M255 Unit 10
UNDERGRADUATE COMPUTING

Object-oriented programming with Java

Collections: Sets and maps

Unit 10

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 6791 9

2.1

CONTENTS

Introduction 5

1 Set – a collection that allows no duplicates 6

- 1.1 Creating a set 7
- 1.2 Operations on sets 10
- 1.3 Primitive data types and collections 13
- 1.4 Iterating over a set 16
- 1.5 `TreeSet` – a set with order 18

2 Map – a collection with keys and values 20

- 2.1 The map, the key and the value 20
- 2.2 Operations on maps 23
- 2.3 Iterating over maps 27
- 2.4 A summary of some `Map` methods 30

3 Overview of the Collections Framework 32

- 3.1 The collection interfaces 32
- 3.2 The collection classes 34

4 A dating agency 37

- 4.1 Bulk operations on sets 38
- 4.2 A map implementation of the dating agency 43

5 Summary 49

Glossary 52

Index 54

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

Collections are indispensable in programming.

Try to imagine developing a banking system, say, in which every customer account was represented by an individually named variable `acc1`, `acc2`, etc. There could be tens of thousands of these! We would have no general way of locating a particular account when we wanted it, printing out a list of accounts would require many thousands of print statements, one per account, and every time a new customer joined the bank we would have to recompile the program.

The previous chapter showed how to overcome this problem: use *collections*. A collection is a structure which enables multiple items to be bundled as a single unit and dealt with collectively. Once formed, a collection can be passed around as a single object and its contents can be processed all together, by stepping through the collection doing the same thing to each item as we go.

Different kinds of collection

The arrays we met in *Unit 9* are one kind of collection, although as we shall see they are rather different in some ways from the collections we shall be studying here. Many kinds of collection exist, each one having a different behaviour and structure. Each kind of collection is suitable for particular sorts of application.

When you come across a new kind of collection, the following questions are usually worth asking.

- Fixed size versus dynamic: Is the size of the collection fixed once it has been created, or can the size be changed as required?
- Ordering: Are the contents in any particular order or not? If there is an order, how is it defined? Can it be changed, and if so how?
- Homogeneity: Must the contents all be the same type of data or can they be mixed?
- Indexing: Are elements in the collection given a label, index, key, or address of some sort, which can be used to retrieve them? (For example, telephone numbers can be found from a directory by looking up the name.)
- Duplicates: Are duplicates allowed in the collection?

Answers to these questions help to understand the nature of a collection and what sorts of application it might be useful for.

For example, the arrays of *Unit 9* are of fixed size, homogeneous, indexed using integer keys, which may also represent a natural order, and permit duplicate elements.

Suppose we wanted to record the average temperature for each month of the year. Then an array would be ideal. The size is known in advance to be 12; the data are all numbers; we want to be able look up the average temperature given the month number; the month numbers have an order; and it is possible for two or more months to have the same average temperature.

On the other hand, if we wanted to record the people present at a tutorial at any given moment – perhaps in case of fire evacuation – an array might not be very suitable. We will not know in advance how many people will attend, and individuals may come late and/or leave early, so the numbers may fluctuate. Those attending are not in any special order, so giving each person a number would be meaningless. Moreover, a person is either present or not – they will never appear on the list twice.

In this unit you will learn about two very different kinds of collection, sets and maps, which let us do things that an array is not suited for. You will also be introduced to the Java Collections Framework.

1 Set – a collection that allows no duplicates

A set is a collection in which every element is unique. A set can grow and shrink as elements are added or removed. The contents of a set are not indexed, so we have no way of reaching a particular item except by stepping through the elements until we come to the one we want. The most general kind of set is unordered, although we shall see that ordered sets also exist.

So you should think of a set as a pool of objects, none of which can appear twice, and we can add things to the pool or remove them as we please.

Sets occur in many situations outside computing. When would we need a collection which can grow and shrink, has no indexing and contains no duplicates? Examples of such sets would be a *glossary* (like the one at the end of this unit), a *word list*, a *mailing list*, or an *invitation list*. Mailing lists with duplicates are a well-known social hazard.

With collections like these there is often no way of knowing in advance exactly what size the final list will be, and it would be most inconvenient if we had to begin by specifying a fixed length which could not be altered later. We want the length to be flexible and the list to grow as items are added in, or shrink again if they are removed.

Not having to index the collection is also a benefit in these applications – it saves having to specify an index every time we add, access or remove an item. If we used an array-like structure we would have to specify exactly what went where, even though this information was not relevant.

Examples of sets crop up in many other practical situations. As an illustration, suppose I am involved in a garden bird survey, which involves recording the range of different bird species that have been spotted in my garden:

robin

collared dove

blackbird

blue tit

...

I want to write each species down only once, so if a bird comes along which is of a kind already recorded – another blackbird, say – there is no need for a duplicate entry.

Java has several **collection classes** that implement the general notion of a set as described above, but with various additional features. Because there are numerous similar collection classes of this kind, we will not focus primarily on any one particular set class, but rather on the common interface that they all implement – the interface `Set`. This defines a set of common behaviour that allows us to perform operations such as adding and removing elements, finding how many elements the set contains, discovering whether it contains a particular object, and so on. All the different sets found in Java implement this interface.

1.1 Creating a set

In A.A. Milne's *Winnie the Pooh*, the gloomy Eeyore is given two birthday presents – a burst balloon from Piglet and an empty honey jar from Pooh. Far from being disappointed by these gifts, Eeyore is delighted when he realises that the pot is a Useful Pot, ideal for putting the burst balloon in and taking it out again. His friends soon catch on to the possibilities. 'It goes in!' exclaims Pooh. 'And it comes out!' adds Piglet.

In the spirit of Eeyore's useful pot, in the next few activities you will create a set, put things in, and take them out again.

Whenever we use any kind of collection from the **Java Collections Framework** there are three things we must do first.

1 Decide the interface type – what protocol do we want to use with our collection?
2 Choose an implementation class – what class will the collection belong to?
3 Declare an element type – what data type will the collection contain?

These are the three key questions that must be answered before we can create our first set.

Deciding the interface type

We already know the interface type is going to be `Set`, which defines a protocol for working with a set. But there is a little more to think about. To work with a collection, as with any kind of object, we will generally need a variable which refers to it. This variable will have to be declared. When it is declared, what type should it be?

When using the Collections Framework, it is best practice to declare the type of the variable in terms of the *interface* (here `Set`) rather than the implementing class. The reason for doing this is that it adds flexibility. If we later decide to use a different implementation (class) of set there is no need to rewrite all the variable declarations.

So, rather than declare a variable using a specific kind of set, we would write:

```
Set mySet;
```

`mySet` can then be used to reference any kind of set that we please.

Choosing an implementation class

As we have just discussed, `Set` is an interface, not an actual class. Whenever you want to use a set, you will also need to choose a class that implements `Set`. Fortunately, this is straightforward. Unless something special is required, a good general-purpose choice is the class `HashSet`.

Now that you have the `HashSet` class to implement the interface `Set`, the next step is to think about what type of thing you intend to put in your set.

Declaring an element type

In *Unit 9*, when you declared an array you had to declare its component type. With the collections in the Collections Framework (we will explain what this means later in the unit) there is no notion of 'components' (contiguous blocks of memory labelled by an index). Instead elements are simply put into, and taken out of a collection using messages – how or where these elements are actually stored in the collection is of no concern to us. Hence instead of declaring a component type, with the collections in the Collections Framework, you simply declare an **element type**. Only items compatible with the declared element type can then be added to the collection.

The way in which collections in this unit are declared is syntactically very different from the way this is done with arrays. For example, if you wanted a set for holding strings, then you would declare the variable like this:

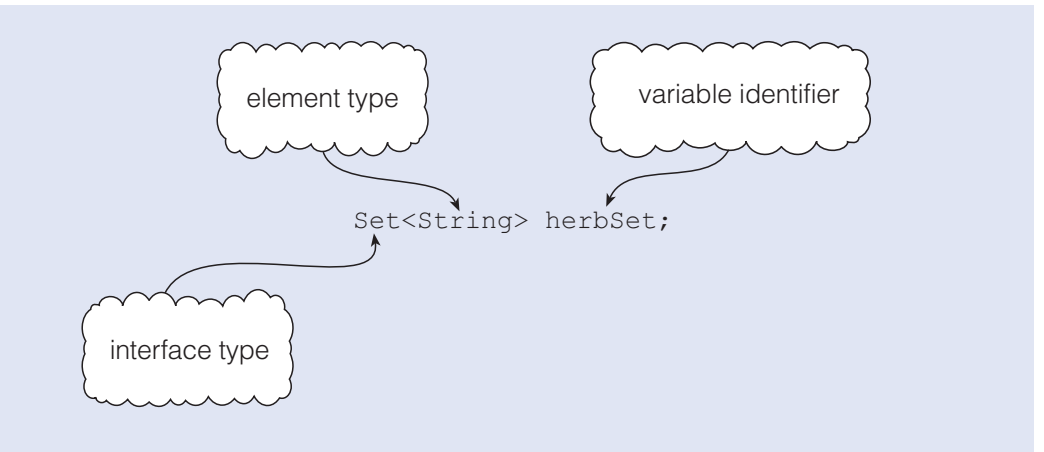


Figure 1 Declaring a variable to reference a set of strings

Then to create an instance of a particular kind of set and assign it to the variable `herbSet`, you would write the following.

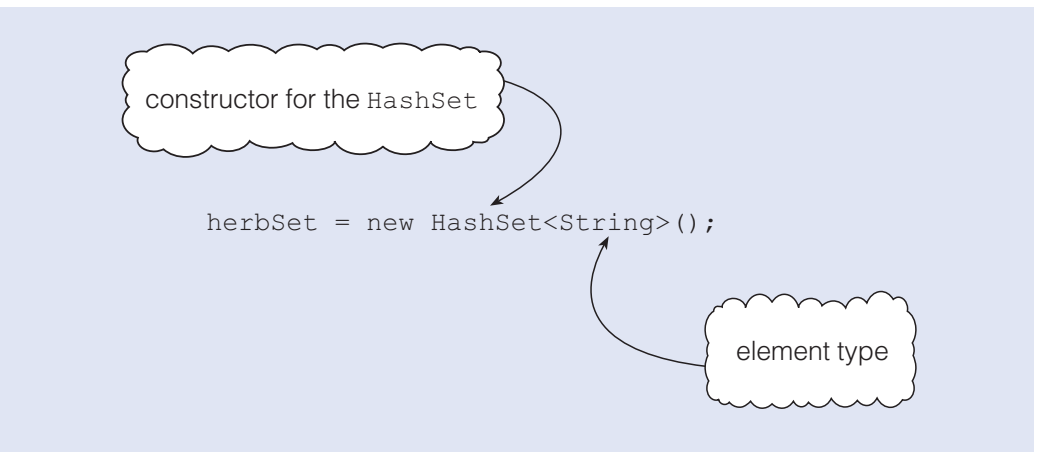


Figure 2 Creating an instance of a set and assigning it to a variable

Notice how the element type is put between '<' and '>'. Note also that `Set<String>` is usually pronounced 'Set of String'.

Of course, you could declare the variable and assign a set to it all in one line:

```
Set<String> herbSet = new HashSet<String>();
```

SAQ 1

Consider the following statement which declares and creates a new set of strings:

```
Set<String> herbSet = new HashSet<String>();
```

Answer the following questions on the basis of the discussions above.

(a) What is the purpose of `<String>` which occurs twice in the above statement?

(b) Given an initial decision to use `HashSet`, why should the instance of `HashSet` be referenced by a variable of type `Set`?

(c) Given an initial decision for `herbSet` to be of type `Set`, why should the `HashSet` class be used?

ANSWER.....

(a) The first occurrence of `<String>` in this statement is used in the declaration of the variable `herbSet`. It specifies that `herbSet` can only be used to reference a set whose element type is type compatible with the element type of `String`. The second occurrence of `<String>` is on the right-hand side of an assignment statement. It is used with the `HashSet` constructor and the operator `new` to create a set with the element type of `String`.

(b) It is good practice, where possible, to declare the variable in terms of an **interface type** (here `Set`), not the type of the implementing class (here `HashSet`). This promotes flexibility and hence maintainability.

(c) If we require a class that implements the `Set` interface and have no other information, `HashSet` is a good general-purpose choice.

Set creation

By this point, you should know enough to be able to make sense of the following statement for creating a set `herbSet`:

```
Set<String> herbSet = new HashSet<String>();
```

This statement cannot compile successfully unless we also tell Java where to find the definitions for both the `Set` interface and the `HashSet` class, so the class in which the statement above appears will have to import these definitions from the `Java.util` package. The simplest approach is to import the whole package with the statement:

```
import java.util.*;
```

In order to explore the behaviour of sets we have written a class called `SetExamples` which contains a number of class methods. Each one is a separate self-contained demonstration of a particular feature or features of sets, which you will be asked to run from the OUWorkspace.

> Important – throughout this unit, when you run a demonstration, the method involved will be static, so remember to invoke it *on the name of the class*.

In this unit we will lay out the code in each of these examples, and explain what is being demonstrated. After some discussion, and sometimes a pencil and paper exercise, there will usually be an activity, in which you will be asked to experiment with the demonstration method concerned by executing it from the OUWorkspace.

In a few cases you are not asked to actually run the code, but the method is still included in the `SetExamples` class for completeness.

Note that these methods are *not* well factored, because they contain a lot of repeated code. They have been structured a bit differently from what we would expect in a real program, in order to keep the demonstrations as clear as possible.

1.2 Operations on sets

Testing the size

One of the simplest things you can do with a set is to test its size. This can be done using the message `size()`. As you might expect, the size of a newly created empty set is `0`. Our first example illustrates this:

```
Set<String> herbSet = new HashSet<String>();
int size = herbSet.size();
System.out.println("The size of the newly created set is: " + size);
```

The above code would print to the Display Pane:

```
The size of the newly created set is: 0.
```

Adding elements

Before we can do anything useful with a set, it needs to have something in it. We can add elements to a set, one at a time, with the message `add()`, as demonstrated by the following code:

```
Set<String> herbSet = new HashSet<String>();
herbSet.add("Parsley");
herbSet.add("Sage");
herbSet.add("Rosemary");
System.out.println("Size after adding elements is: " + herbSet.size());
```

Note we can only add elements that are type compatible with the declared element type of the variable `herbSet`.

No duplicates allowed

A distinctive behaviour of sets is that the same element cannot appear twice. If an attempt is made to add an element already present, the state of the set is left unaltered. The method below demonstrates this. Although `add()` is used five times, there are only three distinct elements involved and consequently the final size is `3` not `5`.

```
Set<String> herbSet = new HashSet<String>();
herbSet.add("Parsley");
herbSet.add("Sage");
herbSet.add("Rosemary");
herbSet.add("Rosemary");
herbSet.add("Rosemary");
System.out.println("Size after attempting to add duplicates is: " + herbSet.size());
```

The above code would print to the Display Pane:

```
Size after attempting to add duplicates is: 3
```

ACTIVITY 1

Open Unit10_Project_1 and look at the class `SetExamples` to satisfy yourself that the methods `sizeDemo()`, `addDemo()` and `noDuplicatesDemo()` are as given above.

Open the OUWorkspace and execute the following statements one by one. Remember the methods are static, so we are invoking them on the class:

```
SetExamples.sizeDemo();
SetExamples.addDemo();
SetExamples.noDuplicatesDemo();
```

What happens if we attempt to add an element already present in a set?

DISCUSSION OF ACTIVITY 1

When first created, the size of the set is `0`, as expected. After adding three distinct elements the size is `3`. Attempts to add one of these items a second and a third time have no effect, since sets cannot contain duplicated elements.

Testing methods

As you have seen, the message `add()` can change the contents of a set. In contrast, the message `size()` always leaves a set unaltered – it simply returns information about it. For this reason, the message `size()` is said to belong to the group of messages defined by the `Set` interface known as the *testing* category. There are two other simple and immediately useful messages in the testing category: `isEmpty()` and `contains()`. The effect of these messages is what you might expect from their names: `isEmpty()` answers with `true` if the receiver is empty, otherwise it answers with `false`. The message `contains()` answers `true` if its argument is contained in the receiver, and `false` otherwise.

The following code demonstrates the effect of sending `isEmpty()` to an empty set:

```
Set<String> herbSet = new HashSet<String>();
boolean result;
result = herbSet.isEmpty();
```

In the above code, the variable `result` would be assigned the value `true`.

The next code example demonstrates the effect of sending `isEmpty()` to a non-empty set:

```
Set<String> herbSet = new HashSet<String>();
boolean result;
herbSet.add("Parsley");
herbSet.add("Sage");
herbSet.add("Rosemary");
result = herbSet.isEmpty();
```

In the above code, the variable `result` would be assigned the value `false`.

This next code example demonstrates how to use the message `contains()` to determine whether a set includes a particular element:

```
Set<String> herbSet = new HashSet<String>();
boolean result;
herbSet.add("Parsley");
herbSet.add("Sage");
herbSet.add("Rosemary");
result = herbSet.contains("Sage");
```

In the above code, the variable `result` would be assigned the value `true`.

ACTIVITY 2

Open Unit10_Project_1 then open the `SetExamples` class in the editor. Now take a look at the `emptyDemo1()`, `emptyDemo2()` and `containsDemo()` class methods, paying attention to both the method comments and the code.

In the OUWorkspace execute the following statements one by one:

```
SetExamples.emptyDemo1();
SetExamples.emptyDemo2();
SetExamples.containsDemo("Rosemary");
SetExamples.containsDemo("Oregano");
```

From the descriptions we have given of the messages `contains()` and `isEmpty()`, are the results shown in the Display Pane as expected?

DISCUSSION OF ACTIVITY 2

The set is at first empty and then populated. `"Rosemary"` is present but `"Oregano"` is not. So we would expect the following:

```
The set is empty is: true
The set is empty is: false
The set contains Rosemary is: true
The set contains Oregano is: false
```

Removing elements

The message `remove()` removes an element from a set object. The message answers with `true` if its argument was found and removed, and `false` if it was not found.

These message replies can be very useful. For example, if an element cannot be removed because it was not present, then for some applications it might be important to take some action such as warning the user. This is illustrated by the following code:

```
Set<String> herbSet = new HashSet<String>();
herbSet.add("Parsley");
herbSet.add("Sage");
herbSet.add("Rosemary");
Boolean removed = herbSet.remove("Parsley");
if (removed)
{
    OUDialog.alert("Herb removed");
}
else
{
    OUDialog.alert("Herb not found");
}
```

ACTIVITY 3

Open Unit10_Project_1 and again open the `SetExamples` class in the editor. Now take a look at the class `removeDemo()` method, paying attention to both the method comment and the code.

Remember, each time you invoke `removeDemo()` a new set of three strings is created.

In the OUWorkspace invoke the method `removeDemo()` on the class `SetExamples` a number of times, giving `removeDemo()` different string arguments each time in order to try removing elements from `herbSet` that are and are not there respectively. What behaviour results?

DISCUSSION OF ACTIVITY 3

Here is one possible answer:

```
SetExamples.removeDemo("Sage");
```

This element will be removed successfully and the size of the set will be reduced by one.

```
SetExamples.removeDemo("Dill");
```

Executing this statement will result in a warning that `"Dill"` is not present, and the size of the set will be unchanged.

1.3 Primitive data types and collections

So far, we have focused on sets of strings, but sets whose element type is any kind of class behave in essentially the same way. However, sets must hold *objects*; the element type cannot be `int` or any other primitive data type. The following is not legal:

Arrays can hold either objects or primitive data values but this is not true of collections from the Java Collections Framework.

```
Set<int> numSet = new HashSet<int>(); //will not compile!
```

Does this mean that we cannot use sets (or any other classes from the Collections Framework) to store values which represent integers (or characters, Booleans, floating-point numbers and so on)? That would be dreadfully inconvenient. We use integers all the time, for example, and we very much want to store them in general collections, not just in arrays.

Fortunately there is a way round the difficulty.

Wrapper classes

Luckily for every primitive type, there is a corresponding class of objects we can use to 'wrap' values of that type. For example, the wrapper class for `int` is `Integer`.

Other wrapper classes are `Long`, `Float`, `Double`, `Short`, `Byte`, `Character`, and `Boolean`. The corresponding primitive data types will be clear from the names.

Given a primitive data value, all we need to do is wrap it in an object of the appropriate wrapper class and then we can store the wrapper object in collections exactly like any other object.

You can create an instance of `Integer` that wraps the primitive value `7` as follows:

```
Integer wrappedNum = new Integer(7);
```

You can extract the primitive value again like this:

```
int unwrappedNum = wrappedNum.intValue();
```

This wrapping and unwrapping is fairly straightforward, but when we use numbers we generally want to do arithmetic with them. The arithmetic cannot be carried out on the `Integer` objects directly, so we have to unwrap the primitive values first, then do the calculation, and then wrap the answer back up again. This can involved a lot of programming overhead. However, prior to Java 1.5 there was no other option.

You will be glad to learn that from Java 1.5 onwards, the wrapping and unwrapping is done for us *automatically*. This is referred to as **autoboxing**. Most of the time it is transparent and we do not have to worry about the distinction between `int` and `Integer`, so we can write things like:

We have used `int` and `Integer` as an example, but Java autoboxes any other primitive data type in a similar way.

```
Set<Integer> myNumbers = new HashSet<Integer>();
myNumbers.add(7);
```

and the compiler will translate them into bytecode that takes care of everything for us. It is as though primitive values can be added directly into the collection – although really, behind the scenes, wrapper objects are being used.

In fact, if we try to do arithmetic directly with `Integer` objects, Java 1.5 again supports us. The following works well:

```
Integer num1 = 23;
Integer num2 = 47;
Integer num3 = num1 + num2;
```

It is most important to realise what is happening here though; the `int` values are still being unwrapped, added, then the answer is being wrapped up. It is just that we do not have to worry about the details ourselves.

Since autoboxing was not present in Java until 1.5, you will still see a lot of code around where the wrapping and unwrapping of primitives has been done 'by hand'.

More about autoboxing

Suppose we have an `Integer` and an `int` value as follows:

```
int aPrimitive;
Integer anObject = 100;
Integer anotherObject;
```

Then in the following statement the autoboxing will automatically unwrap (sometimes called *unbox*) the primitive value and assign it to `aPrimitive`:

```
aPrimitive = anObject; // unwrap primitive
```

If we now assign the primitive back to an `Integer` variable, it will automatically be wrapped again:

```
anotherObject = aPrimitive; // wrap primitive
```

As we have explained above, in the case of collections the key point about autoboxing is that, even though collections cannot literally contain primitive types, autoboxing makes it very easy for collections to contain and work with objects that wrap up values of primitive data types.

SAQ 2

Consider a set declared as follows:

```
Set<Integer> numSet = new HashSet<Integer>();
```

We can add some primitive `int` values to this set as follows:

```
numSet.add(5);
numSet.add(6);
numSet.add(5);
```

The effect of evaluating these statements will be that the primitive values will be autoboxed as `Integers` automatically before they are added to the set. What will the class of each element of `numSet` be after the above statements have been evaluated?

ANSWER.....

The class of each element of `numSet` will be `Integer`.

Exercise 1

You should try the following as a paper and pencil exercise without using a computer. Consider the following method:

```
/**
 * Demonstrates the creation of a set of Integers and
 * the subsequent adding and removing of elements.
 */
public static void integerSetDemo(int anInt)
{
    Set<Integer> numberSet = new HashSet<Integer>();
    numberSet.add(anInt);
    numberSet.add(1);
    numberSet.add(4);
    numberSet.add(5);
    numberSet.remove(6);
    numberSet.remove(1);
}
```

Suppose the following lines of code are executed in turn in the OUWorkspace:

```
SetExamples.integerSetDemo(1);
SetExamples.integerSetDemo(6);
SetExamples.integerSetDemo(2);
```

In each case describe the resulting state of `numberSet`.

Solution

After

```
SetExamples.integerSetDemo(1);
```

the set contains `Integer` objects that wrap the `int` values `4` and `5`.

After

```
SetExamples.integerSetDemo(6);
```

the set contains `Integer` objects that wrap the `int` values `4` and `5`.

After

```
SetExamples.integerSetDemo(2);
```

the set contains `Integer` objects that wrap the `int` values `2`, `4` and `5`.

Now that you know about autoboxing, you should be able to deal with sets having any kind of contents you like. However, so far all we have really done is put things in and take things out (and do various tests on the contents).

To exploit the power of sets more deeply, we need the ability to work our way through a set, operating on each element in the set as we go. For example, we might want to print the textual representation of each element on a separate line in the Display Pane. To do this we must *iterate* through the set.

1.4 Iterating over a set

Sometimes we need to iterate through a set, processing each element in a uniform way.

In *Unit 9*, in the case of arrays, you saw the easiest way of doing this was the `foreach` statement (also known as the enhanced `for` statement which allows you to do the same thing to every element of an array. The `foreach` statement works in exactly the same way with sets and the other collections in the Collections Framework.

The following code, taken from the `iterationDemo()` method of the class `SetExamples`, illustrates iteration over a set using a `foreach` statement:

```
Set<Integer> numSet = new HashSet<Integer>();
numSet.add(12);
numSet.add(9);
numSet.add(88);
int  total = 0;
for (Integer eachNum : numSet)
{
    total = total + eachNum;
}
```

Recall that the header of the `foreach` statement is pronounced as follows:

'For each `Integer eachNum` in `numSet`.'

The `foreach` statement block will be executed once for each `Integer` element in the set `numSet` and each time the variable `eachNum` will reference that element. So after the final iteration the variable `total` will hold the sum of all the integers in the set.

SAQ 3

If the `foreach` statement above declared its local variable with the identifier `each` instead of `eachNum`, would there would be a compilation error?

ANSWER.....

No. It does not matter what identifier is used for a `foreach` statement's local variable However, you might want to argue that calling the variable `eachNum` in this case makes it easier for the human reader to understand what is happening.

SAQ 4

(a) Why has the `foreach` statement's local variable `eachNum` been declared of the `Integer` class, when the code prior to the `foreach` statement has been adding values of the primitive type `int` to the set?

(b) In a `HashSet`, can we predict with certainty in what order the elements will be iterated over?

ANSWER.....

(a) When an attempt is made to add a primitive type to any class in the Collections Framework autoboxing is used, that is, the primitive is automatically wrapped in an instance of its appropriate wrapper class – here `Integer` – therefore all the elements in the set are instances of the `Integer` class.

(b) No. The order is undefined and there is no way to predict it.

Something that you cannot do using a `foreach` loop is remove elements. Nor is it possible to replace elements as we go.

Just as with arrays, it is possible to iterate through a set using a hand-coded `for` loop instead of a `foreach` loop. However, for a set this is less straightforward. With an array we can simply use an `int` counter to represent the numerical index of each element in turn and increment the index each time the loop is executed.

Exercise 2

Why will this plan, which works for an array, not work for a set?

Solution

Sets have no index and so there is simply no way to access the elements of a set using an integer counter. Java provides a way of getting round this by providing helper classes that implement the `Iterator` interface, but this is beyond the scope of this unit.

ACTIVITY 4

The following method creates a set of strings with each element being the name of a herb. The last line of the method then prints to the standard output (in our case the OUWorkspace's Display Pane) the size of the set.

```
public static void herbSet()
{
    Set<String> herbSet = new HashSet<String>();
    herbSet.add("camomile");
    herbSet.add("rosemary");
    herbSet.add("basil");
    herbSet.add("thyme");
    herbSet.add("mint");
    herbSet.add("sage");
    herbSet.add("lovage");
    herbSet.add("parsley");
    herbSet.add("chives");
    herbSet.add("marjoram");
    System.out.println(herbSet.size());
}
```

What if, instead of printing out the size of the set, we wanted to print out the contents of the set, string by string?

Open Unit10_Project_1 and in the `SetExamples` class create a modified version of this class method, called `herbIterationDemo()`. Within the statement block of a `foreach` statement the method should print out to the Display Pane the name of each herb. Remember to declare your method as `static`.

Once you have written your method and got the `SetExamples` class to compile, try out your method in the OUWorkspace and observe the results in the Display Pane.

DISCUSSION OF ACTIVITY 4

Here is our solution:

```
public static void herbIterationDemo(),
{
    Set<String> herbSet = new HashSet<String>();
    herbSet.add("camomile");
    herbSet.add("rosemary");
    herbSet.add("basil");
    herbSet.add("thyme");
    herbSet.add("mint");
    herbSet.add("sage");
    herbSet.add("lovage");
```

```
        herbSet.add("parsley");
        herbSet.add("chives");
        herbSet.add("marjoram");
        for (String herb : herbSet)
        {
            System.out.println(herb);
        }
    }
```

1.5 `TreeSet` – a set with order

So far, we have emphasised the properties of sets that make them *helpful*, such as the dynamic sizing and the rejection of duplicates. Up until now, the *lack of ordering* provided by sets has not made much difference, for what we have been doing.

However, suppose we wanted to have the herbs printed out alphabetically. It turns out that here is another kind of set, `TreeSet`, which *does* have order. If we change *one thing* in the above code, the herbs will be sorted. All we need to do is just alter the actual set class used from `HashSet` to `TreeSet`. The variable, which is declared as an interface type, can stay exactly the same.

ACTIVITY 5

Open Unit10_Project_1. Double-click on the icon for the class `SetExamples` to open the editor. In the class `SetExamples` create a modified version of `herbIterationDemo()`, called `sortedHerbIterationDemo()`, that prints out the names of the herbs in alphabetical order.

Test your method from the OUWorkspace.

DISCUSSION OF ACTIVITY 5

All we need do is alter

```
Set<String> set = new HashSet<String>();
```

to

```
Set<String> set = new TreeSet<String>();
```

The herbs will now automatically be in order. This begins to illustrate some of the power of the Java Collections Framework.

If you had any problems with Activities 4 and 5, the new methods have been added to the class `SetExamples` *in Unit10_Project_2.*

An instance of `TreeSet` will correctly order not just strings but other kinds of elements with a well-defined natural order, such as integers. We can also make `TreeSet` work with elements ordered according to some rule defined by us, although we shall not explore this.

`TreeSet` actually implements the `SortedSet` interface which is a subinterface of `Set`.

Although `TreeSet` and `HashSet` differ in that instances of one hold the elements in a particular order and instances of the other do not, most of their behaviour is identical – both classes implement the `Set` interface.

This is a useful point at which to summarise some of the abstract method definitions defined by the `Set` interface whose implementation by the `HashSet` and `TreeSet` classes we have explored.

Table 1 A summary of some useful methods specified by the `Set` interface

Method definition	Category	Description
`public int size()`	testing	Returns the number of elements in the receiver
`public boolean add(ElementType obj)`	adding	Adds the argument to the receiver, unless it is already there. Returns `true` if the argument was added, `false` if it was not
`public boolean isEmpty()`	testing	Returns `true` if the set is empty, otherwise `false`
`public boolean contains(Object obj)`	testing	Returns `true` if the argument is an element in the receiver, `false` if not
`public boolean remove(Object obj)`	removing	If the argument is an element in the receiver it is removed and the method returns `true`. If the argument is not an element in the receiver it remains unchanged and the method returns `false`

Table 1 summarises some of the most useful messages that sets can respond to. Note that the various messages are often grouped into informal categories such as adding, removing, and testing, etc.

The method `add()` is declared to have an argument type of type `ElementType` – this is simply a placeholder for the actual element type that a particular set will have been declared to have.

SAQ 5

In Table 1, the arguments for the methods `contains()` and `remove()` are declared as being of type `Object` – can you think why this might be?

ANSWER.....

The reason is to do with type substitution: if a method header declares a formal argument to be of type `Object`, then an instance of any class can be used for the actual argument in a corresponding message. All classes inherit from the class `Object`, therefore an object of any class can be used where an instance of `Object` is expected.

Sets, and indeed instances of any collection class can hold instances of any class, therefore it makes sense that these arguments are declared of type `Object`.

Next we will look at a different interface – `Map`.

2 Map – a collection with keys and values

One familiar way of structuring information is the index of a book, with entries such as the following (but usually in alphabetical or some other order):

blueberry	42
blackcurrant	76
lemon grass	11
earl grey	3, 17
cinnamon	11
rosehip	42
raspberry	76, 32
mango and ginseng	18

This extract from a book index has entries consisting of a **key** (the thing you look up) and a **value** (what you find there). For example, 42 is the value corresponding to the key *rosehip*. This way of structuring information is the motivation behind the Java collection known as a **map**. In order to be able to use maps correctly, we need to clarify the idea of 'key' a little more precisely.

2.1 The map, the key and the value

Imagine a computer system used by a publisher to keep track of every book edition that they publish. Such a system might be based on some kind of collection of different instances of a class `BookEdition`, with one instance of `BookEdition` used to represent each book edition published by the firm.

Now, it is perfectly possible, in fact common, for a single firm to publish *different editions* of a single title by a single author (for example, hardback, paperback, etc.). Consequently, merely knowing the book title and author of a book is not always enough to *uniquely identify* a specific edition. Hence, for the purpose of identifying book editions, book title and author alone would not make a sensible key. A better way of uniquely identifying book editions is the International Standard Book Number (ISBN). For distinguishing between book editions, ISBN numbers make good keys.

(Note though that the ISBN would be no good as a key for distinguishing *your* copy of the same edition from *my* copy of the same edition – as they will both have the same ISBN. So, what makes a good key depends on your purposes. It depends on exactly what you need to distinguish.)

Thus a **key** is simply something that can be used to *uniquely identify any object from a given collection*, in a way that meets the needs of a particular application.

What makes a good key may not be immediately obvious. For example, in the case of a directory of telephone extension numbers for a small firm, names alone might make perfectly good keys, because they are all unique. But in a telephone directory for a city the name of a person alone would not be a very good key, because there are likely to be several people with the same name. The combination of name and address would be a better key (good enough for most practical purposes, although there may still be two people with the same name at the same address).

Notice that although the directory should ideally provide some way to identify each person listed uniquely, there is nothing wrong with several people all having the same

telephone number (for example, this could easily happen for members of a single family, flatmates, etc.). Although we want the keys to be unique, there is no reason whatsoever why several keys cannot correspond to the same value.

SAQ 6

For each of the following, say whether or not it makes an acceptable key. If you think it would, suggest what value the key might be associated with.

(a) A works number for employees in a company.

(b) The name of your Open University tutor amongst the body of OU tutors.

(c) Your Open University student identifier amongst all OU students.

ANSWER.....

(a) This would normally be a unique key. The corresponding value might be an employment record or personnel file, etc.

(b) This choice of key might be dangerous for some courses. There are some OU tutors with the same name. A likely value would be a tutorial group.

(c) This is a unique key. The value might be marks for a given course, or the student record, etc.

In summary, when devising any scheme for keys for objects in some group, the vital consideration is that each key should be unique within the collection.

Having clarified the notion of a key, we can now be a bit more precise about the idea of a map. In essence, a map is a collection of **key–value pairs**. From the above general discussion of keys, it should be clear that within a given map instance, each key must be unique. However, as noted already there is no problem if the same *value* appears for more than one key.

The next exercise looks at two examples which are candidates for being a map.

Exercise 3

Which of the two following telephone directories could be represented directly as a map? Give a reason for your answer.

Directory A

Name	Telephone number
Noggin the Nog	3666
Queen Nooka	3666
Nogbad the Bad	3667

Directory B

Name	Telephone number
Capt Pugwash	6001
Tom the Cabin Boy	5204
Pirate Jake	0667
Pirate Jake	0668

Solution

There is no problem with Directory A. Noggin the Nog and Queen Nooka are two keys with the same value, but this is perfectly allowable for maps.

Directory B cannot be used *directly* as a map, since Pirate Jake's name is not unique in this table, and so names cannot act as unique keys for extension numbers.

We could get round this by changing the type of the values to be stored in the map from *individual* extension numbers into *sets* of extension numbers. Thus, Pirate Jake could still have two telephone numbers while everyone else had a single number, but now each key would have a single value, which was a *set* of numbers, even though the majority of sets just had one number in them. This approach is illustrated in the table below.

Name	Telephone number
Capt Pugwash	6001
Tom the Cabin Boy	5204
Pirate Jake	0667, 0668

Having discussed the underlying ideas, we will now investigate the practical detail of how maps are dealt with in Java.

The `Map` interface: dealing with collections of keys and values

For collections of key–value pairs where values need to be looked up using keys, there is a specially designed interface called `Map`, and a set of implementing classes, of which the most generally useful is `HashMap`.

We saw above that a simple example of a map is a telephone directory where the names are the keys, and the telephone numbers are the values. However, the values and the keys in a map can be any type of object, provided that each key is unique. In line with our usual convention, we shall often use the shorthand phrase 'a map' to mean 'an instance of a class satisfying the interface `Map`'.

Before creating a map

When we created a set we had to specify the type of the elements that would be stored in it. Before creating an instance of map, it is necessary to specify *two* things – the type of the keys and the type of the values to be stored.

These types could easily be different from each other. For example, to keep track of customers' bank accounts, you might use strings containing account numbers as keys, and instances of `Account` as values.

As well as declaring the type of the elements, we must choose some specific class that implements the map interface. As already noted, a good general-purpose choice is `HashMap`.

To declare an instance of `HashMap` whose keys are strings, and whose values are accounts, a statement such as the following would be used.

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

SAQ 7

In the above statement, why is the declared type of the variable – `Map` – different from the implementing class – `HashMap`?

In the above statement, what is the purpose of the text in the angle brackets?

ANSWER.....

As we noted with `Set`, it is generally good practice to use a collection *interface* as the type of the variable. This promotes flexibility and maintainability (in the same spirit as when we earlier changed a `HashSet` to a `TreeSet` while keeping everything else exactly the same).

The text in the angle brackets allows the compiler to check that any implementing map assigned to the variable has keys and values of the correct types.

We will now explore the basic operations on maps using a series of class methods, in the same way as we investigated sets. All the methods can be found in the `MapExamples` class.

2.2 Operations on maps

Creating a map, and checking its size

The following statement series creates an instance of `HashMap`. To keep things simple, we have used strings as both keys and values. One of the basic things to do with a map is to test its size – the number of key–value pairs it contains – using the message `size()`. The following code demonstrates this.

```
Map<String, String> phonebook = new HashMap<String, String>();
int size = phonebook.size();
```

As you might expect, the size of a newly created map is zero.

Adding entries to a map

The most significant difference between set and map objects is that a set is a collection of single objects whereas a map is a collection of key–value pairs. A key can be used to add, retrieve or update an associated value. The messages `put()` and `get()` enable this to be done, as we will see below.

Our first step is to add some entries to a newly created map. The message `put()` is used to add a key–value pair (also known as an **entry** or **mapping**) to a map. This is demonstrated below:

```
phonebook.put("Captain Pugwash", "6001");
phonebook.put("Tom the Cabin Boy", "5204");
phonebook.put("Pirate Jake", "0667");
```

The key of the first new entry is `"Captain Pugwash"` and the value is his telephone extension number `"6001"`. You might have wondered why we have represented the telephone numbers as strings, not as integers. The reason is that telephone numbers can have leading zeros, whereas integers cannot – hence our use of strings. This is the same reason why we used strings as account numbers in *Unit 6*.

ACTIVITY 6

Open Unit10_Project_2 and double-click on the `MapExamples` class to open the editor. Now take a look at the `sizeDemo()` and `putDemo()` class methods, paying attention to both the method comments and the code.

1 Open the OUWorkspace and execute statements which try out `sizeDemo()` and `putDemo()`. Check that the results are as expected.

2 Now add a new class method to the `MapExamples` class called `myDemo()` according to the following specification.

(a) The method should create a map referenced by a variable `poetMap`. The keys of this map should be declared as `Integer` and the values as `String`.

(b) The method should then print the string `"The map is empty: "` to the Display Pane followed by `true` or `false`. Whether the map is empty or not can be determined by sending the message `isEmpty()` to the map object.

(c) The method should then add the following entries to `poetMap`.

Key	Value
205646	"Dylan Thomas"
217887	"Stevie Smith"
327886	"Ted Hughes"
433457	"Stevie Smith"
578799	"Seamus Heaney"

(d) The method should then print the string `"The size of the map is:"` to the Display Pane followed by the current size of the map.

Finally test your method by executing `MapExamples.myDemo();` in the OUWorkspace. The map created by the method should have five entries.

DISCUSSION OF ACTIVITY 6

Our solution was as follows:

```
public static void myDemo()
{
    Map<Integer, String> poetMap = new HashMap<Integer, String>();
    System.out.println("The map is empty: " + poetMap.isEmpty());
    poetMap.put(205646, "Dylan Thomas");
    poetMap.put(217887, "Stevie Smith");
    poetMap.put(327886, "Ted Hughes");
    poetMap.put(433457, "Stevie Smith");
    poetMap.put(578799, "Seamus Heaney");
    System.out.println("The size of the map is: " + poetMap.size());
}
```

Retrieving a value

Once some entries are in a map, the message `get()` can be used to retrieve a value associated with a particular key. For example, given the set created by the method `myDemo()` that you wrote for Activity 6, the statement:

```
String poet = poetMap.get(433457);
```

would return the string `"Stevie Smith"` and assign it to the variable `poet`.

If a key is not found

The above example is straightforward – but what happens if we try to retrieve a value from a map using `get()` with a key that does not exist in the map? For example:

```
String poet = poetMap.get(111111);
```

In this case, as `poetMap` does not contain an **entry** with the key `111111`, the message `get()` returns `null`, which is then assigned to `poet`.

The converse – in a sense – of looking for a key that is not in a map is trying to insert an entry (a key–value pair) into a map when the key *already exists* in the map. This does not cause any kind of problem – far from it. Instead the value corresponding to that key is *updated*. This is demonstrated by the code below:

```
poetMap.put(111111, "John Cooper Clarke");
poetMap.put(111111, "William Wordsworth");
```

Putting two different entries into a map with the key `111111` is allowed – there is no error or exception. However, this does not result in two different entries both with the key `1111111` being added to the map. Instead the second `put()` message simply overwrites the value of the entry with the key `111111` with the new value `"William Wordsworth"` and the value `"John Cooper Clarke"` then no longer exists in the map.

This behaviour will only be a problem if we update an entry by mistake and cannot recover the original value.

Removing entries from a map

The message `remove()` takes a key as its argument and returns the object associated with that key, or null if the key is not present in the map. This is demonstrated by the following code:

```
Map<String, String> phonebook = new HashMap<String, String>();
phonebook.put("Captain Pugwash", "6001");
phonebook.put("Tom the Cabin Boy", "5204");
phonebook.put("Pirate Jake", "0667");
phonebook.remove("Pirate Jake");
```

The final line of code will remove the key–value pair `"Pirate Jake"` / `"0667"` from the map.

If you try to remove an entry from a map for which no matching key is present, the code will execute without any problem as `remove()` deals with missing keys in the same way as the message `get()` – it simply returns `null`.

This is a good moment to return to the method `myDemo()` that you created earlier and get some hands-on experience with these new map messages.

ACTIVITY 7

1 Open Unit10_Project_2 and double-click on the `MapExamples` class to open the editor. Now take a look at the `getDemo()`, `keyNotFoundDemo()`, `updateEntryDemo()`, `removalDemo()` and `failedRemovalDemo()` class methods, paying attention to both the method comments and the code. Then, in the OUWorkspace execute statements that invoke these methods on the `MapExamples` class. Check that the results shown in the Display Pane are as expected.

2 Next extend your method `myDemo()` in the `MapExamples` class by adding the following steps at the end of the method. Use messages from the protocol of `Map` as appropriate to carry out the operations.

   (a) Your method should first print out `"The value associated with the key 578799 is: "` followed by the associated value.

   (b) Your method should then print out `"The value associated with the key 123456 is: "` followed by the associated value.

   (c) Your method should then add an entry with key `578799` and value `"Brian Patten"`.

   (d) Your method should then print out `"The value associated with the key 578799 is: "` followed by the associated value.

   (e) The method should remove any entry whose key is `433457`, and print out a message showing what value was found associated with this key.

Try out the modified method by invoking it on the `MapExamples` class from the OUWorkspace.

DISCUSSION OF ACTIVITY 7

The code we added was as follows:

```
System.out.println("The value associated with key 578799 is: "
                   + poetMap.get(578799));
System.out.println("The value associated with key 123456 is: "
                   + poetMap.get(123456));
poetMap.put(578799, "Brian Patten");
System.out.println("The value associated with key 578799 is: "
                   + poetMap.get(578799));
String value = poetMap.remove(433457);
System.out.println(value);
```

We will finish our introduction to maps with three more messages that can be sent to map objects. The messages `isEmpty()`, `containsKey()` and `containsValue()` all make simple tests of a map, and return a Boolean value to indicate the result.

- `isEmpty()` (already seen in Activity 6) tests for whether a given map is empty.
- `containsKey()` tests whether the map has a key that is equal to its argument.
- `containsValue()` tests whether the map has a value equal to its argument.

These are all fairly straightforward, and some demonstration methods can be found in the `MapExamples` class in Unit10_Project_2.

ACTIVITY 8

1 Open Unit10_Project_2 and double-click on the `MapExamples` class to open the editor. Now take a look at the class method `containsDemo()`, paying attention to both the method comments and the code. Then, in the OUWorkspace execute a statement that invokes this method on the `MapExamples` class. Check that the results shown in the Display Pane are as expected.

2 Extend your method `myDemo()` further, making the following changes.

The method should first print out the string `"The map contains an entry for key 217887: "` followed by `true` or `false`. This information should be obtained using a message from the protocol of `Map`.

Your method should then print the string `"The map contains an entry whose value is Ted Hughes: "` followed by `true` or `false`. The required information should be obtained using a message from the protocol of `Map`.

Try out the modified method by invoking it on the `MapExamples` class from the OUWorkspace.

DISCUSSION OF ACTIVITY 8

The additional lines of code are as follows:

```
System.out.println("The map contains an entry for key 217887: "
                   + poetMap.containsKey(217887));
System.out.println("The map contains an entry whose value is Ted Hughes: "
                   + poetMap.containsValue("Ted Hughes"));
```

If you had any problems with Activities 6, 7 and 8, the final code for `myDemo()` *has been added to the class* `MapExamples` *in Unit10_Project_3.*

2.3 Iterating over maps

As with any kind of collection, sometimes we need to work our way through all the entries in a map, processing each entry as we go.

Earlier you saw how to iterate over sets using a `foreach` loop. The way a `foreach` loop works with an instance of `Map` is just the same in principle as iterating over a set, but is slightly different in detail. There is more than one way to iterate over a map, but we will focus on the simplest.

The chief difference between sets and maps for the purpose of iteration is that, in the case of a set, each element is a single object. In the case of a map, each element will be an entry, that is, a **key–value pair**.

For this reason, when iterating over a map, depending on what exactly is required for the purposes of the program, it may be necessary to extract both the key and value for each entry. This makes maps potentially more complicated to iterate over than simple sets. However, you will be glad to learn there is a simple way to avoid this potential complexity.

Classes which implement the `Map` interface have two very useful messages `keySet()` and `values()` in their instance protocols. The first of these returns a `Set` containing *all* the *keys* of the map, and the second returns a `Collection` containing *all* the *values* of the map.

In order to access each entry (key–value pair) in a map in turn, we first get the set of keys using `keySet()`, then iterate over that set of keys, as demonstrated below.

```
Map<String, String> dictionary = new HashMap<String, String>();
dictionary.put("one", "ichi");
dictionary.put("two", "ni");
dictionary.put("three", "san");
dictionary.put("four", "shi");
Set<String> keys = dictionary.keySet();
for (String eachEnglishWord : keys)
{
    System.out.println("The Japanese for " + eachEnglishWord
                       + " is " + dictionary.get(eachEnglishWord));
}
```

The statement `Set<String> keys = dictionary.keySet();` assigns a set containing all the keys in `dictionary` to the variable `keys`. Next, the statement block controlled by the `foreach` statement is evaluated four times because there are four keys in the set referenced by `keys`. Each time the statement block is evaluated, the `foreach` variable, `eachEnglishWord`, references a successive element of the set `keys`.

The code in the statement block uses each successive element referenced by `eachEnglishWord` as a key to access the map `dictionary` to retrieve the associated value (which is then appended to a string and printed out).

Note that because sets are unordered, there is no way of knowing in what order each English word will reference the elements of dictionary keys.

Although the above `foreach` loop is iterating over a set of keys, the effect is that of iterating over the map as well. In this way, iterating over a map can be made as simple as iterating over a set. If you need to iterate over a map, all you need do is obtain the set of keys and iterate over that. The next activity will give you some hands-on experience of this approach.

ACTIVITY 9

Open Unit10_Project_3 and double-click on the icon for the `MapExamples` class to open the editor. Scroll through the class to find the class method `iterationDemo()`. Read the method comment and the code and then in the OUWorkspace invoke the `iterationDemo()` method on the `MapExamples` class. Check that the results shown in the Display Pane are as expected.

Now write a new class method for the `MapExamples` class called `averageAgeIterationDemo()`. In this method create a map referenced by a variable `ageMap`, whose keys are strings and whose values are integers, and then add the following entries to `ageMap`:

Key	Value
"Jill"	33
"James"	45
"Louise"	45
"Peter"	26
"Sally"	40

Create an integer variable `totalAge` whose initial value is `0`. Using the message `keySet()` get the set of the map's keys. Then iterate over that set. For each key in the set, look up the corresponding age in `ageMap` and add it to the total age. When the iteration is complete, calculate the average age by dividing the total age by the size of the map (which corresponds to the number of entries). Print the average age to the Display Pane.

DISCUSSION OF ACTIVITY 9

One solution is as follows (comment omitted for brevity):

```
public static void averageAgeIterationDemo()
{
    Map<String, Integer> ageMap = new HashMap<String, Integer>();
    ageMap.put("Jill", 33);
    ageMap.put("James", 45);
    ageMap.put("Louise", 45);
    ageMap.put("Peter", 26);
    ageMap.put("Sally", 40);
    int totalAge = 0;
    for (String eachName : ageMap.keySet())
    {
        totalAge = totalAge + ageMap.get(eachName);
    }
    int averageAge = totalAge/(ageMap.size());
    System.out.println("The average age is: " + averageAge);
}
```

The code for `averageAgeIterationDemo()` *has been added to the class* `MapExamples` *in Unit10_Project_4.*

Sending `keySet()` to a map does not in any way disturb the keys or values in the original map (that is, it is a *non-destructive operation*). However, although the set of keys is a new collection, it is important to note that the elements in the set are the actual keys from the original map, not copies. Consequently should you remove a key from the set, the corresponding **key–value pair** will be removed from the map. The reverse is also true: removing a key–value pair from the map will remove the key from the set.

Similarly the collection of values returned from sending the message `values()` to a map are also the actual values in the map. Therefore removing a value from the collection will result in the corresponding key–value pair being removed from the map.

SAQ 8

(a) The message `keySet()` when sent to a map will collect all the map's keys into a set. The nature of sets will demand that only unique elements are stored in the set – any duplicate keys would be lost. How can the loss of duplicate keys be prevented?

(b) Whenever you send the message `keySet()` to a map, a new set is returned. However, the keys held in this new set are not copies of the keys in the map, they are the actual keys. If you subsequently removed a key from the set of keys what would happen to the key in the original map?

ANSWER.....

(a) There is no such possible danger. The keys in a map must be unique in the first place – no duplicate keys are allowed.

(b) The key and its corresponding value would be removed from the original map.

This is a useful point at which to summarise some of the abstract method definitions defined by the `Map` interface and whose implementation by the class `HashMap` we have so far explored.

2.4 A summary of useful `Map` methods

The table below summarises some of the most useful messages that maps can respond to. Note that, as for sets, the various messages are often grouped into informal categories such as adding, removing, testing, etc.

In some ways the protocol of `Map` resembles that of `Set`, but there are vital differences. For a start, maps have to deal with both keys and values, whereas sets just have simple elements. Also, because `Map` is an indexed collection, we have a new method category, accessing.

Table 2 Summary of some useful methods specified by the `Map` interface

Method definition	Category	Description
`size()`	testing	Returns the number of elements in the map (i.e. the number of key–value pairs).
`put(KeyType key, ValueType value)`	adding	Puts a key–value pair into the map. If a key–value pair already exists with the same key, then the old value is overwritten by the new value. The method returns the previous value for the key, if there was one, otherwise it returns `null`.
`get(Object key)`	accessing	If the key exists in the map, then the method returns the associated value. Otherwise `null` is returned.
`keySet()`	accessing	Returns the set of keys contained in the map.
`values()`	accessing	Returns a collection of the values contained in the map.
`remove(Object key)`	removing	Removes the key–value pair associated with the specified key (if one exists). Returns the previous value for the key, if there was one, otherwise `null`.
`isEmpty()`	testing	Tests whether the map contains any key–value pairs. The method returns `true` if the map is empty, otherwise `false`.
`containsKey (Object key)`	testing	Tests whether a given key is present. The method returns `true` if the key is present, `false` otherwise.
`containsValue (Object value);`	testing	Tests whether a given value is present for one or more keys. The method returns `true` if the value is present, `false` otherwise.

The method `put()` is declared to have two arguments, one of type `KeyType` and one of type `ValueType` – these type names are simply placeholders for the actual key and value type names given between angled brackets when a map is declared and created.

SAQ 9

(a) What is the main difference between maps and sets?

(b) A collection is needed in an Open University system to store student identifiers with the current total of OU credits each student has studied. Would a map be an appropriate collection interface for this task? Give a reason for your answer.

ANSWER

(a) The elements of a set are single objects whereas the elements of a map are key–value pairs.

(b) Student identifiers are unique and so make good keys. Total credits make legitimate values. So a map would be a good collection interface for this task.

Exercise 4

Which of the following statements are true and which are false? If you think a statement is false, give a reason.

(a) The message `put()` always enters a new entry in a map.

(b) Each entry in a map has a key and one or more values.

(c) Duplicate keys are allowed in a map.

(d) The message `get()` returns the value of some entry within a map which has a key that matches the message's argument.

Solution

(a) False. It may update an exiting entry.

(b) False. An entry has only one value.

(c) False. Keys are unique in a map.

(d) True. This is exactly what `get()` does.

3 Overview of the Collections Framework

Now that you have investigated some of the collection classes that are available in Java it is time to step back a little and look at the general structure of the Collections Framework and how everything fits together.

So far in this unit you have investigated three collection classes: `HashSet`, `TreeSet` and `HashMap`. In fact, there are *many* different collection classes available in Java. Indeed, there are so many that it would take a very long time to investigate them all.

Fortunately, there is a wonderful shortcut in Java that cuts down hugely on what you need to learn about collection classes, a shortcut which you have already come across. The key to this is to focus, where possible, not on collection *classes* at all, but on collection *interfaces*. There are only a relatively small number of different collection interfaces, and it is relatively easy to learn the basics of each. Once you have done this, you can use collections most of the time without having to worry too much about the details of various collection classes that implement the interfaces. You already know much about two of the most important collection interfaces, `Set` and `Map`.

3.1 The collection interfaces

The Collections Framework specifies a set of collection interfaces (those shown in Figure 3), recommends some constructors for the classes that implement them and provides some utility classes. These, together with collection classes meeting these specifications are said to constitute the Collections Framework.

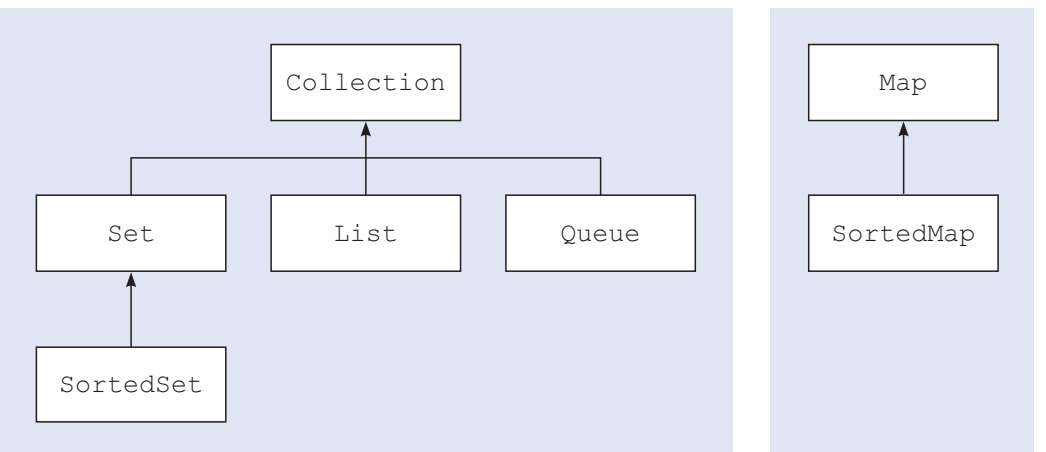


Figure 3 The principal collection interfaces

Each box in the above diagram represents one collection interface. As you may recall from *Unit 6*, an interface is essentially a list of methods without any actual implementation, in effect they are abstract methods. For any class which undertakes to implement an interface, the list is like a specification, listing methods that need to be implemented or inherited by that class. You have already seen some of the methods in the interfaces for sets and maps itemised in Table 1 and Table 2 earlier. An interface does not, and cannot, define any specific behaviour to go with any of the methods it lists. Nevertheless, by choosing appropriate method names and by appropriate accompanying documentation, a programmer who writes an interface can indicate what *general kinds of action* are expected from the implementation.

One of the interesting things about Figure 3 is that it is a tree diagram, that is, some interfaces are shown as the children of other interfaces. For example, the `SortedSet` interface is shown as a child of the `Set` interface. We say that `SortedSet` is a **subinterface** of `Set`. Simply by the fact of this relationship, we know that `SortedSet` automatically inherits every abstract method specified in `Set`, and has other additional ones – as you will discover in Activity 10, the `SortedSet` interface defines two rather useful abstract methods, `first()` and `last()`. The method `first()` returns the first element in a sorted set, while `last()` returns the last element in a sorted set. Similarly, as you can see from Figure 3, `SortedMap` is a subinterface of `Map`.

SAQ 10

Name a method in `SortedSet` that is not in `Set`.

ANSWER

Two good answers are `first()` and `last()`. Other answers are possible as you will see if you inspect the Java documentation.

The `Collection` interface is a **superinterface** of all of the collection interfaces dealt with in this unit (except for the collections in the `Map` interface family). All of the abstract methods in `Collection` are automatically part of the interfaces `Set`, `List`, `Queue`, etc. The `Collection` interface specifies methods such as `add()`, `size()` and `isEmpty()` which are common to very many kinds of collection. Table 3 summarises some of the principal abstract methods defined in the `Collection` interface.

Table 3 Principal methods specified by the `Collection` interface

Method	Category	Description
`add(ElementType obj)`	adding	Adds the argument to the collection. Returns `true` if the operation was successful, `false` otherwise.
`addAll(Collection <ElementType> aCol)`	adding	Adds all of the elements in the argument to the receiver. Returns `true` if the operation was successful, `false` otherwise.
`remove(Object obj)`	removing	Removes one occurrence of the argument from the collection. Returns `true` if the operation was successful, `false` otherwise.
`clear()`	removing	Removes all elements from the collection (if there were 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 in the collection, `false` otherwise.
`isEmpty()`	testing	Tests whether the collection is empty. Returns `true` if the set is empty, `false` otherwise.

Java does not contain any concrete classes designed *specifically* to implement the `Collection` interface. However, there are many concrete classes that implement **subinterfaces** of `Collection` such as `Set` and `SortedSet`, and all of these concrete classes necessarily implement the protocol of `Collection`, simply as a result of implementing their respective subinterfaces.

SAQ 11

Do all of the classes in the Collections Framework implement the `Collection` interface?

ANSWER.....

Not all of the collection classes implement the `Collection` interface. Classes such as `HashMap` and `TreeMap` implement the `Map` interface. The `Map` interface is not a subinterface of the `Collection` interface, as can be seen in Figure 3. The separateness of keys and values in any map keeps maps slightly apart from the other collections. For example, the message `add()`, which is part of the `Collection` interface but not in the protocol of `Maps` would need to be dealt with rather differently in the case of a map. Despite this, we say that maps are part of the Collections Framework, since the framework was designed to cover not just the `Collection` interface but also the `Map` interface.

SAQ 12

Name any three methods in the `Set` interface and any three methods in the `Map` interface. At least two should be common to both interfaces.

ANSWER.....

Many correct answers are possible.

`isEmpty()`, `size()`, `clear()`, and `remove()` appear in both the `Set` and `Map` interfaces.

The protocol for `Set` includes `add()`, `contains()` and many others.

The protocol for `Map` includes `get()`, `put()`, `containsKey()`, `containsValue()` and many others.

SAQ 13

We have not yet said anything about the interfaces `List` and `Queue`, but you can be sure that any abstract methods declared in `Collection` must be inherited by the interfaces `Set`, `List` and `Queue`. Why can we be sure of this?

ANSWER.....

We can be sure of this because, as Figure 3 shows, `Set`, `List` and `Queue` are subinterfaces of `Collection`.

3.2 The collection classes

The Collections Framework is very good news for programmers: once you have learnt the basics of a small number of collection interfaces, you can do much of what you need to do with collections without having to learn all the details of every class. Although you do not need to know very much about most of the concrete collection classes, or memorise all of their names, it is useful to glance at a diagram of some of the most important ones (Figure 4).

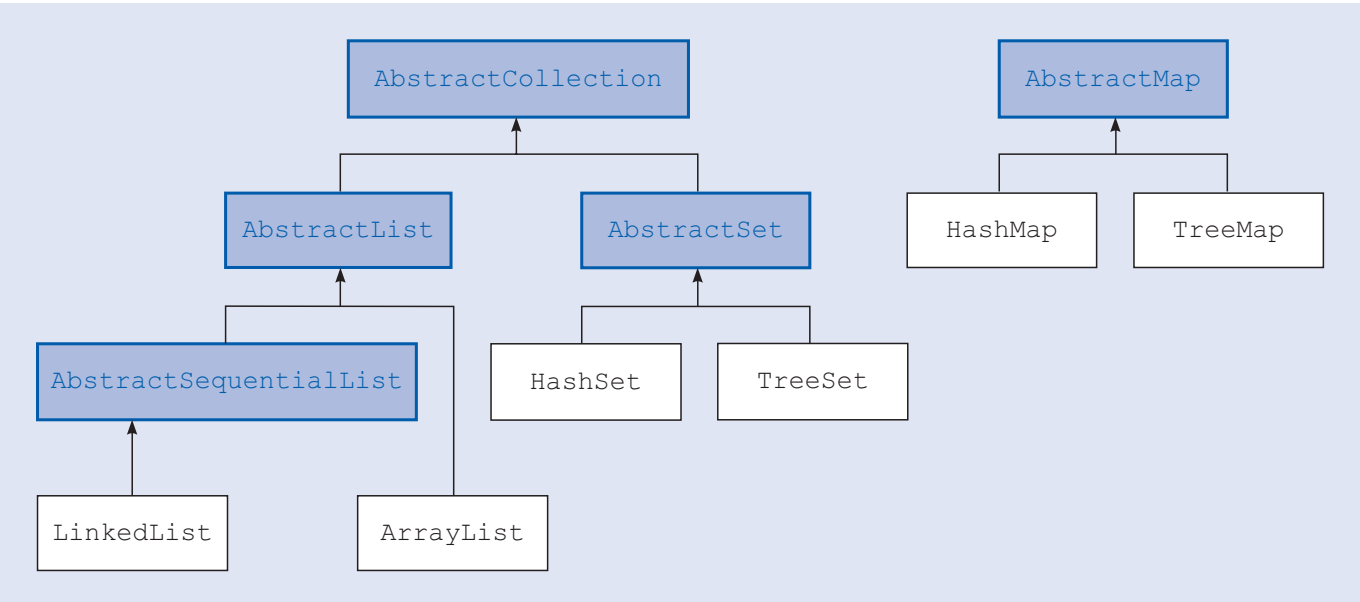


Figure 4 Some of the principal collection classes

The five shaded classes in Figure 4 are all *abstract*. The abstract collection classes will never themselves have instances, but they are an important way of factoring their subclass's classes to avoid present and future duplication.

When a class, for example `HashSet` or `HashMap`, declares that it will implement a given interface (`Set` and `Map` respectively), then that class must define (or inherit) methods with method headers to match each method header in the interface. An interface may be seen as a way of making sure that several classes all understand and implement a shared protocol. For example, `HashSet` and `TreeSet` both implement the `Set` protocol. Each implementing class may respond slightly differently to given messages, and in some cases may add additional behaviour. For example, you saw in Activity 5 how changing the object referenced by a variable declared as the interface type `Set` from an instance of `HashSet` to an instance of `TreeSet` gave us a set that was automatically sorted.

ACTIVITY 10

Open Unit10_Project_3 and double-click on the `SetExamples` class to open the editor. The version of the `SetExamples` class in this project contains the code for the method `sortedHerbIterationDemo()` we asked you to write in Activity 5. Write a modified version of this method called `firstAndLast()`. In this version, `herbSet`, the variable to reference an instance of `TreeSet`, should be declared to be of type `SortedSet`. At the end of your method you should print out the first and last entries of the sorted set by using the messages `first()` and `last()`.

Would your program have compiled if the variable `herbSet`, had remained declared as being of type `Set`? Try this. Give a reason for your answer.

DISCUSSION OF ACTIVITY 10

Here is one way to code this method.

```
public static void firstAndLast()
{
    SortedSet<String> herbSet = new TreeSet<String>();
    herbSet.add("camomile");
    herbSet.add("rosemary");
    herbSet.add("basil");
    herbSet.add("thyme");
    herbSet.add("mint");
    herbSet.add("sage");
    herbSet.add("lovage");
    herbSet.add("parsley");
    herbSet.add("chives");
    herbSet.add("marjoram");
    for (String herb : herbSet)
    {
        System.out.println(herb);
    }
    System.out.println( "First herb is " + herbSet.first());
    System.out.println( "last herb is " + herbSet.last());
}
```

Just changing the type of the `herbSet` variable from `Set` to `SortedSet`, enabled you to send the messages `first()` and `last()` to the set referenced by `herbSet`. As `herbSet` references an instance of `TreeSet` it should be clear to you that the `TreeSet` class must implement the `SortedSet` interface.

Even though the `TreeSet` class implements the `first()` and `last()` methods and `herbSet` references an instance of `TreeSet`, the method would not compile if the variable `herbSet` was declared to be of type `Set`, as `first()` and `last()` are not in the protocol specified by the `Set` interface.

The code for `firstAndLast()` *has been added to the class* `SetExamples` *in Unit10_Project_4.*

Arrays are not part of the Collections Framework

Arrays pre-date the Collections Framework and do not implement any of its interfaces, and consequently are not considered to be part of the Collections Framework.

This historical accident explains much that would otherwise be mysterious about collections. For example, this explains (in part) why the notation for declaring arrays (square brackets after the component type) is so very different to that of sets and maps (angle brackets around the element type).

The history also affects terminology. Generally, the term '**collection class**' – even without initial capitals – will indicate a class which implements one of the interfaces from the Collections Framework. Of course, an array is a collection in the normal everyday sense, but treating arrays separately from the Collections Framework makes possible general statements such as 'All collection classes implement either the `Collection` or the `Map` interface' without the need to add 'except for arrays' every time.

4 A dating agency

In this section we are going to use sets and maps to partially implement the code for a dating agency.

A dating or personal introduction agency usually asks its clients for their interests and then tries to pair each client with someone of similar interests. In our agency, to keep things easy, clients will just be asked to name their interests. (A real-life agency would also ask for interests to be put in priority order but this is a complication we shall leave out.) We shall also keep things simple by considering only two clients, which is all we need to explain the principles.

A set is a suitable collection for storing a person's interests because we just want to record each of their interests once; there should be no duplicate entries.

For our purposes, interests will be represented as strings and there will be no restriction on the number of interests that may be entered for a person. We shall input the interests via dialogue boxes, adding them to the corresponding set as we go.

Our main goal is to be able to enter the interests for two clients and then determine what interests they have in common. Here is a concrete example.

Suppose Fred has interests represented by a set with the following four elements:

```
"TV" "Films" "Eating" "Walking"
```

and Wilma has interests represented by a set with the five elements:

```
"Golf" "Swimming" "Eating" "TV" "Cats"
```

The common interests may then be represented by a set containing the two elements:

```
"Eating" "TV"
```

The set containing just the elements that two other sets have in common – the overlapping elements – is usually called the **intersection** of the sets. The intersection of the sets above is shown diagrammatically in Figure 5. (We have deliberately shuffled the elements a bit to emphasise that the order does not matter.)

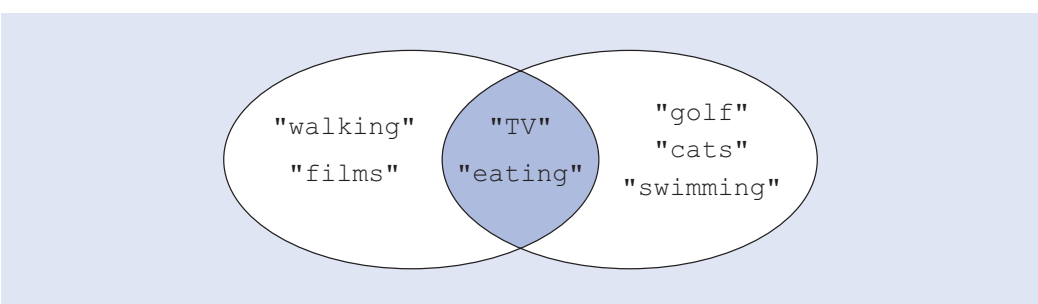


Figure 5 The intersection of two sets

Before we implement our simple dating agency we need to take a look at some bulk operations, which let us operate not on individual elements but on a set *as a whole*.

4.1 Bulk operations on sets

The interface `Set` defines several useful bulk operations, that is, operations that involve every element in one or more sets. Here are some of the most useful messages that can be sent to instances of classes that implement the `Set` interface.

- `set1.containsAll(set2);`
  Answers `true` if `set1` contains all the elements in `set2`.
- `set1.addAll(set2);`
  Alters `set1` by adding all the elements in `set2`.
- `set1.retainAll(set2);`
  Alters `set1` to contain only those elements it has in common with `set2`.
- `set1.removeAll(set2);`
  Alters `set1` to remove those elements it has in common with `set2`.

Note the last three messages are **destructive** on the set to which they are sent, that is to say, they may alter its contents. To apply them **non-destructively** (that is, without altering the contents of the original sets) we must copy the original set and work with the copy. Fortunately there is a simple way to obtain a copy. We can copy a set using the `new` operator in conjunction with a constructor with the signature `HashSet(Collection)` with the set we wish to copy as the argument. This constructor populates the newly-created set with exactly the same elements as its argument.

Given two existing sets, referenced by the variables `set1` and `set2`, the idioms shown below are the recommended way to use the bulk operations non-destructively.

The result of the code below is called the **union** of `set1` and `set2`.

```
Set<Type> union = new HashSet<Type>(set1);
union.addAll(set2);
```

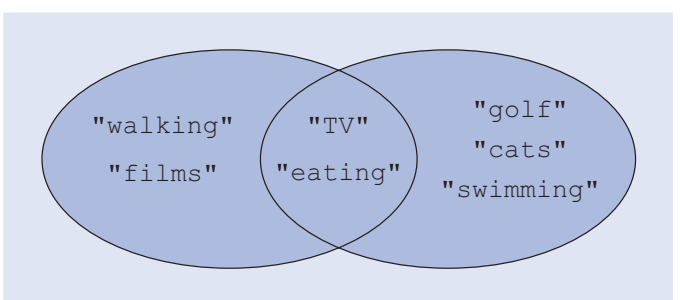
The result of the code below is called the **intersection** of `set1` and `set2`.

```
Set<Type> intersection = new HashSet<Type>(set1);
intersection.retainAll(set2);
```

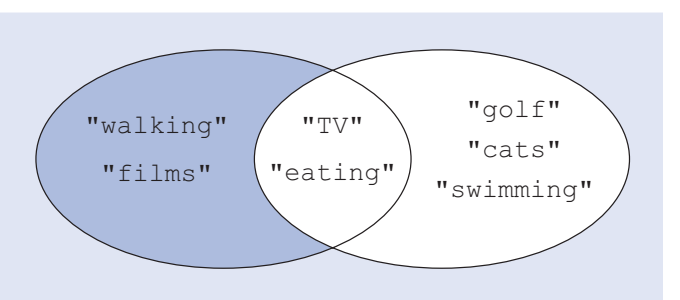
The result of the code below is called the **difference** of `set1` and `set2`.

```
Set<Type> difference = new HashSet<Type>(set1);
difference.removeAll(set2);
```

We have already seen a diagram of the intersection of two sets in Figure 5. Figures 6(a) and (b) illustrate the union and the difference of two sets respectively.



(a)



(b)

Figure 6 (a) The union of two sets (b) The difference of two sets

ACTIVITY 11

Open Unit10_Project_4 and double-click on the icon for the `SetExamples` class to open the editor. Write a new class method of `SetExamples` called `destructiveSetIntersectionDemo()`.

The method should:

1 Create two empty sets of strings referenced by the variables `set1` and `set2`.

2 Using appropriate messages, initialise `set1` so that it contains the elements `"a"`, `"b"`, `"c"`, `"d"` and initialise `set2` so that it contains `"f"`, `"c"`, `"g"`, `"a"`.

3 Alter the contents of `set1` to the intersection of the two sets (that is, a set containing only the elements the two sets have in common).

4 Print out the contents of `set1`.

Test your method in the OUWorkspace.

DISCUSSION OF ACTIVITY 11

One possible solution is as follows:

```
public static void destructiveSetIntersectionDemo()
{
    Set<String> set1 = new HashSet<String>();
    Set<String> set2 = new HashSet<String>();
    set1.add("a");
    set1.add("b");
    set1.add("c");
    set1.add("d");
    set2.add("f");
    set2.add("c");
    set2.add("g");
    set2.add("a");
    set1.retainAll(set2);
    System.out.println("The intersection is: ");
    for (String element : set1)
    {
        System.out.println(element);
    }
}
```

ACTIVITY 12

Again, using Unit10_Project_4, write a new *class* method of `SetExamples` called `safeSetUnionDemo()`. The method should:

1 Create two empty sets of strings referenced by the variables `set1` and `set2`.

2 Using appropriate messages, initialise `set1` so that it contains the elements `"a"`, `"b"`, `"c"`, `"d"` and initialise `set2` so that it contains `"f"`, `"c"`, `"g"`, `"a"`.

3 Create a set of strings `set3` which is a copy of `set1` by using the constructor with the signature `HashSet(Collection)` with `set1` as argument.

4 Alter the contents of `set3` so that it contains the union of `set3` and `set2` (that is, a set containing the elements from both `set3` and `set2`), leaving the original `set1` unaltered.

5 Print out the contents of `set3` to the Display Pane one line at a time.

Test your method in the OUWorkspace.

DISCUSSION OF ACTIVITY 12

One possible solution is as follows:

```
public static void safeSetUnionDemo()
{
    Set<String> set1 = new HashSet<String>();
    Set<String> set2 = new HashSet<String>();
    set1.add("a");
    set1.add("b");
    set1.add("c");
    set1.add("d");
    set2.add("f");
    set2.add("c");
    set2.add("g");
    set2.add("a");
    Set<String> set3 = new HashSet<String>(set1);
    set3.addAll(set2);
    System.out.println("The union is:");
    for (String element : set3)
    {
        System.out.println(element);
    }
}
```

The class methods `destructiveSetIntersectionDemo()` *and* `safeSetUnionDemo()` *have been added to the* `SetExamples` *class in Unit10_Project_5.*

In the next activity you will start to develop a new class called `DatingAgency`.

ACTIVITY 13

Open Unit10_Project_5, then click on the New Class button which can be found to the left of the BlueJ window. When prompted for a class name enter `DatingAgency` (no spaces) and then click Ok. An icon for the `DatingAgency` class will appear in the main BlueJ window. Double-click on this icon; an editor window will now open for you to start writing the code for this new class.

You will first need to write two import statements at the top of the file:

```
import java.util.*;
import ou.*;
```

The first import statement is needed for the Collections Framework and the second for the package containing the `OUDialog` class.

As you will notice, BlueJ has automatically inserted a default constructor for objects of this class. As no object of this type will ever be created, this constructor will never be called, so if you like you may delete it (as we have done in the solution).

Next, we begin to address the problem of finding the common interests of two dating agency clients by developing a class method for `DatingAgency` which allows a client's interests to be entered from the keyboard and stored in an instance of `HashSet`. The method you must write should be based on the following incomplete code.

```
/**
 * Creates a set referenced by the variable interestsSet and then
 * repeatedly displays a dialog box (until Cancel is pressed) in
 * order to gather a client's interests and add them to interestsSet.
 * Finally the method returns interestsSet
 */
public static Set<String> gatherInterests(String client)
{
    // Replace these comments with the code to create a
    // new empty instance of HashSet<String> and assign it to
    // a variable called interestsSet declared as type Set<String>

    String dialogString = "Please input an interest for client "
                                + client + ". Press Cancel to end.";
    String input = OUDialog.request(dialogString);
    while (input != null)
    {
        // Replace these comments with the code to add
        // input to the set interestsSet
        input = OUDialog.request(dialogString);
    }
    return interestsSet;
}
```

Once you have got your class to compile, test your method `gatherInterests()` in the OUWorkspace by invoking it on the `DatingAgency` class and observe the results in the Display Pane.

DISCUSSION OF ACTIVITY 13

Here is one possible solution (the comments have been omitted for brevity):

```
import java.util.*;
import ou. *;
public class DatingAgency
{
    public static Set<String> gatherInterests(String client)
    {
        Set<String> interestsSet = new HashSet<String>();
        String dialogString = "Please input an interest for client "
                                    + client + ". Press Cancel to end.";
        String input = OUDialog.request(dialogString);
        while (input != null)
        {
            interestsSet.add(input);
            input = OUDialog.request(dialogString);
        }
        return interestsSet;
    }
}
```

The code for the class `DatingAgency` *as developed so far has been added to Unit10_Project_6.*

ACTIVITY 14

Open Unit10_Project_6; our next step is to create a second class method for the class `DatingAgency` with the following header:

```
public static void run(String clientA, String clientB);
```

This method should invoke the class method `gatherInterests()` on the `DatingAgency` class twice, first for `clientA` and then for `clientB`, and store the results in local variables of type `Set<String>` called `clientAInterests` and `clientBInterests`.

Your method should then iterate through both sets, printing out the interests to the Display Pane. Before printing each set, your code should first output a line which reads `"Interests for client "`, followed by the name of the client.

Once you have got your method to compile, test it by invoking `run()` on the `DatingAgency` class in the OUWorkspace.

DISCUSSION OF ACTIVITY 14

Here is one possible solution:

```
/**
 * Prints to the Display Pane the interests of a pair of clients
 * whose names are represented by the arguments.
 */
public static void run(String clientA, String clientB)
{
    Set<String> clientAInterests = DatingAgency.gatherInterests(clientA);
    Set<String> clientBInterests = DatingAgency.gatherInterests(clientB);
    System.out.println("Interests for client " + clientA);
    for (String interest : clientAInterests)
    {
        System.out.println(interest);
    }
    System.out.println("Interests for client " + clientB);
    for (String interest : clientBInterests)
    {
        System.out.println(interest);
    }
}
```

The class method `run()` *has been added to the* `DatingAgency` *class in Unit10_Project_7.*

ACTIVITY 15

Now we are ready to find the common interests of two clients. This will be represented by the intersection of the two sets of client interests that have been input.

Open Unit10_Project_7 and modify the method `run()` so that, once it has printed out the interests of both clients, it creates a new set referenced by the variable declared as `Set<String> commonInterests`, which is a copy of the set referenced by `clientAInterests`. The set referenced by `commonInterests` should then be altered to form the intersection with the set referenced by `clientBInterests`.

Then add code to print `"The common interests are "`, followed by all the common interests.

Compile your code and test it by invoking `run()` on the `DatingAgency` class in the OUWorkspace.

DISCUSSION OF ACTIVITY 15

Here is one possible solution:

```
/**
 * Finds and prints the common interests of a pair of clients.
 */
public static void run(String clientA, String clientB)
{
    Set<String> clientAInterests = DatingAgency.gatherInterests(clientA);
    Set<String> clientBInterests = DatingAgency.gatherInterests(clientB);
    System.out.println("Interests for client " + clientA);
    for (String interest : clientAInterests)
    {
        System.out.println(interest);
    }
    System.out.println("Interests for client " + clientB);
    for (String interest : clientBInterests)
    {
        System.out.println(interest);
    }
    Set<String> commonInterests = new HashSet<String>(clientAInterests);
    commonInterests.retainAll(clientBInterests);
    System.out.println("The common interests are: ");
    for (String interest : commonInterests)
    {
        System.out.println(interest);
    }
}
```

The updated method `run()` *has been added to the* `DatingAgency` *class in Unit10_Project_8.*

4.2 A map implementation of the dating agency

In the final series of activities we will look at a map implementation of the dating agency. Rather than keeping each client's interests in a separate collection, the entire dataset will be housed in a single collection of type `Map<String, Set<String>>`. This may look a bit intimidating but don't panic! It simply means that the keys are strings and the corresponding values *sets* of strings. The strings which are the keys represent the client names and the associated sets of strings represent the interests of each client, something like the following.

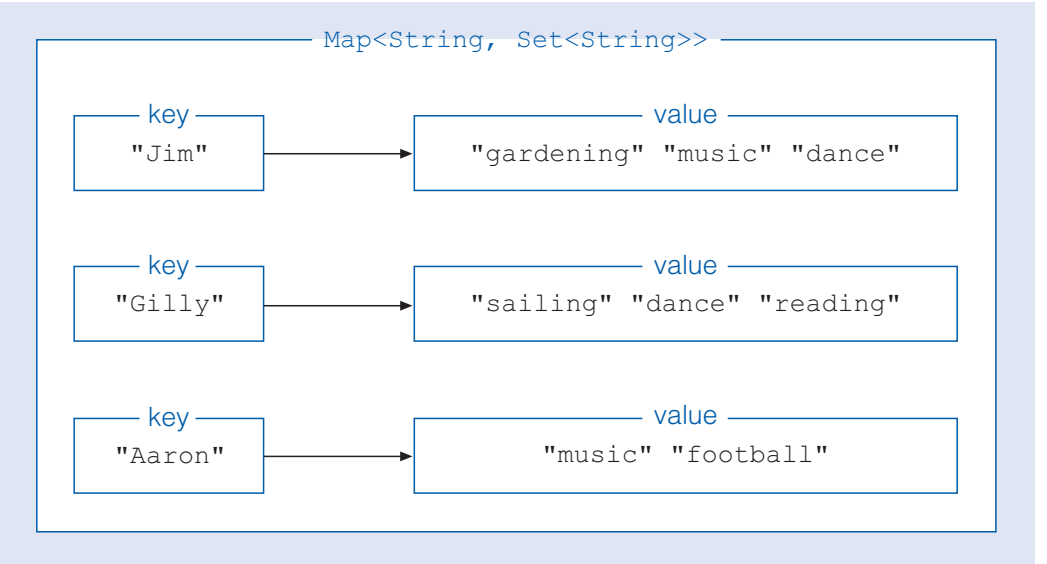


Figure 7 A map whose keys are strings and values are sets of strings

ACTIVITY 16

The method below, which can be found in the `DatingAgency2` class in Unit10_Project_8, creates a map which records the names and interests of three clients:

```
/**
 * Stores and retrieves data about client names and interests.
 */
public static void datingDemo()
{
    Map<String, Set<String>> clientMap = new HashMap<String, Set<String>>();
    Set<String> interests = new HashSet<String>();
    interests.add("food");
    interests.add("wine");
    interests.add("tandems");
    clientMap.put("Hal", interests);
    interests = new HashSet<String>();
    interests.add("food");
    interests.add("walking");
    interests.add("welding");
    clientMap.put("Gemma", interests);
    interests = new HashSet<String>();
    interests.add("food");
    interests.add("cinema");
    interests.add("wine");
    clientMap.put("Caroline", interests);
    interests = clientMap.get("Xander");
    System.out.println("Xanders interests are " + interests);
    if ((clientMap.remove("Xander")) == null)
    {
        OUDialog.alert("Xander not found");
    }
    for (String eachClient : clientMap.keySet())
    {
        interests = clientMap.get(eachClient);
        System.out.println(eachClient + " is interested in: " + interests);
    }
}
```

Note that in the final line of code we simply concatenated the `interests` set with the rest of the string we wish to output to the Display Pane. The `println()` method will automatically send a `toString()` message to `interests` to gain its textual representation.

An alternative approach, if we wanted some control on how the individual elements of the `interests` set were printed (perhaps one per line, prefixed by an arrow), would have been to iterate through the set, printing each element. To do this we would have needed nested `foreach` statements; an outer one iterating through the map and an inner one iterating through each set. For example:

```
for (String eachClient : clientMap.keySet())
{
    interests = clientMap.get(eachClient);
    System.out.println(eachClient + " is interested in: ");
    for (String eachInterest : interests)
    {
        System.out.println("—> " + eachInterest);
    }
}
```

However, as we did not need such formatting, we simply relied on `println()` to output the default textual representation for us and so avoided this complication.

Open Unit10_Project_8 then try out `datingDemo()` by invoking it on `DatingAgency2` in the OUWorkspace.

Examine the code of the method and answer the following questions.

1 Why was it necessary to repeat the following statement:

```
interests = new HashSet<String>();
```

after putting an entry into the map?

2 The `Set` interface contains a method `clear()` which removes all the elements from a set. Why can we not just use `clear()`, instead of creating a new `HashMap` instance each time?

DISCUSSION OF ACTIVITY 16

1 Because otherwise there would only ever be one set and each time we added an entry into the map it would be this same set that got used.

At the end, all the clients would be associated with exactly the same `HashSet` object, and all the interests that had been added would have been inserted into this one set. The result would be that every client would appear to have an identical set of interests, containing the pooled interests of all the clients.

2 This would have a similar effect to the one described in item 1, except that the one and only set involved would have been cleared for each client. Consequently at the end all that would be retained would be the interests of the final client, and the set containing these would be entered against the name of every single client.

ACTIVITY 17

As you have already seen, the simplest way of iterating over a map is to obtain a set of the keys and then iterate over those, accessing the corresponding values as we go.

The method `matchesDemo()` shown below, is an extension of the `datingDemo()` class method you encountered in the `DatingAgency2` class in Activity 16. It includes an additional step to check which of Hal's interests match those of the other clients.

First the set of Hal's interests are retrieved from `clientMap` and assigned to the variable `halsInterests`.

Then the key set of `clientMap` is obtained and iterated over using a `foreach` statement. Each client name in the key set is then used to access the corresponding set of interests. The bulk operation `retainAll()` is used to find the interests the client and Hal have in common and these common interests are then printed out to the Display Pane.

```
/**
 * Finds and outputs matching interests.
 */
public static void matchesDemo()
{
    Map<String, Set<String>> clientMap = new HashMap<String, Set<String>>();
    Set<String> interests = new HashSet<String>();
    interests.add("food");
    interests.add("wine");
    interests.add("tandems");
    clientMap.put("Hal", interests);

    interests = new HashSet<String>();
    interests.add("food");
    interests.add("walking");
    interests.add("welding");
    clientMap.put("Gemma", interests);

    interests = new HashSet<String>();
    interests.add("food");
    interests.add("cinema");
    interests.add("wine");
    clientMap.put("Caroline", interests);

    Set<String> halsInterests = clientMap.get("Hal");
    String outputString;
    Set<String> eachClientInterests;
    Set<String> intersection;
    for (String eachClient : clientMap.keySet())
    {
        eachClientInterests = clientMap.get(eachClient);
        intersection = new HashSet<String>(halsInterests);
        intersection.retainAll(eachClientInterests);
        outputString = "Hal and " + eachClient
                       + " have common interests: "
                       + intersection;
        System.out.println(outputString);
    }
}
```

With Unit10_Project_8 open invoke the `matchesDemo()` method on the `DatingAgency2` class in the OUWorkspace and convince yourself that it does indeed produce the common interests of Hal with each person in the map. In the body of the `foreach` loop, why is `intersection` reinitialised each time with a fresh set containing Hal's interests?

DISCUSSION OF ACTIVITY 17

The set containing Hal's interests is recreated as a fresh set with the same contents before calculating common interests, because the set message `retainAll()` used to calculate set intersection, is destructive – it changes the contents of the receiver. This would disturb subsequent calculations of common interests.

The method `matchesDemo()` reports that Hal and Hal have common interests: `"wine"`, `"food"` and `"tandems"`. This is rather silly because Hal should be compared only with everyone other than himself. The next activity rectifies this.

ACTIVITY 18

Again using Unit10_Project_8 and the `DatingAgency2` class, create a modified version of the method `matchesDemo()` called `matchesDemo2()` this method should ensure that Hal does not have his interests compared with himself.

DISCUSSION OF ACTIVITY 18

We want the Display Pane to display the common interests only when `eachClient` is not equal to `"Hal"`. There are various ways of doing this. One approach is to iterate over a map from which Hal's entry has been removed. Another is to use an `if` statement in the iteration.

If we use the first approach the relevant part of the code would look as follows:

```
Map<String, Set<String>> mapWithoutHal =
                        new HashMap<String, Set<String>>(clientMap);
mapWithoutHal.remove("Hal");
for (String eachClient : mapWithoutHal.keySet())
{
    eachClientInterests = clientMap.get(eachClient);
    intersection = new HashSet<String>(halsInterests);
    intersection.retainAll(eachClientInterests);
    outputString = "Hal and " + eachClient + " have common interests: "
                   + intersection;
    System.out.println(outputString);
}
```

The code below shows how we would implement the second idea:

```
for (String eachClient : clientMap.keySet())
{
    if (eachClient != "Hal")
    {
        eachClientInterests = clientMap.get(eachClient);
        intersection = new HashSet<String>(halsInterests);
        intersection.retainAll(eachClientInterests);
        outputString = "Hal and " + eachClient + " have common interests: "
                       + intersection;
        System.out.println(outputString);
    }
}
```

You can find for the complete code for each of these solutions in the `DatingAgency2` *class that can be found in Unit10_Project_9. The two methods are named* `matchesDemo2()` *and* `matchesDemo3()` *respectively.*

5 Summary

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

- Different kinds of collection can be distinguished according to the following properties.
- Fixed size versus dynamically resizable.
- Ordered versus unordered.
- Homogeneous versus non-homogeneous.
- Indexed versus unindexed.
- Whether duplicates are allowed or not.
- From Java 1.5 onwards all collections in Java are homogeneous, and the element type generally needs to be declared.
- Sets are dynamically sized. No element in a set can be duplicated. A set is not indexed but some kinds of set are ordered.
- It is best practice to declare variables as being of an interface type, such as `Set` or `Map`, rather than of some implementation class, such as `HashSet` or `HashMap`.
- `HashSet` and `TreeSet` are classes which are both implementations of the `Set` interface. `HashSet` implements the `Set` interface. `TreeSet` implements the `SortedSet` interface. Therefore, as `SortedSet` is a subinterface of `Set`, then `TreeSet` must also implement the `Set` interface.
- Instances of `TreeSet` are ordered; Instances of `HashSet` are not.
- Collections cannot contain primitive data values in the literal sense. However, from Java 1.5 onwards, values of primitive data types are automatically wrapped in their corresponding wrapper classes when added to a collection and unwrapped when extracted from a collection. This process is known as autoboxing. Prior to 1.5, programmers had to code the boxing and unboxing operations explicitly.
- The `foreach` loop (also known as the enhanced `for` loop) can be used to iterate over any collection.
- The protocol defined by the `Set` interface includes `size()`, `isEmpty()`, `contains()`, `add()` and `remove()`. These operations can be divided into the informal categories of testing, adding and removing.
- Maps have key–value pairs, also known as entries, as their elements. The same key is not allowed more than once in a map, although the same value may occur any number of times.
- Maps are dynamically sized, indexed by keys and do not allow duplicate keys.
- The `HashMap` class is an implementation of the `Map` interface.
- A key is something that can be used to identify uniquely an object within a collection.
- The protocol defined by the `Map` interface includes `size()`, `isEmpty()`, `containsKey()`, `containsValue()`, `put()`, `get()` and `remove()`.
- If the same key is put into a map twice with different values, the second value will overwrite the first.
- If `get()` or `remove()` are used with a key which is not present in the map, `null` is returned.
- The Collections Framework unifies the collection classes. It contains two root interfaces, `Collection` and `Map`. `Set` is a subinterface of `Collection`.

- Arrays are not part of the Collections Framework, as they do not implement either the `Collection` or `Map` interfaces.
- A simple way to iterate over a map is to extract its key set and iterate over the key set using each key to access the associated value.
- The bulk operation messages `retainAll()`, `addAll()` and `removeAll()` correspond to the set operations known as intersection, union and set difference. These messages are destructive – they alter the receiver – so it is normally best to work with a copy.
- The message `containsAll()` is non-destructive.
- A `HashSet` can be copied by using the constructor `HashSet(aCollection)` in conjunction with the operator `new`.

LEARNING OUTCOMES

After studying this unit you should be able to:

- understand why collections are essential in any programming language;
- classify collections on the basis of whether they are fixed-size or dynamically resizable, ordered or unordered, homogeneous or non-homogeneous, indexed or not, and whether duplicates are allowed;
- understand that all collections in Java are homogeneous and that when declaring variables or creating collection objects we must specify the element type of the collection between angle brackets <>;
- understand that a set is a variable-size collection which allows no duplicates;
- understand that is good practice to declare variables as being of an interface type rather than an implementation class type, because this aids maintainability;
- send messages to set objects corresponding to the abstract methods defined in the `Set` and `SortedSet` interfaces;
- use the set implementation classes `HashSet` and `TreeSet` and know that the main difference is that instances of `TreeSet` are ordered;
- understand that collections can only contain objects but that we can use collections with primitive data types because Java will automatically box a primitive data value in a wrapper object which can be then be stored in a collection, a process called autoboxing;
- iterate through sets using `foreach` loops;
- explain that a map is a variable-sized collection whose elements are entries consisting of a key–value pair;
- understand that a map cannot have duplicate keys – all keys in a map must be unique;
- create instances of the class `HashMap` which implements the `Map` interface;
- send messages to `HashMap` objects corresponding to the abstract methods defined in the `Map` interface;
- iterate through a map, by obtaining the key set for the map and iterating through the key set with a `foreach` loop, using each key to access the corresponding value in the map;
- have an overview of the Collections Framework;
- understand that arrays are rather different from other types of collection in Java. They involve a different syntax and they can hold primitive data values without using wrapper objects. Arrays do not form part of the Collections Framework because they do not implement either `Set` or `Map`;
- use bulk operation messages to find the intersection, union, and difference of sets;
- create copies of sets in order to use destructive bulk operation messages without affecting the original sets;
- work with simple examples of collections whose elements are themselves collections.

Glossary

autoboxing Collections that are part of the Collections Framework can only contain objects – not primitives types. Before Java 1.5, this meant that the programmer had to manually wrap each primitive value in an instance of its corresponding **wrapper class** before it could be added to a collection. Conversely, on removal from a collection, each primitive value had to be removed from its wrapper before use. This process was laborious, tedious and error-prone, and is still widespread in older code. However, since the advent of Java 1.5, primitive values may now in effect be added to and accessed from collection classes, since the compiler now ensures that the above process takes place automatically. This automatic process of wrapping and unwrapping primitive values when needed is called autoboxing.

bulk operation Iteration offers the programmer the opportunity to access or process each element of a collection in turn and deal with each as they think fit. By contrast, bulk operations are operations that act on an entire collection without requiring the programmer to focus on individual elements.

Collections Framework Shorthand for the **Java Collections Framework**.

collection classes/Collection Classes When capitalised, this phrase usually refers to collection classes that are part of the **Collections Framework** – by this fact *excluding* arrays. When used in lower case, the same phrase generally means any class that implements a collection (in the looser English language sense) – which *includes* arrays.

destructive operation An operation on a collection (i.e. a message-send) that changes the contents in some way is said to be destructive**.** It is not always obvious whether a message is destructive or **non-destructive** – for example, a **bulk operation** that might be expected to change the contents may leave it unaffected but return an altered copy, or vice versa.

difference Difference is a mathematical operation on two sets, that in Java is implemented by a `removeAll()` message. Given two sets, `set1` and `set2`, the message-send `set1.removeAll(set2)` would remove from `set1` all the elements contained by `set2`. The term 'difference' is used to refer to both the result and the operation.

entry An entry in a map means a **key–value pair** – not just the **key** or the value.

intersection Intersection is a mathematical operation on two or more sets that results in a set containing only all common elements. In Java, intersection is implemented by a `retainAll()` message. Given two sets, `set1` and `set2`, the message-send `set1.retainAll(set2)` would remove from `set1` any elements that were not also contained by `set2`. The term intersection is used to refer both to the result and to the operation.

Java Collections Framework The Collection Framework includes a set of collection interfaces (`Collection`, `Set`, `List`, `Map` and others), a set of collection classes (e.g. `HashSet` and `HashMap`) that implement them, and some supporting utility classes (e.g. `Collections`). The framework also includes a set of associated recommendations about constructors and about expected behaviours in common. Arrays are *not* part of the Collections Framework.

key Some collections of information are set up for quick convenient access using some pre-chosen part of the information stored – e.g. looking up telephone numbers using names. In such a situation, a key is something that can be used to uniquely identify any object from a given collection. What constitutes a good key may depend on the purpose of the collection.

key–value pair Where a collection is organised for looking up values using a single kind of **key**, the collection can be viewed as a set of key–value pairs. Each entry in a `Map` is a key–value pair. For example, a key might be a person's name, and its associated value their telephone number.

map An unordered collection of key–value pairs. Duplicate keys are not allowed as they are used to look up their associated values. A map has a dynamic size (i.e. it can grow and shrink as key–value pairs are added and removed).

non-destructive operation An operation on a collection that does not change the contents in any way is said to be non-destructive.

set An unordered collection which has no indexing, that contains no duplicates and which has a dynamic size (i.e. it can grow and shrink as elements are added and removed).

subinterface A subinterface is an interface that contains all the signatures of some parent interface (called its **superinterface**), and which typically contains additional signatures.

superinterface A superinterface is an interface viewed from the perspective of another interface (called its **subinterface**). The subinterface contains all the signatures of the superinterface and typically contains additional signatures.

union Union is a mathematical operation on two or more sets that results in a set containing all the elements of all the sets. In Java, union is implemented by an `addAll()` message. Given two sets, `set1` and `set2`, the message-send `set1.addAll(set2)` would add to `set1` all the elements contained by `set2`. The term intersection is used to refer both to the result and to the operation.

wrapper class See **autoboxing** to understand why these classes need to exist. Every primitive type has a corresponding wrapper class whose purpose is to hold a primitive value in a place where a primitive value is intuitively what is needed, but where the type checking requires an object. For example, the wrapper class of `int` is `Integer`.

Index

A
autoboxing 13

C
collection classes 6
    terminology 36

D
destructive 38

difference 38

E
element type 7

entry 23

H
`HashMap` 22

`HashSet` 7

I
interface type 9

intersection 38

J
Java Collections Framework 7

`Java.util` 9

K
key 20

key–value pair 21

M
map 20

mapping 23

N
non-destructive 38

S
set 6

`size()` 10

subinterface 33

superinterface 33

U
union 38

W
wrapper class 13