in the list.

ACTIVITY 5

This activity is about programming the riddle below. What word is this?

Can you guess what I am at first and what I become?

Remove my second and I am like smoke.

Change my first and I become something you could put on the fire.

Change my middle and I am a delay

Change my end and I am a boy.

With an extra letter I become heavy.

Change my first and I am an amphibian.

Try it before you read the solution!

I was a FROG, then a FOG. Next a LOG and then a LAG. I was a LAD, then a LOAD. In the end I was a TOAD.

Your task is to write a method which stores the separate letters of FROG in a list and changes, removes or inserts letters as required at each step, ending up with a list which stores the letters of TOAD. At each step the current list should be printed out.

Open Unit11_Project_2 and then open the editor on the class `ListExamples`, where you will find an incomplete method `myDemo2()`. Complete the code where indicated. Then recompile the class and invoke the method from the OUWorkspace to check your riddle is working correctly!

DISCUSSION OF ACTIVITY 5

Your completed method should be similar to this.

```
public static void myDemo2()
{
    List<String> word = new ArrayList<String>();
    word.add("F");
    word.add("R");
    word.add("O");
    word.add("G");
    System.out.println(word);
    word.remove(1);
    System.out.println(word);
    word.set(0, "L");
    System.out.println(word);
    word.set(1, "A");
    System.out.println(word);
    word.set(2, "D");
    System.out.println(word);
    word.add(1, "O");
    System.out.println(word);
    word.set(0, "T");
    System.out.println(word);
}
```

The completed code for the `myDemo2()` *method has been added to the* `ListExamples` *class in Unit11_Project_3.*

1.3 A summary of some `List` methods

This is a useful point at which to summarise the abstract method definitions defined by the `List` interface whose implementation by the `ArrayList` class we have explored so far. Table 2 lists these and also includes the useful testing methods `contains()` and `isEmpty()`.

Table 2 Some methods of the `List` **interface**

Method definition	Category	Description
`add(ElementType element)`	adding	Adds the argument to the list. The `ArrayList` implementation always returns `true`.
`add(int index, ElementType element)`	adding	Adds the second argument to the list at the index specified by the first argument.
`remove(int index)`	removing	Removes and returns the element at the index specified by the argument.
`remove(Object obj)`	removing	If the argument is an element in the list its first occurrence is removed and the method returns `true`. If the argument is not an element in the list it remains unchanged and the method returns `false`.
`set(int index, ElementType element)`	replacing	Replaces the element at the index specified by the first argument with the second argument and returns the original element.
`get(int index)`	accessing	Returns the element at the index specified by the argument.
`indexOf(Object obj)`	searching	Returns the index in the list of the first occurrence of the argument, or `-1` if the list does not contain the argument.
`lastIndexOf(Object obj)`	searching	Returns the index in the list of the last occurrence of the argument, or `-1` if the list does not contain this argument.
`size()`	testing	Returns the number of elements in the list.
`contains(Object obj)`	testing	Returns `true` if the argument is an element in the list, `false` otherwise.
`isEmpty()`	testing	Returns `true` if the list is empty, `false` otherwise.

1.4 Lists and arrays compared

This point of similarity may have had something to do with the name of the class `ArrayList`.

You will probably have noticed that lists are somewhat like arrays. Both structures resemble a series of numbered storage locations, and we can access and change the item stored at a given position.

However, lists are far more flexible. They grow and shrink dynamically, and when elements are inserted or removed all the indexes are automatically adjusted, to make space or close up the gaps. Of course, similar effects are possible using arrays – we can shuffle all the items in an array along, and if we run out of space everything can be copied into a new, bigger array – but we would have to do all the work for ourselves.

Lists also offer test methods, such as `contains()` and `indexOf()`. Again, similar effects are perfectly possible with an array, but we have to write our own code to search through the array for the item we are seeking. This is hard work, and moreover there is a possibility of error. Reusing the list methods, which have been exhaustively tested, saves effort and makes our code more reliable and easier to follow.

Another very strong advantage of lists is that because they are part of the **Java Collections Framework** they work harmoniously with the other collection classes. Arrays are not part of the Collections Framework and so are awkward to integrate with other sorts of collection.

In view of this, why would anyone want to work with arrays? One reason is that arrays, although not so flexible as lists, are very simple and can be very efficient. For certain straightforward tasks, an array may be perfectly adequate and there may simply be no need for the more powerful facilities of the Java Collections Framework. An array is then a better choice. There are also some occasions when only an array will do; for example, some methods require an argument which is an array.

It's possible to start with a list and create an equivalent array – containing the same elements in the same order – by sending it the message `toArray()`. It's also possible to start with an array and produce a list, using a static method of the utility class `Arrays`. Here is how it's done:

You met the utility class `Arrays` in *Unit 9*.

```
String[] stringArray = {"The", "cat", "sat", "on", "the", "mat"};
List<String> stringList = Arrays.asList(stringArray);
```

The method `arrayToListDemo()` in the class `ListExamples` demonstrates this conversion.

Exercise 1

Now that you have met a range of collection classes and the interfaces they implement, as well as fixed-size arrays, and have been introduced to their particular properties, try to decide what sort of structure might be best to model the following practical problems. For each problem, would it be best to use a collection class? If so what interface would that class implement? Or would an array be all that is required? In each case try to give the element or component type as well as a description.

(a) A programmer wants to store the names of the days of the week as strings and access them by day number, starting with Sunday as day zero.

(b) A secretary wants to store the telephone numbers of the members of a focus group and retrieve a person's number given their name.

(c) A doctor wants to keep a record of the names of patients visited out of hours.

(d) A geographer wants to record the heights of mountains in metres and find the height of a mountain given its name.

(e) A classical scholar wants to record all the distinct Latin words that occur in the surviving works of Horace.

(f) A restaurant manager wants to store a menu of the currently available dishes. The list is to be numbered for the convenience of customers ordering food for home delivery. The particular offerings on the menu may alter from time to time, as the restaurant tries out new dishes, or discontinues ones which have not proved popular enough.

(g) A travel agent wants to be able to record the names of winter holiday resorts and the names of the hotels at each resort, and look up the hotels at a given resort.

Solution

(a) This is a cast-iron case for an array! The requirements are very simple. The size is known in advance to be `7` and will never change. The contents, once set, will never need to be altered. All we will ever need to do is access the items by day number, handily starting from `0`, so the obvious thing to do is make the day number the array index. The component type would be `String`.

(b) We must be able to find the number given the name, so a collection class that implements the `Map` interface is needed here, with the name as the key and the number as the value. The element type would be `<String, String>`, i.e. both the keys and the values would be strings. (Telephone numbers cannot conveniently be stored as integers, because a leading zero is significant.)

(c) This looks like a list of names, since a particular patient may have been visited several times and we want to record all the visits, so some names will be repeated. A collection class that implements the `List` interface is needed here. The element type would be `String`.

(d) Again it looks as though we need a collection class that implements the `Map` interface, with the name of the mountain as the key and the height as the value. A suitable element type would be `<String, Integer>`, i.e. the keys would be strings and the values would be integers.

(e) The scholar does not want to know how many times each word appears or anything like that; she simply wants to list the vocabulary used, and each word should appear only once. So a collection class that implements the `Set` interface would be suitable, because it does not allow duplicates. In fact, since the word list will need to be sorted, we could use the `TreeSet` class that we met briefly in *Unit 10*. The element type would be `String`.

(f) The menu items are numbered. Dishes may be added, or taken off, so the size of the menu can vary. New dishes will probably get inserted amongst existing ones, and, when dishes are removed, that section of the menu will contract. From the description it sounds as though a collection class that implements the `List` interface would be best, with the type of the elements being `String`.

(g) This is more complicated! To record the names of the hotels at any particular resort a set would be suitable. The number of hotels at that resort can vary from year to year, they are probably not recorded in any special order, and no hotel ought to appear more than once – if it did, that would be a mistake. All these considerations suggest using a class that implements the `Set` interface.

We also note that the travel agent needs to be able to find the names of the hotels given the name of a resort. This suggests we also need a collection class that implements the `Map` interface, with the resort names as keys, and the values being the sets of hotel names. So the element type would be `<String, Set<String>>`.

2 Can a list be sorted?

As you have seen, lists are **ordered collections**. The elements are stored in the order given by their indexes, and if we iterate through a list this is the order that is followed.

However, the order does not have to reflect anything to do with the elements themselves. It is simply determined by where each element has been placed in the list. There is no general reason to expect that the elements in a list will automatically be in alphabetical or numerical order. For example, the elements in a list of herbs might have been added just in the order they occurred to us!

parsley, basil, thyme, camomile, fennel, ...

This leads to the question: can lists be sorted? For example, can we put our list of herbs into alphabetical order?

basil, camomile, fennel, parsley, thyme, ...

Yes we can!

In *Unit 9* we saw that an *array* can be sorted using a static method of the utility class `Arrays`. This will not work for lists, but there is another utility class that will do what we want. `Collections` (not to be confused with `Collection` – note the final '`s`') is an important part of the Collections Framework and has many static methods for working with collections.

2.1 `Collections` – a utility class

Many of the class (static) methods provided by the `Collections` class are designed to work with lists, and perform sorting and other related services. Here are a few of the most useful and interesting ones. We will first describe them, and then give you a chance to try them out in an activity.

Reversing

To reverse a list, pass it as an argument to the `reverse()` method. For example,

```
Collections.reverse(herbList);
```

will reverse the order of the elements in `herbList`.

Note that this and several of the methods below are **destructive** – they modify the original collection. If you do not want this to happen you will need to create a copy of the collection and work with that instead.

One way of creating a copy of a collection was discussed in *Unit 10*: construct a new collection with the same elements. We will look at this technique again shortly.

Sorting

If the elements in a list have a **natural ordering** (e.g. alphabetical order for strings or numerical order for numbers) they can be sorted according to this ordering by passing the list as an argument to the `sort()` method. For example,

```
Collections.sort(herbList);
```

will sort the contents of `herbList` alphabetically. If we sort a list of numbers, they will be put into ascending numerical order.

This method is destructive.

Swapping

Two elements can be swapped by passing the list and the positions of the items concerned to the `swap()` method. For example,

```
Collections.swap(herbList, 2, 4);
```

will exchange the element of `herbList` at index `2` with the element at index `4`.

If the method is called with either index out of range an exception will be thrown.

This method is destructive.

Shuffling

The elements in a list can be put into random order by passing the list as an argument to the `shuffle()` method. For example,

```
Collections.shuffle(cardList);
```

would put a deck of cards into a random sequence.

This method is destructive.

Locating sublists

Consider two lists, one longer than the other named `longerList` and `shorterList`. The list `shorterlist` *may* represent a **sublist** of `longerlist`. To test if this is indeed the case the `Collections` method `indexOfSubList()` can be used.

The message `indexOfSubList()` searches for occurrences of a given sublist and returns the first starting position at which it is found, or `-1` if it is not present.

For example if `longerList` references

```
["chives", "basil", "thyme", "chervil", "sage", "bay", "thyme", "chervil", "coriander"]
```

and `shorterList` references

```
["thyme", "chervil"]
```

then

```
Collections.indexOfSubList(longerList, shorterList);
```

will return the index `2`; the index at which the sublist starts in the longer list.

This method is non-destructive.

Finding the maximum and minimum

The maximum or minimum element of a list can be found using the `max()` and `min()` methods respectively. These methods return the actual maximum and minimum values, not their positions. Also as with sorting, the elements must have a defined ordering, such as alphabetical or numerical.

For example,

```
String minScore = Collections.min(scoreList);
String maxScore = Collections.max(scoreList);
```

will make `minScore` and `maxScore` reference the smallest and largest values in `scoreList`.

These methods are non-destructive.

ACTIVITY 6

This activity uses a class called `CollectionsExamples`, which is designed to demonstrate the class methods of the utility `Collections` class described above. Open Unit11_Project_3 and then open the editor on the class `CollectionsExamples`, where you will find the following class methods:

- `reverseDemo()`
- `sortDemo()`
- `swapDemo()`
- `shuffleDemo()`
- `indexOfSubListDemo()`
- `maxMinDemo()`

Read through the methods to get an idea of what they do. Then, in the OUWorkspace, invoke each of the methods on the `CollectionsExamples` class and observe the results shown in the Display Pane. When you type the name of the class `CollectionsExamples`, take care not to miss out the 's' – it's easy to type `CollectionExample` by mistake!

DISCUSSION OF ACTIVITY 6

If all has gone according to plan you should have seen all the utility methods in action and understood how you might use them in your own code!

ACTIVITY 7

Open Unit11_Project_3 and then open the editor on the class `CollectionsExamples`, where you will find a skeleton method called `myDemo3()`. Complete this method according to the following specification.

The overall purpose of the method is to allow a user to enter a series of numbers via request dialogue boxes, until Cancel is pressed, and then to present some statistics calculated from the batch of user data. For simplicity we will assume that all the numbers entered are integers.

1. The method should first declare a variable `userData` of type `List` and assign to it a new instance of `ArrayList`.
2. Next the user should be repeatedly prompted, with request dialogue boxes, to enter a series of integers one at a time, until the user signals the end of the series by pressing Cancel. Note that anything typed into a dialogue box will be interpreted as a string. So each entry should be parsed to an integer and added to the list `userData`.
3. The list `userData` should then be sorted.
4. Next an `int` variable `total` should be initialised to `0`. The method should then iterate through the list adding each element to `total`.

   Once the numbers in the list have been totalled, the result should be cast to a `float` and divided by the size of the list to obtain an average, which should be assigned to a `float` variable `avg`. We need to do the calculation this way because the average will not usually be a whole number.

If we want to cast `x` to a `float`, divide it by `y` and assign the result to `z`, it can be done like this:

```
float z = ((float) x)/y;
```

5. The next statistic to be calculated is the median, defined as follows.

   If the number of elements is odd, the median is the middle one in the sorted list.

   If the number of elements is even, the median is the average of the middle two elements.

   For example, the median of `1, 2, 2, 3, 5` is `2`, the middle value. The median of `1, 2, 2, 3, 4, 5` is `2.5`, the average of the two middle values, `2` and `3`.

Once you have calculated the median you should assign it to a `float` variable `med`. The code for this calculation is somewhat tricky, and if you get stuck you can find out how it's done by looking in the project's README.TXT file.

6 Now the maximum and minimum should be assigned to `int` variables `maxVal` and `minVal`. Since the list was sorted in step 3 above, the maximum and minimum will be the last and first elements of the list.

7 Finally the statistics should be output in a form such as:

```
There were 10 numbers.
The average was 4.2
The median was 3.5
The maximum was 7 and the minimum was 1
```

When you have written your method, compile it and test it from the OUWorkspace. If you input the numbers `3, 4, 6, 6, 2, 1, 7, 3, 3` and `7` you should obtain the results above. You should also try it with an odd number of data, to make sure the median is calculated correctly in that case.

DISCUSSION OF ACTIVITY 7

Our version of the method is as follows. We have indicated the steps given in the specification.

```
public static void myDemo3()
{
    // Step 1 – create empty list
    List<Integer> userData = new ArrayList<Integer>();

    // Step 2 – data entry
    String dialogString = "Please input an integer. Press Cancel to end.";
    String input = OUDialog.request(dialogString);
    while (input != null)
    {
        userData.add(Integer.parseInt(input));
        input = OUDialog.request(dialogString);
    }

    // Step 3 – sort data
    Collections.sort(userData);

    // Step 4 – total data and calculate average
    int total = 0;
    for (int eachElement : userData)
    {
        total = total + eachElement;
    }
    float avg = ((float)total) / userData.size();

    // Step 5 – find median
    float med;
    int size = userData.size();
    if (size % 2 == 1 ) // Odd number of data
```

```
    {
        // Middle item
        int mid = (size - 1) / 2;
        med = userData.get(mid);
    }
    else // Even number of data
    {
        // Average of two middle items
        int mid1 = (size / 2) - 1;
        int mid2 = size / 2;
        med = ((float) (userData.get(mid1) + userData.get(mid2))) / 2;
    }

    // Step 6 – find maximum and minimum
    int maxVal = userData.get(size - 1);
    int minVal = userData.get(0);

    // Step 7 – output results
    System.out.println("There were " + size + " numbers.");
    System.out.println("The average was " + avg);
    System.out.println("The median was " + med);
    System.out.println("The maximum was " + maxVal + " and the minimum was " + minVal);
}
```

The completed code for the `myDemo3()` *method has been added to the* `CollectionsExamples` *class in Unit11_Project_4.*

2.2 New collections for old

Classes that implement the `Collection` interface – including all classes that implement the `Set` and `List` interfaces – are expected to have two constructors: one that takes no argument, and creates an empty collection; and one that takes any type of collection as an argument, and creates a new collection with the same elements.

It's easy to see why the first is needed, but what is the purpose of the second? We saw one use in *Unit 10*, when we were looking at bulk operations. If an operation modifies a collection, we may want to create a copy and work with that, so that the original is not destroyed. A copy can be obtained by passing the original as an argument to the constructor. In this case both the collections, the old and the new, are of the same kind.

There is another reason why we might use the second constructor: to create a collection of a *different* kind but with the same elements.

Why would we want to do this? Imagine we have a mailing list, stored as an instance of `ArrayList` perhaps, and we discover that it contains duplicates which should not be there. (This is a common problem in real-life mailing lists!) Do we have to go through and weed them out manually somehow?

There is a better way. We use the list to create a set. A set cannot contain duplicates, so they are all eliminated! Each name will now appear only once.

But maybe there was a good reason why we used a list in the first place and a set is not what we want. No problem – we simply use the set to create a list! Now we are back where we started, but with all the duplicates automatically eliminated – and all in a single line of code. This is a good illustration of the extraordinary power and flexibility of the **collection classes**.

2.3 Bulk operations on lists

In *Unit 10* we looked at bulk operations on sets and you will probably not be surprised to learn that there are similar bulk operations on lists as well. Much of the power of the Collections Framework – the way it enhances our productivity as programmers – comes from these bulk operations, which let us manipulate entire collections as individual objects, without even needing to iterate through the elements.

Bulk operations on lists include `addAll()`, `containsAll()`, `removeAll()` and `retainAll()`. Their effect on lists is broadly similar to their effect on sets, except of course that lists allow duplicates whereas sets do not. The difference this makes will be demonstrated in the next activity.

ACTIVITY 8

Open Unit11_Project_4 and open the `BulkExamples` class in the editor. You will find it contains a single static method `myDemo4()`, which is incomplete. If you read the method you will find that so far all it does is set up two lists of `Character` objects, assigned to `list1` and `list2` respectively, and displays them. Your task is to create and display some new lists obtained using bulk operations.

Since the operations are destructive you will need to create a new copy of `list1` for each bulk operation.

1 Add to `myDemo4()` code that will:
   - make a copy of `list1` and assign it to `list3`;
   - send `list3` the message `addAll(list2)`;
   - print out the resulting list.

   Compile the class and execute the method from the OUWorkspace. Observe the results in the Display Pane.

2 Add and execute code to try out `removeAll()` in a similar way.

3 Do the same for `retainAll()`.

4 Finally add and execute the code to create a new instance of `TreeSet<Character>`, passing `list1` as the argument to the constructor, and print out the result.

DISCUSSION OF ACTIVITY 8

The method `myDemo4()` first prints to the Display Pane the textual representations of `list1` and `list2`:

```
This is list 1: [a, b, r, a, c, a, d, a, b, r, a]
This is list 2: [d, a, b, c, h, i, c, k]
```

1 Here is the code we added to demonstrate the effects of `addAll()`:

```
List<Character> list3 = new ArrayList<Character>(list1);
list3.addAll(list2);
System.out.println("This is list 3: " + list3);
```

   This produces the following in the Display Pane:

```
This is list 3: [a, b, r, a, c, a, d, a, b, r, a, d, a, b, c, h, i, c, k]
```

   The effect of `addAll()` was to add the elements of `list2` to the end of `list1`.

2 Here is the code we added to demonstrate the effects of `removeAll()`:

```
List<Character> list4 = new ArrayList<Character>(list1);
list4.removeAll(list2);
System.out.println("This is list 4: " + list4);
```

This produces the following in the Display Pane:

```
This is list 4: [r, r]
```

We can see that the effect of `removeAll()` is to remove from `list4` any element that also appears in `list2`.

3 Here is the code we added to demonstrate the effects of `retainAll()`:

```
List<Character> list5 = new ArrayList<Character>(list1);
list5.retainAll(list2);
System.out.println("This is list 5: " + list5);
```

This produces the following in the Display Pane:

```
This is list 5: [a, b, a, c, a, d, a, b, a]
```

The effect of `retainAll()` is to retain in `list5` only those elements that also appear in `list2`.

4 Finally, here is the code we added to create a `TreeSet` version of `list1`:

```
Set<Character> set1 = new TreeSet<Character>(list1);
System.out.println("This is set 1: " + set1);
```

This produces the following in the Display Pane:

```
This is set 1: [a, b, c, d, r]
```

By creating a `TreeSet` with `list1` as the argument to the `TreeSet` constructor we eliminate all duplicates and the elements are sorted alphabetically.

The completed code for the `myDemo4()` *method has been added to the* `BulkExamples` *class in Unit11_Project_5.*

3 How to keep frogs in order

We have seen that we can sort a list, provided the elements have a well-defined ordering, so we can speak of one element coming before another in the ordering. Numbers have an obvious natural order of size, and strings and characters can be arranged alphabetically.

But what if we tried to sort frogs? Here is a method which tries to do just that; it creates a list of `Frog`, then creates three frogs, moves each one to a separate stone, and adds them to the list. Then it calls on `Collections` to sort the list.

```
public static void sortFrogs()
{
    List<Frog> frogList = new ArrayList<Frog>();
    Frog algy = new Frog();
    algy.setPosition(3);
    Frog prudence = new Frog();
    prudence.setPosition(1);
    Frog lena = new Frog();
    lena.setPosition(2);
    frogList.add(algy);
    frogList.add(prudence);
    frogList.add(lena);
    Collections.sort(frogList); // Will not compile!
}
```

Unfortunately this will not compile. We get a semantic error:

```
cannot find symbol – method sort(java.util.List<Frog>)
```

This is not terribly helpful, but what the error message is trying to tell us is that `Collections` has no method `sort()` that can take a list of `Frog` as its argument. The reason for this is quite simple really: the argument passed to the method must be a list of things that *can* be sorted, and so far no ordering has been defined for frogs. So how can they be sorted into 'order'? Our request did not really make sense.

3.1 The `Comparable` interface

Fortunately, it is easy to specify an ordering for a class. All we need to do is to make the class implement the interface `Comparable`. Since this interface contains only a single method `compareTo()` this is not too hard.

The idea is that if Java wants to check the ordering of two objects `a` and `b` it uses the result of evaluating the message expression

```
a.compareTo(b);
```

If the result is:

- a negative number, Java will consider that `a` comes before `b` in the ordering;
- a positive number, Java will consider that `a` comes after `b` in the ordering;
- zero, Java will consider that `a` comes neither before nor after `b` – they occupy an equal position in the ordering.

We can try this out in the OUWorkspace;

```
"chalk".compareTo("cheese");
```

returns `-4`, showing `"chalk"` is before `"cheese"`, alphabetically speaking.

(Do not worry about why we get `-4` rather than any other negative number; `-4` just happens to be the result of a particular calculation involving the characters making up the two strings.)

The expression

```
"cheese".compareTo("chalk");
```

returns `4`, showing `"cheese"` is after `"chalk"`, and the expression

```
"cheese".compareTo("cheese");
```

returns `0`, since `"cheese"` has an equal position to `"cheese"`.

ComparableFrog

To be able to sort frogs we introduce a new amphibian species, the `ComparableFrog`. As we have seen, this class must implement the `Comparable` interface, and so it must have a `compareTo()` method, which will return a negative, zero or positive value as described above. But what does it mean to say that one frog comes *before* another?

The answer is that it means whatever we want it to mean! It's entirely up to us and we can make it mean whatever we find useful. As it happens, there is a fairly obvious choice in the case of frogs – we can order them according to their position, and say one frog comes before another if it comes first as we look from left to right.

To make the `compareTo()` method return a negative, zero or positive number as required is easy – we just subtract the position of one frog from the position of the other.

```
public int compareTo(ComparableFrog anotherFrog)
{
   return (this.getPosition() - anotherFrog.getPosition());
}
```

For example, if the position of the receiver is `3` and the position of the argument is `5`, the comparison yields `3 - 5`, which evaluates to `-2`, which is negative, indicating that the receiver should come before the argument.

ACTIVITY 9

This is a short activity designed to show that a list of `ComparableFrog` objects really does get sorted from left to right as planned!

Open Unit11_Project_5 and open the editor on the `Froggery` class, which includes a class method `sortFrogs()` which puts three instances of `ComparableFrog` into a list, then uses `Collections.sort()` to sort the list.

The dictionary defines a *froggery* as a collection of frogs. Honest!

Read through the method to see what it does, then in the OUWorkspace invoke `sortFrogs()` on the `Froggery` class.

DISCUSSION OF ACTIVITY 9

You should find that instances of `ComparableFrog` are indeed sorted according to position.

In the next series of activities you are going to write a new class called `ClubMember`, which will implement the `Comparable` interface. Instances of `ClubMember` are very simple: they just have a `firstName`, a `secondName` and an `age`. The idea of this new class is to give you some experience of defining and using a class with a natural ordering.

ACTIVITY 10

We begin by creating a basic class. At this stage it does not have to implement `Comparable`; we will attend to that later.

1 Open Unit11_Project_5 and create a new class by clicking on BlueJ's New Class button. When prompted by the dialogue box, give this new class the name `ClubMember`. Declare two instances of type `String` called `firstName` and `lastName` respectively. Then declare an instance variable of type `int` called `age`. Then write the accessor methods for these instance variables.

2 Write a constructor for `ClubMember` that takes three arguments – two strings and an integer – and uses them to set `firstName`, `lastName` and `age` respectively.

DISCUSSION OF ACTIVITY 10

This is our version of the class so far:

```
/**
 * Instances of ClubMember simulate members of a club
 * and have the attributes firstName, lastName and age
 */
public class ClubMember
{
    private int age;
    private String firstName;
    private String lastName;

    public ClubMember(String first, String last, int years)
    {
        this.firstName = first;
        this.lastName = last;
        this.age = years;
    }

    public void setAge(int memberAge)
    {
        this.age = memberAge;
    }

    public int getAge()
    {
        return this.age;
    }
```

```
    public void setFirstName(String memberFirstName)
    {
        this.firstName = memberFirstName;
    }

    public String getFirstName()
    {
        return this.firstName;
    }

    public void setLastName(String memberLastName)
    {
        this.lastName = LastName;
    }

    public String getLastName()
    {
        return this.lastName;
    }
}
```

The code for the `ClubMember` class, as defined so far, has been added to Unit11_Project_6.

ACTIVITY 11

Two possible ways in which instances of `ClubMember` might reasonably want to order themselves for different purposes are by `age` and by `lastName`.

Recall that a `compareTo()` method should return a value which is negative, positive or zero, to give its verdict on the relative ordering of receiver and argument. If the receiver comes before the argument, the value should be negative. If the receiver comes after the argument, the value should be positive. And if they occupy the same position the return value should be zero.

Open Unit11_Project_6 and then open the editor on the class `ClubMember`.

1 Alter `ClubMember` so that the class is declared as implementing `Comparable<ClubMember>`. Make sure you include the type `<ClubMember>` that the comparison will involve.

2 Write a `compareTo()` instance method for `ClubMember` which defines an ordering based on `age`.

3 Compile your class and test your method in the OUWorkspace by executing the following statements one by one.

```
ClubMember x = new ClubMember("Joe", "Green", 27);
ClubMember y = new ClubMember("Maria", "Brown", 21);
x.compareTo(y);
y.compareTo(x);
x.compareTo(x);
```

4 Unit11_Project_6 contains a class called `Club`, which has a static method called `showMembers()` which creates and populates a list of `ClubMember`, then sorts it and prints out a description of its contents. Before you can use this class you will need to compile it – it cannot be compiled until the `ClubMember` class is written and compiled.

Once you have compiled the `Club` class, in the OUWorkspace, invoke the `showMembers()` method on the class. If all has gone according to plan – and if the `compareTo()` method functions correctly – you should see in the Display Pane information about some instances of `ClubMember` displayed in order of `age`.

DISCUSSION OF ACTIVITY 11

1 The first line of code in the class should now be:

```
public class ClubMember implements Comparable<ClubMember>
```

2 The method that compares instances of `ClubMember` by `age` is as follows:

```
public int compareTo(ClubMember anotherMember)
{
   return (this.getAge() - anotherMember.getAge());
}
```

3 The results in the workspace should be

```
6
-6
0
```

4 The output should be:

```
Club member: Gaius Albus, age 21
Club member: Aemilia Rubra, age 27
Club member: Decimus Flammeus, age 30
Club member: Claudia Fusca, age 35
Club member: Brutus Albus, age 48
Club member: Felicia Prasina, age 61
```

The updated code for the `ClubMember` *class, as defined so far, has been added to Unit11_Project_7.*

ACTIVITY 12

In this activity you will modify the `compareTo()` method in the `ClubMember` class so it now compares instances of `ClubMember` according to `lastName`.

The method will need to return a negative, zero or positive result, depending on which object's `lastName` comes first alphabetically. Luckily there is a method in the `String` class which will produce exactly the result we need to return. This method is called ... `compareTo()`! The expression

```
aString.compareTo(bString)
```

will evaluate to a negative, zero or positive result depending on whether `aString` is alphabetically before, the same as, or after `bString`. So all we need do is compare the last names using this method and return the result.

Open Unit11_Project_7 and then open the editor on the class `ClubMember`.

1 Modify the `compareTo()` method in `ClubMember` so it performs a comparison based on the `lastName` instance variable.

2 Once you have recompiled the `ClubMember` class, in the OUWorkspace invoke the class method `showMembers()` on the `Club` class. This time you should see the club members displayed in order of `lastName`.

DISCUSSION OF ACTIVITY 12

1 Our method was as follows.

```
public int compareTo(ClubMember anotherMember)
{
   return this.getLastName().compareTo(anotherMember.getLastName());
}
```

2 The output from invoking the class method `showMembers()` on the `Club` class should now be:

```
Club member: Gaius Albus, age 21
Club member: Brutus Albus, age 48
Club member: Decimus Flammeus, age 30
Club member: Claudia Fusca, age 35
Club member: Felicia Prasina, age 61
Club member: Aemilia Rubra, age 27
```

Notice that two members share the last name Albus, so they occupy the same position in the ordering. In these circumstances the sort does not reorder the two but leaves them in the original order, with Gaius before Brutus. A sort that behaves like this – not reordering equal elements – is known as a **stable sort**.

If we were sorting the names manually we *would* have reordered them, because when last names tie the usual practice is to arrange them according to the alphabetical order of the *first names*. So we would have placed Brutus before Gaius. This is called a **secondary sort**, the sort according to last name in this case being the **primary sort**.

How can we arrange for secondary sorting? The solution is to make our `compareTo()` method return the result of comparing the last names *unless* they are equal, in which case the result of comparing the *first names* is returned instead.

To demonstrate this idea in action we have written the classes `ClubMember2` and `Club2`, which exactly parallel the original `ClubMember` and `Club`, except that the `compareTo()` method in `Club2` uses the idea above to do a primary and then a secondary sort. If you invoke `Club2.showMembers()` you should find that Brutus is now in his rightful position!

The updated code for `compareTo()` *in the* `ClubMember` *class has been added to Unit11_Project_8.*

4 Sorted collections

As we have seen, lists are ordered. If the element type of a list has a natural ordering – in other words, if it implements `Comparable` and so has a `compareTo()` method – we can call on the services of `Collections` to sort the elements. After sorting, the elements will be arranged in the list according to their natural ordering. Numbers will follow size order, strings will be in alphabetical order, instances of `comparableFrog` will be in order of position, and so on.

But what if we add a new element? If we just tag it on at the end, the chances are it will not be in the right place as far as the ordering goes, so we will have to sort the list all over again. Alternatively we might write code to work out where the new element should fit in and insert it into the list at the right point. This can certainly be done but it's fiddly. Wouldn't it be nice if we had a kind of collection which automatically kept all its elements sorted, and whenever we added a new element to it put the element in the right position according to its natural ordering *without us having to do anything*? Ready-sorted collections, you might say.

As you will have guessed, such a kind of collection does exist, and it's called a sorted collection. You have already met one kind of sorted collection briefly in *Unit 10*: the `TreeSet`. You may recall that when we stored our herbs in a `TreeSet` rather than a `HashSet` they magically became sorted into alphabetical order.

Sorted collections implement one of two interfaces: `SortedSet`, which is a subinterface of `Set`; and `SortedMap`, which is a subinterface of `Map`.

`TreeSet` implements `SortedSet` and is like `HashSet` except that its elements are always arranged in their natural order. Similarly there is a collection class `TreeMap`, which implements `SortedMap` and is like a `HashMap` except that its entries are always arranged according to the natural order of the keys.

Exercise 2

(a) Which of the following types of ordered set do you think are legal?

- `TreeSet<HoverFrog>`
- `TreeSet<ComparableFrog>`
- `TreeSet<Integer>`
- `TreeSet<String>`
- `TreeSet<int>`

(b) Why is there no interface `SortedList`?

Solution

(a) The contents of a `TreeSet` are kept sorted by natural order, so the elements must have a natural order defined on them. This rules out `TreeSet<HoverFrog>`, because `HoverFrog` does not implement `Comparable` and so instances of `HoverFrog` cannot be sorted.

`TreeSet<ComparableFrog>` is fine, since `ComparableFrog` implements `Comparable`.

`Integer` and `String` likewise have a natural order, so `TreeSet<Integer>` and `TreeSet<String>` are also fine.

`TreeSet<int>` is illegal because collections can contain only objects, not primitive data types. Of course we could use a `TreeSet` of `Integer`, in which case autoboxing would make it *seem* as though `int` values could be added to it.

(b) It would be a contradiction in terms!

If it is a *sorted* collection, the implication is that its elements will *at all times* be arranged in their natural order. But a list is an *indexed* collection: implying that an element can be stored at a chosen index. These are contradictory requirements. If we tried to insert at new element at a particular index, `12` say, it would instantly be whisked away to whatever position it should occupy according to the natural ordering.

The price of using sorted collections

However the convenience of using a sorted collection comes at a price. As you know, sets cannot contain duplicate elements, and maps cannot contain duplicate keys. What does duplicate mean precisely? For a sorted collection it means equal according to the natural ordering – in other words if `a.compareTo(b)` returns zero, `a` and `b` are duplicates! Of course they might actually be different objects – two distinct instances of `ComparableFrog` that just happen to be on the same stone for instance – but nevertheless *to a sorted collection* they appear to be one and the same. So if we want to sort a collection but retain duplicate elements, we will just have to use a list and sort it as required using `Collections`.

The next activity demonstrates the behaviour of a sorted set.

ACTIVITY 13

Open Unit11_Project_8 and open the OUWorkspace.

1 In the class `Froggery` there is a method `treeFrogs()`, which creates three instances of `ComparableFrog`. To make sure there is no doubt they are different frogs, we set their colours so the first is coloured red, the second green and the third blue. However the position of the frogs is not changed, so they are all at home position `1`.

They are then added to a `TreeSet` object, `frogSet`, and the contents of the set are printed to the Display Pane.

Execute the method from the OUWorkspace and observe the output in the Display Pane. What do you find?

2 Open `Froggery` and try the effect of editing the method `treeFrogs()` to include the statements

```
greenFrog.right();
blueFrog.left();
```

before the frogs are added to `frogSet`. Recompile and execute `Froggery.treeFrogs()` again.

DISCUSSION OF ACTIVITY 13

1 There is only one `ComparableFrog` in `frogSet`. Although the frogs have different colours, the `compareTo()` method only takes account of their position. Since all three have their `position` instance variable set to `1`, the `TreeSet` object considers them to be the same, and so the green and blue frogs are considered duplicates and do not get added.

2 There are now three ComparableFrogs in frogSet. This is because we changed the green frog's position to 2 (with the right() message) and the position of the blue frog to 0 (with the left() message), so now all three frogs have different positions and so are different in the eyes of the TreeSet object. As they are added to frogSet they are automatically inserted in sorted order, so the blue frog is at position 0, the red frog is at position 1 and the green frog is at position 2.

4.1 Are some objects more equal than others?

So sorted sets treat two objects as the same if the result of compareTo() is zero. But what happens in unsorted sets, such as HashSet? More generally, what does it mean to say that two objects are *equal*?

You know that we can compare two objects using ==. You also know that there is an equals() method. This method is defined in Object and since all classes inherit directly or indirectly from Object, all objects in the Java universe understand the message equals(anotherObject).

SAQ 1

What is the difference between == and equals()?

ANSWER...

For object references a and b, a == b is true only if they both reference exactly the same object, i.e. they both point to the same address in memory.

a.equals(b) may be testing a weaker condition – not whether they are the same object, but rather whether they are the same for our purposes. This usually means that they are indistinguishable in terms of *state*. For example, aString.equals(bString) will be true if both strings contain the same characters in the same order, even if they are actually different objects.

It makes sense for String equals() to work in this way. Imagine if we wanted to compare a password string with a stored password when a user logged in. The two strings would be different objects but we would want them to be considered equal if the user had typed in the right sequence of characters. If the user had to supply the identical *object* they would be unable to log in!

In the class Object, however, == and equals() do precisely the same thing as one another. In fact the definition of Object equals() is as follows:

```
public boolean equals(Object obj)
{
    return (this == obj);
}
```

So how did String equals() come to be *different* from ==? Well, the designers of the String class decided that for strings, 'equals' ought to mean having the same characters in the same order, which was obviously a sensible decision. So they *overrode* the equals() method inherited from Object in favour of one which compared the content of two strings.

The designer of any class can do the same, and equals() can mean whatever we like. We simply override the inherited equals() with a version that does what we want.

You may be wondering what all this has to do with collections. The importance lies in the fact that unsorted collections, such as sets and maps, use `equals()` as the basis for deciding whether two elements duplicate one another. So if we try to add a new element to an instance of `HashSet`, or a new key/value pair to an instance of `HashMap`, it will be `equals()` that is used to detect whether the element or the key is already present. `equals()` is also used in testing methods, such as `contains()`.

Writing an `equals()` method

To demonstrate the writing of an `equals()` method, we introduce a simple class, `RowOfStars`. Objects of this class represent a row of asterisks like `******` and have a single integer attribute, `length`, which denotes the number of stars in the row. Two instances of `RowOfStars` are reckoned as equal if they consist of the same number of stars.

To make this method work with collection classes that decide whether elements are duplicates by using `equals()`, we must make sure that we are overriding the `equals()` method inherited from `Object`. We looked at the method inherited from `Object` earlier, but here it is again:

```
public boolean equals(Object obj)
{
    return (this == obj);
}
```

Note the signature, `equals(Object obj)`. To override this method we must use the same signature. If we used `equals(RowOfStars obj)`it would have a different signature, and would merely *overload* the method, not override it.

So we know the method we are trying to write must have the form

```
public boolean equals(Object obj)
{
    // Method body – our code
}
```

Filling in the method body is not too hard. Two `RowOfStars` instances are deemed equal if their lengths are the same, so we could write this:

```
public boolean equals(Object obj)
{
    return (this.getLength() == obj.getLength());
}
```

But this will not work! Can you see why?

The reason is that the argument `obj` is declared as `Object obj` in the method header. Therefore we can only send it messages from the protocol of object, and this does not include any such message as `getLength()`!

We can get round this with a cast. We declare a `RowOfStars` variable `row` and then cast the argument `obj` to an instance of `RowOfStars`, like this:

```
RowOfStars row = (RowOfStars) obj;
```

This works because the object concerned actually is a `RowOfStars`, so Java can cast it back to that type.

Using the cast, we rewrite the `equals()` method as follows

```
public boolean equals(Object obj)
{
    RowOfStars row = (RowOfStars) obj;
    return (this.getLength() == row.getLength());
}
```

A better `equals()` method would check that the cast is safe before attempting it, and would also test whether the argument is `null`. But we have not bothered about these checks, because we want to keep the method as simple as possible.

This now works fine, since `getLength()` is in the protocol of `RowOfStars`. There is a danger though – what if, when the program runs, `obj` does *not* actually refer to an instance of `RowOfStars`? The answer is that an exception will occur and the method will fail. However, we plan to use the method only in circumstances where we know that the object will be an instance of `RowOfStars`, so the issue will not crop up for us.

ACTIVITY 14

1 Open Unit11_Project_8, and then open the editor on the class `RowOfStars` and verify that the `equals()` method is as given above.

2 In the OUWorkspace execute the following statements:

```
RowOfStars rs1 = new RowOfStars(7);
RowOfStars rs2 = new RowOfStars(7);
RowOfStars rs3 = new RowOfStars(11);
rs1.equals(rs2);
rs1.equals(rs3);
```

You should find that as expected `rs1.equals(rs2)` returns `true` (the states of the two objects are the same) but `rs1.equals(rs3)` returns `false` (the states are different). Our `equals()` method is working as it is supposed to.

3 Next open the class `Galaxy` and read the single static method `showStars()`, which creates a `HashSet` of `RowOfStars`, adds some instances of `RowOfStars` to it, and displays the result.

Note that several of the instances added to the set have the same state, so they are all equal, and would be duplicates in the set. We expect only one of them to be added, since a set cannot contain duplicates.

4 Now execute `Galaxy.showStars()` in the OUWorkspace and observe what actually happens! Do not spend any time puzzling over it but come straight back here to the activity discussion where all will be explained!

DISCUSSION OF ACTIVITY 14

After executing `Galaxy.showStars()` we saw the following printed in the Display Pane:

```
***
***
**
***
*
***
```

You will probably have got a different pattern, since the elements in the set are not guaranteed to be in any particular order. However, the important point is that the set contains *duplicates* – `***` shows up four times – and it should *not*. We went to all that trouble to write an `equals()` method, but the duplicates have not been recognised. What has gone wrong?

Do not worry. This behaviour of sets is a classic trap, which has baffled many a programmer! It arises from the way the elements of a set are stored in memory. Once we understand the storage mechanism, the problem is very easily fixed.

What puts the hash in `HashSet`?

To understand what has happened we need to examine how objects are stored in a set and retrieved again later.

One of the benefits of using a set is that searching for a particular element is extremely fast, even if the set contains many thousands of elements. This very high efficiency is achieved by storing the elements in a series of compartments called buckets (Figure 2). Each bucket can hold more than one element, although we try to arrange for the number to be kept as small as possible (we will see why shortly).

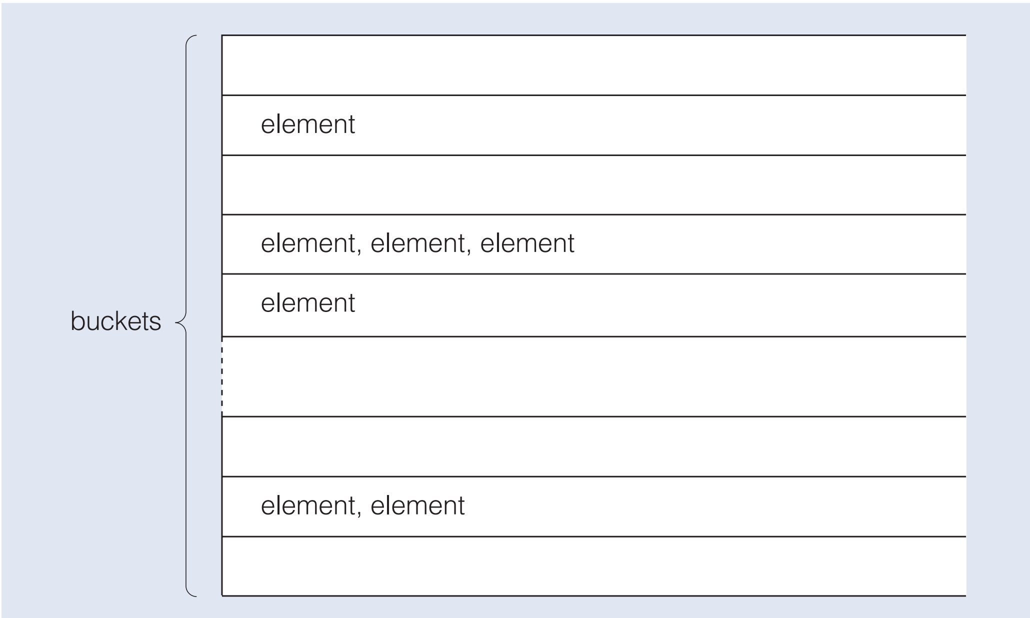


Figure 2 Hash buckets

If Java wants to add an object to a set, it sends the object the message `hashCode()`, which all objects understand, because it is inherited from `Object`. The object returns an `int` value called the **hash code** and from this Java is able to work out which bucket the object should be put in. It checks the contents of the bucket to make sure the new object will not duplicate anything already there, and provided this is the case the object is added to the bucket.

If Java needs to know if an object is present in the set, it gets the hash code for the object, which tells it which bucket to look in, then checks the contents of the bucket to see if any of them match.

Using this scheme means that it's extremely quick to find whether or not a particular object is contained in the set, because we can go immediately to the bucket where the object ought to be, and provided the bucket holds only a few elements, checking them all can be done very rapidly.

However, there is a potential problem. Two objects could be equal in terms of state but have different hash codes. If so, when Java comes to add the second object, it will go to a different bucket, and so it will not recognise that the second object is a duplicate of an object in another bucket. This is precisely what has happened in our example, because our class still uses the `hashCode()` method inherited from `Object`, which returns a hash code totally unrelated to an object's state. Figure 3 illustrates the problem.

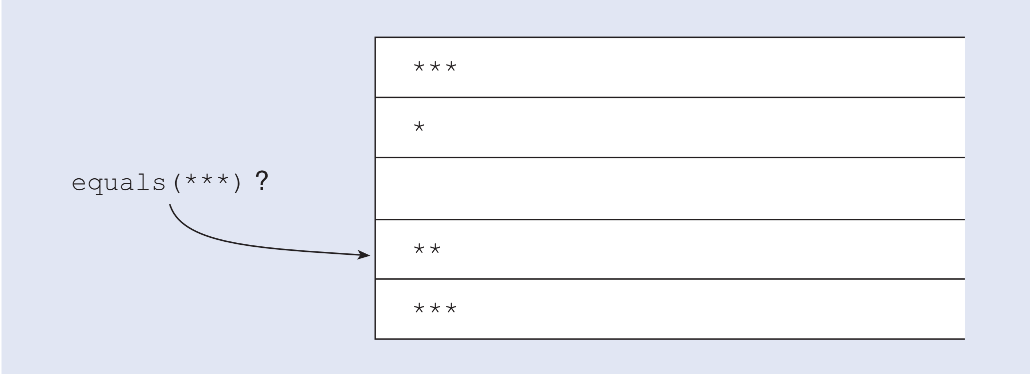


Figure 3 Duplicates in other buckets are not detected

The same difficulty could happen when we search the set for a given object. It might be there, but if Java is looking in a different bucket it will not be found.

Once we understand what is going on, the problem it is fairly easily dealt with. All we need do is make sure that whenever two objects are equal they also have the same hash code, so Java will look in the same bucket for both objects.

In other words, if `a.equals(b)` is true, then `a.hashCode()` should return the same number as `b.hashCode()`. As class designers, all we need to do is override the inherited `hashCode()` method so that this is the case.

There is too little space to explore the details here, but when a class has several attributes, finding a suitable formula for calculating the hash code will generally be a little more complex. We need to make sure the hash codes are well spread out, because we want the objects to be distributed as evenly as possible amongst the buckets.

In our particular example the class `RowOfStars` should override the inherited `hashCode()` so that two rows of equal length will answer with the same hash code. No fancy calculation is required – we simply give the hash code the same value as the length!

```
public int hashCode()
{
   return this.getLength();
}
```

ACTIVITY 15

Open Unit11_Project_8. Open the editor on the class `RowOfStars` and enter the `hashCode()` method given above. Once you have recompiled the class, execute `Galaxy.showStars()` in the OUWorkspace.

DISCUSSION OF ACTIVITY 15

Hurray! No duplicates!

The code for the `hashCode()` method has been added to the `RowOfStars` class in Unit11_Project_9.

Three kinds of equals

Equality seems such a basic idea – surely two things are either the same or not? But now we have uncovered three different forms of equality in Java:

1 Object identity, tested with ==. If `a` and `b` are references and `a == b` is true, this means both `a` and `b` reference exactly the same area in memory. This is always the case: we cannot redefine ==.

2 Equality of state or content. The version of `equals()` inherited from `Object` is no different in its effect from ==. But class designers can override `equals()` to suit their own purposes. Usually, if `a.equals(b)` is true, this means `a` and `b` have the same state, although we have the freedom to make it mean whatever we like.

However – as we have seen – whenever we override `equals()` we had better override `hashCode()` as well! Otherwise we will get the version of `hashCode()` from `Object`, which yields a different hash code for every object, and then our class will not work properly with some collections. If `a.equals(b)` is true, `a` and `b` should return the same hash-code value.

3 Equality in a natural ordering. If `a.compareTo(b)` is zero, `a` and `b` will be treated as equal in sorted collections. If we want, we can define `compareTo()` and `equals()` completely separately, but it is not a very good idea, because then two objects could be treated as the same in one kind of collection yet different in another. So we ought to make sure that if we define a `compareTo()` method for a class it is consistent with our `equals()` method.

Instances of `RowOfStars` cannot be stored in a sorted collection, because the `RowOfStars` class does not implement the `Comparable` interface.

If we wanted to give `RowOfStars` a `compareTo()` method, we should base it on the length:

```
public int compareTo(RowOfStars anotherRow)
{
   return (this.getLength() - anotherRow.getLength());
}
```

This would give a class in which `equals()`, `hashCode()` and `compareTo()` were all consistent with one another. This approach is taken in the `compareTo()` method of the `ComparableRowOfStars` class, which you can find in Unit11_Project_10. Objects of this class will be stored correctly in any type of Java collection.

ACTIVITY 16

Open Unit11_Project_10 and open the editor on the class `ComparableRowOfStars`. Confirm that the class has implemented the `Comparable` interface and that it has defined the `compareTo()` method in terms of the `length` instance variable. Next open the editor on the class `Galaxy2` and note that this class creates a `TreeSet` and adds a number of instances of `ComparableRowOfStars` to the sorted set before printing the textual representation of the elements to the Display Pane.

In the OUWorkspace execute `Galaxy2.showStars()` and observe the results in the Display Pane.

DISCUSSION OF ACTIVITY 16

Double hurrah! There are no duplicates and the elements are sorted!

5 Summary

After studying this unit, you should understand the following ideas:

- A list is a variable-sized collection that is ordered, has an integer index and permits duplicate elements.
- `List` is a subinterface of `Collection`, and so it implements the protocol of `Collection`, as well as some additional behaviour.
- The protocol of `List` includes methods for adding elements, inserting elements, accessing elements by index, removing elements and overwriting elements already present. It also includes methods for testing the list and searching for a given object.
- If elements are inserted in or removed from a list, other items are moved up or down to make room or close the gap as necessary.
- `ArrayList` is a good general-purpose implementation of `List`.
- We may iterate through a list using a `foreach` loop, which will access the elements in index order. We can also iterate by using a `for` loop to access each element in turn by its index.
- Lists are more versatile than arrays, but for some applications where this versatility is not needed arrays may provide a simpler solution.
- In addition to having an index order, the element type of a list may have a natural order, such as alphabetical or numerical order. Rearranging the elements so that the index order follows the natural ordering is called sorting the list.
- The utility class `Collections` provides a wide range of static methods for working with collections, including ones to do tasks such as reversing, sorting and shuffling lists, swapping elements in a list, locating maximum and minimum elements, and searching for substrings. Many of these methods are destructive.
- A collection may be passed as the argument for the constructor of a different type of collection. Creating a set from a list in this way is a useful idiom for eliminating duplicates.
- Classes that implement the `List` interface (for example, `ArrayList`) support a number of bulk operations similar to those for sets.
- There are sorted collections whose elements or entries are automatically kept sorted, so they are always in their natural order. Sorted collections implement the `SortedSet` and `SortedMap` interfaces. Only objects which implement the `Comparable` interface may be stored in such collections. Sorted collections cannot contain two elements which have equal order.
- The class `TreeSet` implements the `SortedSet` interface, and `TreeMap` implements the `SortedMap` interface.
- We can define a natural order of our own for a class by implementing the `Comparable` interface.
- Lists are not sorted collections, but they can be sorted by the `sort()` method of the `Collections` class, so long as the elements in the list implement the `Comparable` interface.
- As well as object identity, tested with ==, there is another from of equality, which is tested using `equals(Object)`. In `Object` the two forms have exactly the same effect. However a class designer can redefine the `equals()` inherited from `Object` so that it tests for equal state, or some other similarity.

- Whenever `equals()` is overridden, `hashCode()` must also be overridden so that equal objects return the same hash code. Otherwise, objects will not be stored properly in collections that implement the `Set` and `Map` interfaces (or subinterfaces).
- If a class implements `Comparable` its `equals()`, `hashCode()` and `compareTo()` methods should be consistent with one another, so that objects that have equal state also occupy the same point in the natural ordering. Objects of such a class can be stored in any kind of Java collection.

LEARNING OUTCOMES

After studying this unit you should be able to:

- explain the properties of a list;
- use messages that are defined by the `List` interface, including ones to add, insert, access, remove, overwrite and search for elements, and test the list;
- use the class `ArrayList`, which implements the `List` interface;
- understand where collection classes should be used and where arrays would be adequate;
- use the class methods of the utility class `Collections` to perform a range of operations on lists;
- copy a list before invoking a destructive operation;
- use bulk operations with lists;
- use the idiom of creating a set with the same elements as a given collection as a way of removing duplicates;
- explain with reasons what sort of collection might be most suitable for a given application;
- understand what is meant by a natural ordering;
- define a natural ordering for a class by making it implement the `Comparable` interface;
- understand that a sorted collection maintains its elements in natural order and explain how it decides whether elements are duplicates;
- override the `equals()` method inherited from `Object` to make it test for equality of state;
- understand that if `equals()` is overridden then `hashCode()` must also be overidden, so that objects for which `equals()` returns `true` produce the same hash codes, and that if this is not done, objects will not be stored correctly in collections that implement the `Set` and `Map` interfaces;
- understand that `compareTo()` should be consistent with `equals()`;
- understand how to write a simple class that overrides `equals()`, `hashCode()` and `compareTo()` so that they are all consistent, in order that instances of the class can be stored in any kind of Java collection.

Glossary

`Comparable` An interface that defines the header for a single method: `compareTo()`. Classes which implement this interface enable their instances to have a natural order and can therefore be added as elements to sorted collections.

`compareTo()` The sole method of the `Comparable` interface. Allows classes of objects implementing this interface to be sorted (i.e. gives them a **natural ordering**).

`equals()` A method inherited by all classes from `Object`. In the `Object` class the method simply compares the receiver with the argument using the == operator and so will return true if they both reference exactly the same object – in other words, they both point to the same address in memory. However, the method is frequently overridden to define versions of equality appropriate to particular classes and particular purposes. For many classes, such as `String`, the method `equals()` is defined effectively to mean 'has the same state as'. More generally, `equals()` is coded to mean 'should be regarded as equal for our purposes'.

hash bucket A hash bucket is a 'compartment' used in the internal implementation of a set. The set uses the **hash code** of each incoming element to decide which hash bucket to store it in.

hash code In Java, a hash code is an integer that can be calculated for any object, by sending it the message `hashCode()`. This method is inherited from `Object`, but in general must be overridden if `equals()` has been overridden. Hash codes are used by sets and related collections to allow them to store elements efficiently and to check rapidly for duplicates. If a class redefines `equals()`, then the `hashCode()` method for that class must take the definition into account; otherwise the type will not behave properly in collections.

index In some collections (e.g. lists, but not sets), any element can be accessed by using a place number. Such collections in effect allow you to say 'Give me the first element', 'Give me the second element' etc. An index is just a place number. Additionally, the word 'index' is used in several closely related ways, either to refer to a particular place number or set of place numbers or, in everyday English, to refer to a complete table of place numbers and the elements they refer to. In computing, the term 'index' can cover not only place numbers, but also non-numeric keys used to much the same effect. Sometime the phrase *integer index* is used to narrow down an index to the place-number kind.

list A list is an **ordered collection** of variable size that permits duplicate elements.

natural ordering In Java, an element type is said to have a natural ordering if it implements the interface `Comparable` and consequently has a `compareTo()` method. For numbers and strings, the natural ordering corresponds simply to everyday numerical order and alphabetical order.

ordered collection An ordered collection is a type of collection where each element in the collection has a well-defined place, as indicated by its index number. Thus, an ordered collection allows us to access and find the first, second or last element etc. However, the order does not have to reflect anything to do with the elements themselves. It is simply determined by where each element has been placed in the list. This contrasts with a **sorted collection**.

primary versus secondary sort Sometimes a collection should be sorted primarily according to a particular key, but in the case of elements that tie according to this criterion, the elements should be further sorted by some secondary key. These two levels of sorting are referred to as the primary and secondary sort respectively. Both sorts can be accomplished at once by an appropriately designed `compareTo()` method.

sorted collection A sorted collection is a collection that always keeps its elements in their **natural ordering**. This contrasts with an **ordered collection**, where the ordering may just be an accident of where elements happen to have been placed in the collection.

stable sort A sort that does not reorder equal elements is known as a **stable sort**.

sublist A **list**, together with any two positions in the list may be seen as defining a new list containing just those elements between the two positions, retaining the same order. The new list is known as a sublist.

Index