MT201

Unit 10

Data structures

Course team

Developer: Herbert Shiu, Consultant

Designer: Dr Rex G Sharman, OUHK

Coordinator: Kelvin Lee, OUHK

Member: Dr Vanessa Ng, OUHK

External Course Assessor

Professor Jimmy Lee, Chinese University of Hong Kong

Production

ETPU Publishing Team

The Open University of Hong Kong
30 Good Shepherd Street
Ho Man Tin, Kowloon
Hong Kong

Contents

Introduction

Here comes the last unit in the course. In the previous two units, you learned how to perform input/output operations and develop GUI-based software applications using the object-oriented approach. Performing input/output operations bridges the outside world with the JVM so that data can be transmitted between them. GUI-based software applications enable you to provide better user interfaces to the users, to operate the software applications and present the processed results to the users.

This unit concentrates on discussing how to maintain structured data items or objects within the JVM so that the data can be organized in a more systematic way. A collection of data items that can be manipulated via a single name is known as a *data structure*. Based on this definition, an object can be treated as a data structure because an object encapsulates its own data. You came across a basic data structure in *Unit 5* and *Unit 6* — the array. Arrays enable the data or objects of the same type to be maintained by an array object. The elements maintained can be accessed arbitrarily, by providing a subscript (or index). Using a subscript to access an element is fast. However, a native problem with arrays is that their size is determined at the time they are created. Once an array object is created, it is not possible to resize it to accommodate more elements. In this unit, we discuss some utility classes in which the objects can behave like array objects but are much more flexible (but the time to access the elements may be a bit longer than that of accessing elements of an array). These classes are known as the *collection* classes. Additionally, we discuss how to develop our own data structures if the existing collection classes do not fulfill all our requirements.

Most data structures discussed in the unit can store the references of the type `Object` and hence all objects except primitive types can be maintained. For example, a variable of the non-primitive type cannot store any primitive value such as an integer. To resolve the problem that a variable of `Object` type cannot maintain primitive values, another set of utility classes are defined that can encapsulate primitive values. These classes are known as *wrapper* classes. We discuss the reasons for the existences of wrapper classes and their usage in detail.

Since the beginning of the course, we have been using the `String` object to represent textual data. In this unit, we discuss the methods defined by the `String` class for string manipulations. In some instances, `String` objects are not preferable due to performance problems, but an alternative class `StringBuffer` can be used instead. We discuss the scenarios in which `String` objects are not preferable and how the `StringBuffer` class resolves the problem and its usage.

For most parts of the unit, utility classes with their attributes and methods are introduced. This unit does not cover all attributes and methods defined by those classes — only the commonly used ones. Therefore, for detailed descriptions of the classes, please refer to the API documentation.

Objectives

At the end of *Unit 10*, you should be able to:

1 *Describe* the usage of data structures.

2 *Develop* simple data structures.

3 *Apply* Java collections.

4 *Apply* wrapper classes.

5 *Apply* string manipulation with `String` and `StringBuffer` classes.

Data structures

In *Unit 1*, we said that computers are machines that can process data, and the data can be represented in various formats. In the Java programming language, the simplest way to represent a data item is by using primitive variables to store the values to be processed. For example, we can use two primitive variables to store two integers to be processed by the computer to obtain the sum of the two integers. However, the use of individual variables for representing data items cannot handle a large number of data items easily. For most problems, you will find that you have a collection of data items to be processed, and you need data structures to consolidate the data items.

In *Unit 5* and *Unit 6*, we discussed a basic data structure — array — that can be used to manipulate a list of elements that are of the same type. In the following sections, we discuss what data structures are and the alternatives to using arrays.

What are data structures?

In a few words, a data structure is a collection of data that can be handled or processed under a single name. Therefore, you have actually come across some data that can be considered data structures.

1 In *Unit 6*, in the section 'Bitwise operators', each bit of a single integer was used to represent enrolment in a course. In this instance, a single integer can be considered a data structure, because each single bit is a data item.

2 An object can be considered a data structure. For example, a `Time` object encapsulates the data items for hour, minute and second.

3 Array, discussed in *Unit 5* and *Unit 6*, is a typical data structure that can be used to consolidate a group of data items of the same type.

In the following sections, we discuss some common data structures and their capabilities and uses.

Common data structures

Let's investigate some common data structures used in various programs. The discussions are based on their external behaviours without being concerned with how to implement them. Later in the unit, we discuss their implementations with sample programs.

List

A list maintains data items with a predefined order among them. The order may simply be the order of the items entering the list. If you need to write down a list of items to buy from a supermarket, the order of the

items follows the sequence of the items appearing in your mind. If a collection of student names is maintained by a list, it is expected that the names would be arranged in a particular order, such as alphabetical order.

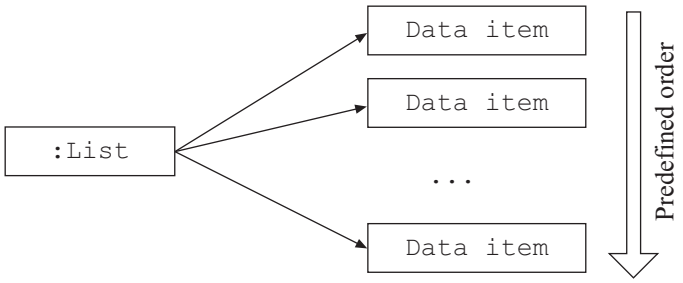


Figure 10.1 A collection of data items maintained by a list arranged in a predefined order

With a collection of data items maintained by a list, you may process the data items one by one from the first data item to the last one sequentially. When a new data item is to be added to the list, you find a suitable position among the data items for inserting it, so that the order of data items is always maintained. Similarly, if a data item is removed from a list — that is, the object is no longer maintained by the list — the remaining data items must still be kept in the predefined order by the list.

In other words, a list not only keeps track of the objects, but there is some kind of ordering among the objects. As a result, it makes sense to assign each object maintained by a list an index, so that an index can be used to access the objects maintained by the list.

If the order among the objects maintained by a list is the order in which the objects are added to the list, whenever new data are added to the list, the data item is appended to the end of the list. If every time the list is processed and the first data item is processed and then removed from the list, the list actually behaves like a queue. We discuss the behaviours and the design of a queue later in the unit.

You might feel that a list is similar to an array object. They are similar in some aspects. Actually, an array can be used to implement a list. If an array object is not yet full, a data item can be added to it. If all array elements are used, it is necessary to create a new one and copy all existing elements to the new one. If an array element is removed, other subsequent elements need to be moved forward one position.

Stack

Stack is another data structure that maintains a collection of data items. It behaves like a pile of plates in a restaurant in that the last plate you place on the pile is the first one you get from the pile. With respect to data items maintained by a stack, you can add data items to it one by one. The data item obtained from a stack is always the last one added to it. In *Unit 5* of the course, we discussed a simple stack and how to implement it in the Java programming language.

There may not be many situations in which data items have to be handled with a stack data structure in the real world. However, many algorithms in computer science need a stack data structure to handle the data items to be processed.

Tree

For some structured data that are hierarchical, the above-mentioned data structures cannot maintain the data items and the relationships among them properly. A typical example is to maintain the names of the directories and files maintained by a computer drive. Each file resides in a particular sub-directory of the drive. The sub-directory is considered the parent directory of the file. The sub-directory subsequently resides in another parent directory. To represent these files and directories, a data structure that looks like the one in Figure 10.2 can be used.

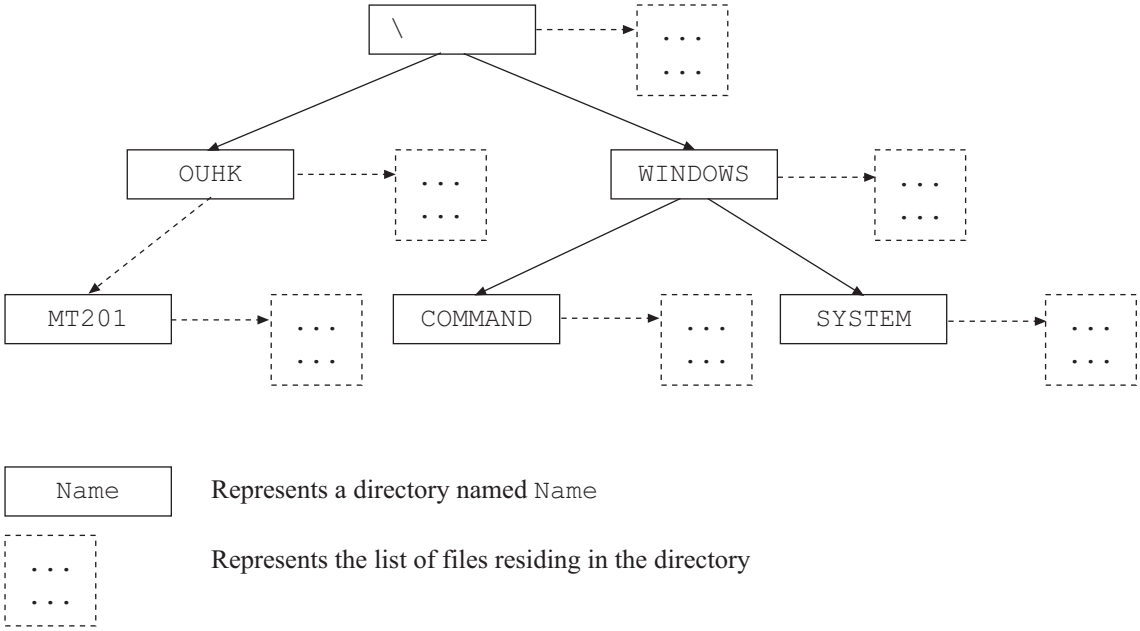


Figure 10.2 A sample structure of directories and files maintained by a computer drive

In Figure 10.2, the solid arrows are pointing from the parent directory to its sub-directories. Each directory has a list of files, which is indicated by the dashed arrow pointing from the parent directory to the list of files it contains. A data structure that looks like the one shown in Figure 10.2 is known as a tree data structure.

The characteristics of a tree data structure are:

1 The rectangles in Figure 10.2 are known as *nodes*. Each tree starts with a root node. In our example, the root node is the root directory `\`. There are parent-child relationships among the nodes, and the solid arrows point from a parent node to a child node. For example, the node named `OUHK` is considered the parent node of the child node `MT201`. If a node has at least one child node, it is considered an internal node. Otherwise, it is known as a leaf node.

2 Each node can maintain its own attributes or data. In our case, each node is a directory that maintains the list of files it has.

3 The data structure is also known as an inverted tree, because the root node is usually drawn at the top of the diagram.

Even though it is possible to use a `List` data structure to maintain the directories and files, and the hierarchical relationships among them, the approach is non-trivial. Therefore, if you can use a `Tree` data structure, it is preferable to use it to maintain data with hierarchical relationships instead of a `List` data structure.

Another typical use of a tree data structure is to maintain a collection of data items that exhibit an order among them. For example, a sequence of ordered numbers, 10, 23, 31, 45, 57, 60 and 70 can be maintained by the tree data structure shown in Figure 10.3.

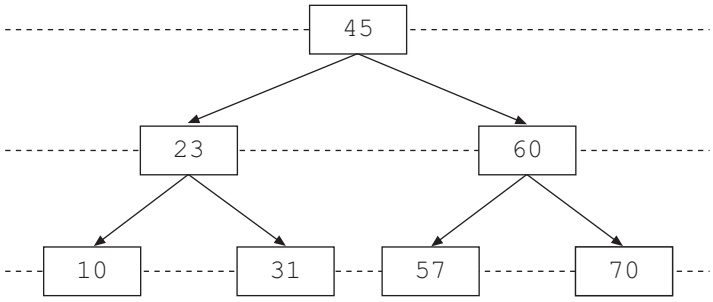


Figure 10.3 A tree data structure maintaining a collection of numeric values

Each node in the tree data structure presented in Figure 10.3 contains a number and two child nodes — a left child node and right child node. If you study the tree carefully, you will find that for each node, all numbers that are less than the node value are accessible by its left child node, and all numbers that are greater than the node value are accessible by its right child node. For example, the numbers `10`, `23` and `31` are less than `45` and the three nodes that contain these three values are accessible via the left child node of the node with the value `45`.

The height of a tree is determined by the number of levels it exhibits. The height of the tree data structure shown in Figure 10.3 is 3. (In Figure 10.3, the dotted lines are drawn to link all the nodes of the same level.)

To verify whether a number is maintained by the tree data structure, start by comparing the target number with the root node. If the target number is less than the root node, keep searching on the left child node. However, if the target number is larger than the root node, search the number on the right child node. Therefore, you can easily determine whether a given number is maintained by the tree data structure. Furthermore, during the searching, one element of each level is to be compared, and the total number of comparisons required for searching a data item in a tree data structure is equal to its height. This is the reason it is more efficient to search with a tree data structure. If the numbers are maintained by a list

data structure, it is necessary to scan through each data item sequentially until the number is found or the end of the list is reached.

For example, to determine whether the number 57 is maintained by the tree data structure shown in Figure 10.3, the searching starts with the root node with content 45. The number 57 is larger than 45, so searching continues on the right node (with content 60) of the root node. Then, searching continues at the node with content 60 and is compared with the target number 57. The number 57 is not equal to 60 and is less than 60, so searching continues on the left node (with content 57) of the current node. Finally, the node with content 57 is compared with the target number 57, and it is determined that the number 57 is maintained by the tree data structure.

Another example is to determine whether the tree data structure contains the number 15. Like the previous example, searching starts with the root node with content 45. The target 15 is less than 45, and the searching continues on the left node (with content 23) of the root node. Then, the target number 15 is compared again with the node with content 23. The target number is less than 23, and the searching continues at its left node with content 10. Afterwards, the content of the node 10 is compared with the target number 15. As the target number 15 is larger than 10, searching should continue on the right node, but there is no such node. It indicates that the target number 15 cannot be found in the above tree data structure.

The two examples — directory and file names and sequence of ordered numbers — illustrate that tree data structures not only maintain the data items but also the relationships among them.

Self-test 10.1

Study the following types of data item. Determine the suitable data structure to maintain the data items.

1 the numbers of days in the months of a year

2 a collection of student data

3 a collection of website addresses

Collections

In the previous section, we discussed the various data structures for maintaining data items. As these data structures are frequently used, the Java standard software library provides implementations in the `java.util` package that can be used for maintaining data items. As a result, programmers are freed from the responsibilities of writing those implementation classes.

As all these implementations can maintain a collection of data items, most of the data structures in the package are the subclasses of the Java interface Collection. Such an interface defines the baseline functionality a data structure or collection must have. As a reminder, Java interface was introduced in Unit 9 while we were discussing event handling. All methods defined by a Java interface are abstract, and they are to be implemented by the implementing concrete subclasses. In other words, the approaches used by the implementing concrete subclasses are up to the subclasses themselves. A collection is similar to an array and the major differences are (1) the size is not fixed, and (2) there are useful methods written for your use. These methods are important since it is convenient to use them and it can save programmers a lot of time.

In the `java.util` package are other Java interfaces that define the baseline behaviours of some common data structures, which are `List` and `Set` and `Map`. All implementing concrete subclasses of these Java interfaces can maintain a collection of objects, but the ways of handling them are different. In the following subsections, we discuss these Java interfaces. First of all, let's discuss the most fundamental one — the `Collection` interface.

Given a `Collection` object, that is, an object of the concrete subclass of the `Collection` interface, the methods frequently used that are defined by the `Collection` interface are presented in Table 10.1.

The descriptions in Table 10.1 give you a general idea of the use of the methods, but the actual behavioural details are determined by the concrete subclasses. Therefore, while you are writing programs with the concrete subclasses, you should consult the API documentation to make sure that the specific behaviour of the methods fits your requirements.

Most implementation subclasses of the Java interface `Collection` define their `toString()` methods that show all objects on the screen with their textual representation. Therefore, if you want to show the contents of a `Collection` object, you can simply use the `println()` method, such as:

```
Collection<Object> objects = ...;
System.out.println(objects);
```

The `<Object>` in the above code segment can be replaced by `<String>` (or other classes such as `<Integer>`) if you want to store strings (or objects of other classes such as integers) in the collection.

Table 10.1 Commonly used methods of a `Collection` object

Method	Use
`public boolean add( Object o)`	Ensures that the supplied `Object` object is maintained by the `Collection` object. If the data encapsulated by the `Collection` object change due to the execution of the method, a `boolean` value `true` is returned. In general, an implication is that if the `Collection` object does not contain the supplied object, the supplied object is added to the `Collection` object and a `boolean` value `true` is returned. Otherwise, nothing happens and the method returns a `boolean` `false`.
`public void clear()`	Removes all elements maintained by the `Collection` object. After executing the method, the `Collection` object no longer maintains the previously maintained objects.
`public boolean contains( Object o)`	Returns `boolean` value `true` if the `Collection` object is maintaining the supplied object. Otherwise, the method returns `false`.
`public boolean isEmpty()`	Returns `boolean` value `true` if the `Collection` object is maintaining no objects. Otherwise, the method returns `false`.
`public boolean remove( Object o)`	If the supplied object is currently maintained by the `Collection` object, the supplied object is removed from the `Collection` object and the `boolean` value `true` is returned. Otherwise, that is, the `Collection` object does not contain the supplied object, the method returns the `boolean` value `false`.
`public int size()`	Returns the number of elements maintained by this `Collection` object.
`public Object[] toArray()`	Returns the reference of an array object with element type `Object` that refers to all existing objects of the `Collection` object.

If you want to manipulate each element maintained by a `Collection` object, you can use an enhanced `for` loop. For example:

```
Collection<Object> objects = ...;
for (Object anElement: objects) {
    ... // Process each element using anElement
}
```

Each element in `objects` will be assigned to `anElement` and you can write statements in the loop body to process them.

Lists

In the `java.util` package, all implementation classes of the list data structure are the subclasses of the Java interface `List`, which defines the baseline functionality of a list data structure. All implementation classes of the `List` interface can maintain a collection of objects and the order among them, and it is therefore possible to access each object by an index. An object can be added to the `List` object more than once, and a `List` object can maintain an arbitrary number of objects. Its storage can be extended whenever necessary. Therefore, before discussing the implementation classes of the `List` interface, it is preferable to look at the methods defined by the `List` interface that are frequently used. These methods are presented in Table 10.2. As all implementation classes are the subclasses of the `List` interface, they must possess these methods; the differences are only in the ways they implement the methods.

Table 10.2 Commonly used methods of a `List` object

Method	Usage

`public void add(`
`  int index,`
`  Object element)`
	Inserts the supplied object to the specified location (by parameter `index`) in the list, and all subsequent objects in the list are shifted to right; i.e. their corresponding indexes are increased by one.

`public Object remove(` `  int index)`	Removes the element at the position specified by the `index` and all subsequent objects in the list are shifted to the left. The reference of the deleted object is returned as the return value. It is a runtime error if the supplied index exceeds the number of elements maintained by the `List` object.
`public Object get(` `  int index)`	Returns the reference of the object that is associated with the supplied `index` in the `List` object. Only the reference is returned; the internal storage of the `List` object does not change. It is a runtime error if the supplied index exceeds the number of elements maintained by the `List` object.

`public void set(`
`  int index,`
`  Object element)`
	Sets the supplied object (specified by the parameter `element`) to the location of the list specified by the parameter `index`.

`public int indexOf(` `  Object element)`	Determines and returns the index of the object specified by the parameter `element` maintained by the `List` object. If the specified object is not maintained by the `List` object, a value of `-1` is returned.
`public int lastIndexOf(` `  Object element)`	Determines and returns the last index of the object specified by the parameter `element` maintained by the `List` object. If the specified object is not maintained by the `List` object, a value of `-1` is return.

Table 10.2 lists only some of the methods defined by the `List` interface. For the complete list of method declarations, please refer to the API documentation. You can see that the methods shown in Table 10.2 are index-based. As we mentioned, a list can maintain the ordering among the elements it contains. Therefore, it makes sense to specify each element by an index, which is similar to an array.

By investigating the functionality of a `Collection` object and a `List` object, you can easily figure out that both a `Collection` object and a `List` object can maintain a collection of data. A difference is that a `List` object maintains the ordering among the data items. In other words, a `List` object is a specific type of `Collection` object. In line with this finding, the Java interface `List` is a *subinterface* of the Java interface `Collection`. As a result, all implementation classes of the `List` interface have all methods defined by the Java interface `Collection` and `List`. Therefore, if you are given a `List` object, it has all methods listed in Table 10.1 and Table 10.2. Furthermore, the interpretation of some methods, such as the `add()` and `remove()` methods, are native to a `List` object as presented in Table 10.3.

Table 10.3 Methods defined by `Collection` interface are fine-tuned by the `List` interface

Method	Usage
`public boolean add( Object element)`	The supplied object is appended to the end of the list.
`public boolean remove( Object element)`	Determines and removes the supplied object from the list. If the supplied object is found, all subsequent elements maintained by the `List` object are shifted to the left; that is, their corresponding indexes are decreased by one, and a `boolean` value of `true` is returned. If the supplied object is not maintained by the `List` object, a `boolean` value of `false` is returned.

By using the `set()` and `get()` methods to explicitly access a particular object with the index, you can treat a `List` object as an array object with element type `Object`.

An implementation: `ArrayList`

In the `java.util` package of the Java standard software library are several implementation classes of the Java interface `List` that behave as a list data structure. All of them can maintain arbitrary numbers of objects in an automatically expendable storage ‘area’, and all maintained objects can be accessed using an index. A common implementation of the `List` data structure is the `ArrayList` class (that is, the class `java.util.ArrayList`). Besides `ArrayList`, the `java.util` package provides implementation classes that are also list data structures, which are `LinkedList` and `Vector`. Appendix A of the unit provides you with comparisons of the implementation classes. For the time being,

you can use the `ArrayList` class whenever your program needs a list data structure at runtime.

The `add()` method of the `ArrayList` class implicitly appends the supplied object to the list of existing objects. Therefore, the implicit predefined ordering of an `ArrayList` object is the order of adding the objects to the `ArrayList` object. Furthermore, as we mentioned in the previous section, an `ArrayList` object, which is the implementing class of the `List` interface, can be used as an array object with element type `Object`.

To illustrate the usage of an `ArrayList` class, the class `ArrayList` is written in Figure 10.4. The class uses an `ArrayList` object to maintain `String` objects and a `Date` object. The objects are shown on the screen one by one.

```
// Resolve classes in java.util package
import java.util.*;

// Definition of class TestArrayList
public class TestArrayList {

    // Main executive method
    public static void main(String[] args) {
        // Check program parameter
        if (args.length < 1) {
            // Show usage
            System.out.println("Usage: java TestArrayList <name>");
        } else {
            // Prepare a List object (an ArrayList object)
            List<Object> objList = new ArrayList<Object>();

            // Add element to the List object
            objList.add(0, "Hello ");
            objList.add(1, args[0]);
            objList.add(2, ". It is now ");
            objList.add(3, new Date());
            objList.add(4, ".");

            // Show the objects maintained by the List one by one
            for (Object anElement : objList) {
                System.out.print(anElement);
            }
            System.out.println();
        }
    }
}
```

Figure 10.4 TestArrayList.java

Compile the class and execute the program with a program parameter, such as

```
java TestArrayList Peter
```

and the following output will be shown on the screen:

```
Hello Peter. It is now Sun Oct 12 18:50:59 CST 2003.
```

To illustrate that a `List` object is playing the role of an array object with element type `Object`, the class `TestObjectArray` is written in Figure 10.5 for comparison.

```
// Resolve classes in java.util package
import java.util.*;

// Definition of class TestObjectArray
public class TestObjectArray {

    // Main executive method
    public static void main(String[] args) {
        // Check program parameter
        if (args.length < 1) {
            // Show usage
            System.out.println("Usage: java TestObjectArray <name>");
        } else {
            // Prepare an Object
            Object[] objArray = new Object[5];

            // Add element to the object array
            objArray[0] = "Hello ";
            objArray[1] = args[0];
            objArray[2] = ". It is now ";
            objArray[3] = new Date();
            objArray[4] = ".";

            // Show the objects maintained by the array object
            // one by one
            for (int i=0; i < objArray.length; i++) {
                System.out.print(objArray[i]);
            }
            System.out.println();
        }
    }
}
```

Figure 10.5 TestObjectArray.java

Compile the `TestObjectArray` class. You can then execute it in a way that is similar to executing `TestArrayList`. You can see that the functionality of an `ArrayList` object is comparable to an array object with element type `Object`. The enhanced for loop can also be used in arrays and, after replacing the original `for` loop by this, the code becomes even more similar to that of array list. The `for` loop in Figure 10.5 can be replaced by:

```
for (Object anElement : objArray) {
   System.out.print(anElement);
}
```

In *Unit 5*, we discussed the linear and binary searching methods that were applied to array objects that maintain collections of data items (or objects) to be searched. If you read the API documentation of the Java interface `List`, the interface defines the `contains()` method that can search for the supplied object among the maintained objects and return the search result. Therefore, if you use a `List` object, such as an `ArrayList` object, to maintain the objects, you can use the `contains()` method to determine whether an object is maintained by a `List` object instead of writing the linear searching or binary searching yourself.

Sets

In mathematics, set theory provides the definition of a set and all related theories. In a few words, a set is a collection of unique elements or items, which implies that no elements or items are duplicated in a set. Also, the order of the elements are not important.

What are sets?

In some instances, if your programs have to maintain a collection of data items with no duplication and there is no specific ordering among the data items, you can use a `Set` object to maintain these data items.

Please use the following reading to acquire knowledge of sets and an implementation `BitSet`.

> ***Reading***
>
> King, section 13.6 ‘Sets’, pp. 574–78

You learned from the reading that a `Set` data structure can maintain a collection of data items. However, the implementation class `BitSet`, introduced in the reading, does not maintain a collection of objects but just primitive `boolean` values. In the following subsection, we discuss an implementation of the set data structure that can maintain a collection of objects — the `HashSet` class. Appendix D of the unit gives a case study for the class `BitSet`.

An implementation for maintaining objects: `HashSet`

Basically, a `Set` *is a* specific type of `Collection`. Do you agree with this?

A `Set` object can maintain a group of objects, which is the baseline functionality of a `Collection` object. Furthermore, a `Set` object imposes an extra restriction that it maintains no duplicated objects. Therefore, a `Set` object is a specific type of `Collection` object. It hints that the Java interface `Set` is a subinterface of the Java interface

`Collection`. Therefore, if you are given a `Set` object (an object of an implementation class of the Java interface `Set`), it exhibits all methods defined by the Java interface `Set` and `Collection`.

The Java standard software library provides `TreeSet` and `HashSet` classes that are subclasses of the Java interface `Set` and they can maintain the objects added to it with no duplications. As an example, the `HashSet` class is chosen and an illustration program, the `TestHashSet1` class, is shown in Figure 10.6. Appendix A of the unit gives you differences between the `HashSet` and `TreeSet classes`.

```
// Resolve classes in the java.util package
import java.util.*;

// Definition of class TestHashSet1
public class TestHashSet1 {

    // Main executive method
    public static void main(String[] args) {
        // Create a Set object to store strings
        Set<String> stringSet = new HashSet<String>();

        // Add objects to the set
        stringSet.add("Hello");
        stringSet.add("World");
        stringSet.add("Hello");
        stringSet.add("World");

        // Show the elements maintained by the Set object
        for (String aString : stringSet) {
            System.out.println(aString);
        }
    }
}
```

Figure 10.6 TestHashSet1.java

Compile and execute the `TestHashSet1` program. The output shown on the screen is:

```
World
Hello
```

You can see that even though four String objects were added to the Set object, it only maintains the two distinct objects. Also, the order may not be the same as the order of adding the objects to the set. If the Java standard software library provided no implementations of the Java interface Set, you would have to write your own implementation of a set data structure with a List object, and extra operations would be needed to ensure there are no duplications at any time.

Maps

We said that objects maintained by a `List` object can be accessed using an index, which is a value of primitive type `int`. What about going a step further so that the objects maintained by a data structure can be specified or accessed using another object? Such a data structure is generally known as a `Map`.

What are maps?

You can consider that the objects maintained by a `List` object are referred by a value of type `int`, as visualized in Figure 10.7.

Figure 10.7 A collection of `String` objects referred by numbers

In Figure 10.7, the arrows indicate the objects referred by the numbers. By calling the `get()` method of a `List` object with an index, you can get the reference of the corresponding object. For example, for the scenario shown in Figure 10.7, because a `List` object referred by variable `list` maintains the `String` objects, you can use the following statement:

```
String region = list.get(1);
```

The reference of the `String` object with contents `"Kowloon"` is assigned to the variable `region`. The casting is necessary because the reference obtained by the `get()` method is of type `Object` and needs casting to be assigned to a variable of type `String`.

In some situations, it is preferable to use an object to refer to another object. For example:

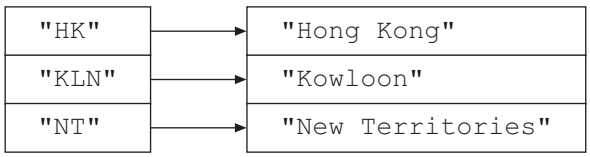


Figure 10.8 A collection of `String` objects referred by `String` objects

With such a data structure, it is expected that an object exists that maintains the `String` objects so that with the following statement

```
String region = DataStructureObject.get("HK");
```

the reference of the `String` object with contents `"Hong Kong"` is obtained and is assigned to the variable `region`. The Java standard software library provides such a data structure, generally known as a `Map`.

The relationship between the `String` objects `"HK"` and `"Hong Kong"` is called a mapping, and the `String` object `"HK"` and the `String` object `"Hong Kong"` are known as *key* object and *value* object respectively. A `Map` object maintains two sets of data items, a *key* set maintaining all key objects and a *value* set maintaining all value objects. Furthermore, it maintains the relationships between the key objects and the value objects, so that by providing a key object to the `Map` object, the `Map` object can find and return the corresponding value object. The above example uses `String` objects for illustration, but the key object and value object actually can be of any type.

Now, with a `Map` object, the following statement becomes possible:

```
Map<String, String> map = ...;
String region = map.get("HK");
```

where the first and second `"String"` are the types of the key and value respectively. In general, the declaration of a map is:

```
Map<KeyType, ValueType> map = ...;
```

where `KeyType` and `ValueType` are the types of the key and value respectively. Similar to the type `List` and `Set`, the type `Map` defines the general or baseline behaviours that a `Map` object should exhibit. Therefore, the type `Map` is defined as a Java interface in the `java.util` package. The frequently used methods defined by the Java interface `Map` are presented in Table 10.4.

Table 10.4 Frequently used methods defined by the Java interface `Map`

Method	Usage
`public Object put( Object key, Object value)`	Adds a pair of key objects and value objects to the `Map` object. Afterwards, you can use the `get()` method with the key value to obtain the value object.
`public Object get( Object key)`	Determines and returns the value object corresponding to the supplied key object. If the `Map` object maintains no such key/value mapping, a value of `null` is returned.
`public Object remove( Object key)`	Removes the mapping with the supplied key value. The method returns the value object if such a mapping exists; it returns `null` otherwise.
`public Set keySet()`	Obtains the reference of a `Set` object that maintains all key objects.
`public Collection values()`	Obtains the reference of a `Collection` object that maintains all value objects.
`public boolean containsKey(Object key)`	Returns a `boolean` value `true` if the supplied key object exists in the set that maintains all key objects, or `false` otherwise.
`public boolean containsValue(Object value)`	Returns a `boolean` value `true` if the supplied value object exists in the collection that maintains all value objects, or `false` otherwise.

The Java standard software library provides several implementation classes of the Java interface `Map`. In the following section, an implementation — the `HashMap` class — is introduced with sample uses.

An implementation: `HashMap`

An implementation class of the Java interface `Map` is the `HashMap` class provided by the `java.util` package. It implements all methods defined by the `Map` interface, including all methods presented in Table 10.4.

To illustrate the use of a `Map` object, a class `TestHashMap` is shown in Figure 10.9.

```
// Resolve classes in java.util package
import java.util.*;

// Definition of class TestHashMap
public class TestHashMap {

    public static void main(String[] args) {
        // Check program parameters
        if (args.length < 1) {
            // Show usage message
            System.out.println("Usage: java TestHashMap <region>");
        } else {
            // Create the mapping between the region codes and the
            // region full names, both of which are strings
            Map<String, String> mapping = new HashMap<String, String>();
            mapping.put("HK", "Hong Kong");
            mapping.put("KLN", "Kowloon");
            mapping.put("NT", "New Territories");

            // Show the contents of the Map objects
            Set<String> keys = mapping.keySet();
            Collection<String> values = mapping.values();
            System.out.println("Keys are: " + keys);
            System.out.println("Values are: " + values);

            // Get the region full name from the Map object
            String region = mapping.get(args[0]);
            // Show the result
            if (region == null) {
                System.out.println(
                    "The supplied region code is invalid");
            } else {
                System.out.println(
                    "The region code ("
                        + args[0]
                        + ") is referring to \""
                        + region + "\".");
            }
        }
    }
}
```

Figure 10.9 TestHashMap.java

Compile the class `TestHashMap` and execute it with a region code as a program parameter. The corresponding full name of the region code will be shown if it is a valid region code; error message is shown otherwise. For example, by executing the program with the program parameter `HK`, the following output will be shown on the screen:

```
Keys are: [HK, KLN, NT]
Values are: [Hong Kong, Kowloon, New Territories]
The region code (HK) is referring to "Hong Kong".
```

Choosing suitable collection types and implementations

Before discussing how to choose a suitable data structure of collection type, let's review the data structures we have discussed so far. The `java.util` package of the Java standard software library provides some classes for maintaining a collection of data items and, more exactly, objects. Whatever a data structure-like object that can maintain other objects actually is, it can generally be treated as a `Collection` object. Such a `Collection` type is defined as a Java interface in the `java.util` package, and it assumes no restrictions on the objects it maintains.

The `List` type is a specific type of the `Collection` type, which not only maintains the objects added to it but also the ordering among the objects. Similarly, the `Set` type is a specific type of the `Collection` type, which maintains no duplicated objects. Implementation classes, `ArrayList` and `HashSet`, were introduced for the `List` and `Set` data structures respectively. The relationship among these Java interfaces and classes is visualized in Figure 10.10.

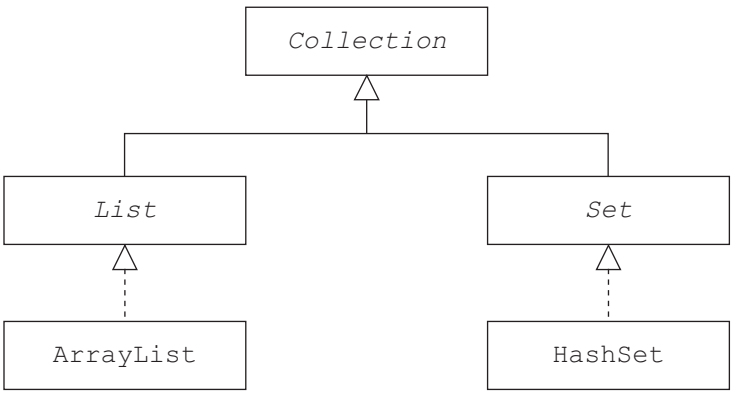


Figure 10.10 The relationship among the different Java interfaces and classes for various data structures

In Figure 10.10, the types `Collection`, `List` and `Set` are set in italics to indicate that they are Java interfaces, which are abstract types. The `ArrayList` and `HashSet` classes, in contrast, are concrete classes that implement the `List` interface and `Set` interface respectively. The solid arrows indicate the *is a* relationship, and the dashed arrows indicate the

classes `ArrayList` and `HashSet` implement the interfaces `List` and `Set` respectively. There are several implementation classes for the `List` and `Set` interfaces respectively; `ArrayList` and `HashSet` are the commonly used ones. Please refer to Appendix A of the unit to get an idea of the other implementation classes.

With respect to the relationships among the Java interfaces `Collection`, `List` and `Set`, all `List` objects and `Set` objects must have the methods defined by the `Collection` interface — hence the implementing classes `ArrayList` and `HashSet` shown in Figure 10.10. For example, the `Collection` interface defines the `contains()` method to determine whether a `Collection` object contains the supplied object. Therefore, an `ArrayList` object must possess a `contains()` method as well. As we said previously that a `List` object behaves like an array object, by making use of a `List` object and calling its `contains()` method, it is not necessary to write a searching program to find an object among the objects (as we did in *Unit 5*).

Other than `Collection`, `List` and `Set`, we discussed the data structure `Map` that maintains mappings that relate key objects with value objects.

If you determine that your program has to maintain a collection of objects for processing, you should consider the flowchart shown in Figure 10.11.

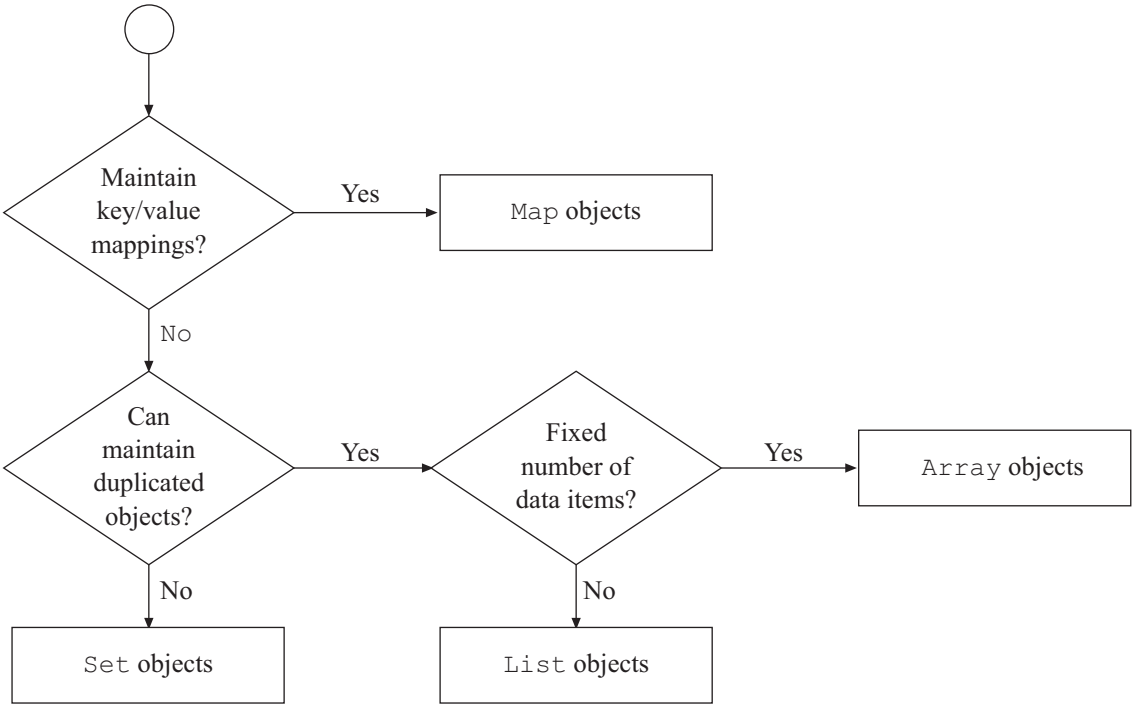


Figure 10.11 A flowchart for choosing a suitable data structure

Please use the following self-test to test your understanding of the various data structures and their uses.

Self-test 10.2

1 We said that a `List` object can be used as if it were an array. Please discuss the differences between of the two implementations.

2 Write a program that uses a `List` object to maintain the supplied program parameters. Then, sort the program parameters, which are `String` objects, and finally display the objects maintained by the `List` object.

(Hint: The `java.util` package provides a `Collections` class that defines `sort()` methods for sorting the objects maintained by a `List` object. Please refer to the API documentation for usage. Notice the `Collections` class is different from the `Collection` interface we discussed before.)

Iterators

In the previous sections, we said that whenever it is necessary to process the objects maintained by a `Collection` object — either a `List` or a `Set` object — we can use the enhanced `for` loop to process each element one by one. The framework of the `Collection` interfaces defined in the `java.util` package provides another way to iterate the maintained objects.

In Figure 10.10 of the last section, we said that `List` and `Set` are specific types of the general type `Collection`. Therefore, the Java interfaces `List` and `Set` are subinterfaces of the Java interface `Collection`, which means that an object of type `List` or `Set` must possess the methods defined by either `List` or `Set` as well as the methods defined by the `Collection` interface.

The API documentation of the `Collection` interface shows that it defines an `iterator()` method that returns an `Iterator` object that can be used to visit each maintained object one by one. As the `List` and `Set` interfaces are subinterfaces of the `Collection` interface, both `List` objects and `Set` objects define such a method that returns an `Iterator` object for visiting the maintained objects, no matter whether the data structure is a `List` or `Set`.

For example, suppose that you were given a `Collection` object. Disregarding the fact that it is a `List` or `Set` object, you can call its `iterator()` method to get an `Iterator` object, such as:

```
Collection<Object> items = ...;
Iterator iterator = items.iterator();
```

The type `Iterator` is also a Java interface defined in the `java.util` package. It defines three methods. The two commonly used ones are:

```
public boolean hasNext()
public Object next()
```

These two methods are usually used together. The following looping structure is commonly used for iterating the objects maintained by a `Collection` object:

```
Collection<Object> items = ...;
Iterator iterator = items.iterator();
while (iterator.hasNext()) {
    Object item = iterator.next();
    // Process the item
    ...
}
```

To illustrate the behaviours of an `Iterator` object, a class `TestHashSet2` is written based on the `TestHashSet1` program. It is shown in Figure 10.12.

```
// Resolve classes in the java.util package
import java.util.*;

// Definition of class TestHashSet2
public class TestHashSet2 {

    // Main executive method
    public static void main(String[] args) {
        // Create a Set object
        Set<String> stringSet = new HashSet<String>();

        // Add objects to the set
        stringSet.add("Hello");
        stringSet.add("World");
        stringSet.add("Hello");
        stringSet.add("World");

        // Obtain an Iterator object from the Set object
        Iterator iterator = stringSet.iterator();  // (1)
        // Show the elements maintained by the Set object
        while (iterator.hasNext()) {               // (2)
            System.out.println(iterator.next());   // (3)
        }
    }
}
```

Figure 10.12 TestHashSet2.java

Compile and execute the `TestHashSet2` program. At the time when the `iterator()` method of the `HashSet` object is called, the scenario is visualized in Figure 10.13.

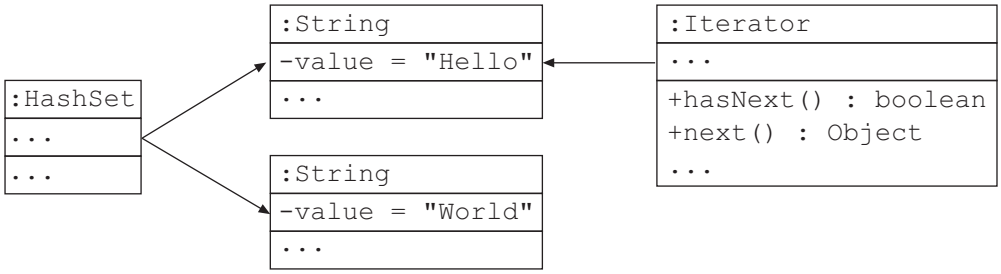


Figure 10.13 The scenario when the `iterator()` method of the `HashSet` object is called

In Figure 10.13, as the implementations of the `HashSet` object and the `Iterator` object are beyond our concern, the diagram uses ellipsis (. . .) for simplicity. The `HashSet` object maintains the two `String` objects, and it contains their references. Then, when the `iterator()` method of the `HashSet` object is called, an `Iterator` object is created and is referred by the variable `iterator`. You can consider that the `Iterator` object is referring to an arbitrary object maintained by `HashSet`. (Since a `Set` object does not maintain the order among the object it maintains, you should not assume the first object referred by the `Iterator` object must be the first object added to the Set object.) The `hasNext()` method returns a `boolean` value `true` if the `Iterator` object is referring to an object. Such an object is the one to be returned by calling its `next()` method.

Therefore, for the first iteration of the `while` loop of `TestHashSet2`, the `hasNext()` method returns `true` and the `next()` method returns the object it is currently referring to, which is the `String` object `"Hello"` in this case. Calling the `next()` method has a side effect that the `Iterator` object will refer to another object that has not been visited. In our case, it will refer to the `String` object `"World"`, as illustrated in Figure 10.14.

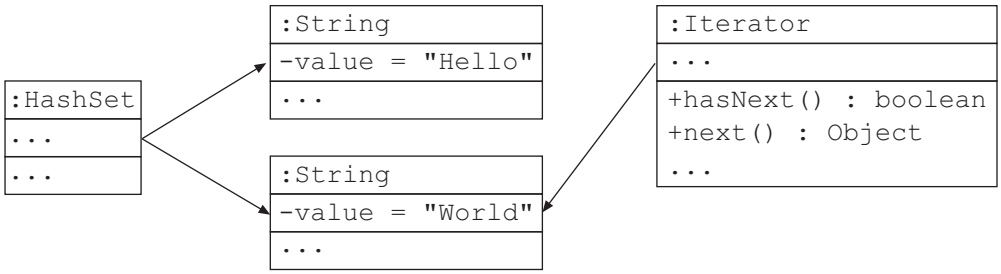


Figure 10.14 The scenario when the `next()` method of the `Iterator` object is called

For the second iteration of the `while` loop, the `hasNext()` method of the `Iterator` object is called again, and a `boolean` value `true` is returned as it is referring to an object. Then, the `next()` method of the `Iterator` object is called, and the reference to the `String` object `"World"` is returned. Similarly, the `Iterator` object tries to find and refer to an unvisited object, but there is no such object.

Finally, the `hasNext()` method is called for the third time. As the `Iterator` object cannot refer to an unvisited object, the `hasNext()` method returns a `boolean` value `false` and the `while` loop is therefore terminated.

Please use Self-test 10.3 to experiment with using `Iterator` objects.

Self-test 10.3

1 Modify the `TestArrayList` class (Figure 10.4) so that it uses an `Iterator` object to display the maintained objects one by one.

2 Modify the `TestHashMap` class (Figure 10.9) so that the program segment for showing the contents of the `Map` object is in the following format:

```
1 : HK - Hong Kong
2 : KLN - Kowloon
3 : NT - New Territories
```

Hint: You should first obtain the key set by calling the `keySet()` method of the `Map` object. Then, you can call the `iterator()` method of the key set to obtain an `Iterator` object for accessing the key objects one by one. Finally, by calling the `get()` method of the `Map` object, you can get the corresponding value objects.

Designing a data structure: queue

If you are given a problem to be solved by writing a program in the object-oriented paradigm, you should first follow the guidelines we discussed in *Unit 2* to derive the objects involved in the problem, so that you can design the candidate classes. Then, you are going to implement the behaviours of each object type, which is to write the corresponding methods of the class.

In *Unit 2*, while we were discussing the case studies, `Map` and `List` objects were mentioned for maintaining the customer objects and staff objects. Using the implementation classes provided in the `java.util` package usually suffices for most cases, but you may encounter situations in which objects of a particular type have to be maintained, or you have to manipulate the collection of objects in a particular way. Sometimes, the operations supplied by the data structures provided by the Java standard software library are too generic and need customization. Then, you might consider designing your own data structure. In this section, we are going to design a new data structure — a queue — by using the existing data structure provided by the Java standard software library.

What is a queue?

In the real world, people wait in various queues, such as for an available teller in a bank or to get on a bus. In general, people wait in queues to be served. A common pattern is that the earlier a person joins a queue, the earlier he or she is served.

In computer science, queues are required in plenty of situations. For example, when you are using a word processor application, the sequence of keystrokes you type at your keyboard is the sequence of characters added to the document being edited. Another example that we learned is the sequence of bytes your program outputs to an output stream that is associated with a file — the bytes are stored in the file in the same sequence. You can consider that when your program sends a byte to an output stream object, the byte is maintained in a queue-like structure and is to be written to the file later.

Common sense tells us that a queue is a place where people wait to be served. People join the queue at one end and leave the queue at the other end when they are to be served. All people waiting in the queue can be considered as maintained by the queue.

From the perspective of programming, a queue is a storage or data structure that maintains a collection of data or objects to be processed. The order for processing them is the same as the order in which they were added to the queue.

Determining attributes and behaviours

At runtime, a queue is an object similar to a `List` object or `Set` object that we discussed earlier. It has its own set of attributes and behaviours. From the above discussion, we know that a queue object possesses at least two behaviours — adding an element to the queue and removing an element from it. Therefore, a `Queue` object must have two methods, `enqueue()` and `dequeue()`, for adding an object to the `Queue` object and obtaining/removing an object from it. Their declarations are:

```
public void enqueue(Object obj);
public Object dequeue();
```

At any time, a `Queue` object maintains a collection of elements; hence it has an attribute `length` that specifies the number of elements maintained. As the number of elements maintained in the queue determines the attribute `length`, it does not make sense to set this attribute arbitrarily, so the attribute is read-only. A `Queue` object, therefore, should define a `getLength()` method for inquiring the current value of the attribute:

```
public int getLength();
```

From the previous section, we know that defining the types `List`, `Set` and `Map` as Java interfaces provides programmers with the flexibility of choosing different implementations. Similarly, the type `Queue` is defined as a Java interface as well, as shown in Figure 10.15.

```
// Definition of interface Queue
public interface Queue {
    // Add an object to the queue
    public void enqueue(Object obj);
    // Obtain and remove an object from the queue
    public Object dequeue();
    // Inquire the number of objects it maintains
    public int getLength();
}
```

Figure 10.15 Queue.java

Java interface concerns the behaviours (or methods) of a type only. It is up to the programmer to determine how to implement these methods.

Developing a queue class

To develop an implementing class of the `Queue` interface, you have to consider how to implement the various methods defined by the interface. A thought may flash in your mind that the class to be built *is a* queue and *has a* list of objects. Therefore, the blueprint of the class, say `LinkedQueue`, is:

```
public class LinkedQueue implements Queue {
    private List<Object> list;
```

```
    public void enqueue(Object obj) {
        ...
    }

    public Object dequeue() {
        ...
    }

    public int getLength() {
        ...
    }
}
```

The *has a* relationship gives you an insight into finding an implementation approach. There are actually many different ways of implementing a queue data structure. When you gain more experience in programming, you will figure out there are many implementation approaches. It's up to you to determine a suitable one.

All methods defined by the `Queue` interface must be implemented by the implementing class — in this case the `LinkedQueue` class. With the possessed `List` object, the three methods can be implemented easily. The last question is which implementation of a `List` should be used. For the `LinkedQueue` class, the `LinkedList` implementation is chosen. The `LinkedList` is another implementation class of the `List` interface. For the details of the `LinkedList` class, please refer to Appendix A of the unit. The complete class definition of the `LinkedQueue` class is shown in Figure 10.16.

```
// Resolve classes in java.util package
import java.util.*;

// Definition of class LinkedQueue
public class LinkedQueue implements Queue {
    // Attribute
    private List<Object> list = new
LinkedList<Object>();

    // Add an object to the queue
    public void enqueue(Object obj) {
        list.add(obj);
    }

    // Obtain and remove an object from the queue
    public Object dequeue() {
        return list.remove(0);
    }

    // Get the length of the queue
    public int getLength() {
        return list.size();
    }
}
```

Figure 10.16 LinkedQueue.java

To test the `LinkedQueue` class, a driver program `TestLinkedQueue` class is written in Figure 10.17.

```
// Definition of class TestLinkedQueue
public class TestLinkedQueue {

    // Main executive method
    public static void main(String[] args) {
        Queue queue = new LinkedQueue();

        int count = args.length;
        // Adding the objects to the Queue object
        for (int i = 0; i < count; i++) {
            queue.enqueue(args[i]);
        }
        // Show the number of elements maintained by the Queue object
        System.out.println("Number of elements = " + queue.getLength());

        // Obtain and remove the elements from the Queue object
        // one by one and show them on the screen
        for (int i = 0; i < count; i++) {
            Object obj = queue.dequeue();
            System.out.println(obj);
        }

        // Show the number of remaining elements maintained by the
        // Queue object
        System.out.println("Number of elements = " + queue.getLength());
    }
}
```

Figure 10.17 TestLinkedQueue.java

Compile the classes and execute the `TestLinkedQueue` program with program parameters. The program parameters are added to the `Queue` object one by one. Afterwards, the objects maintained by the `Queue` are obtained from it and shown on the screen. For example, enter the following command in the Command Prompt:

```
java TestLinkedQueue Hello World from MT201
```

The following output is shown on the screen:

```
Number of elements = 4
Hello
World
from
MT201
Number of elements = 0
```

The `LinkedQueue` delegates the operations of handling the objects to the possessed `List` object. The implementation can therefore be greatly simplified, and the reliability of the `LinkedQueue` class is guaranteed.

Notice that `LinkedQueue` can store primitive types such as integers. This is possible from JDK 1.5.

In Appendix B of the unit, a class `IntegerQueue` is dedicated to handle a queue of `int` values to avoid the overheads of the above method, and its performance is optimized. You can then compare the pros and cons of the two different implementations.

Self-test 10.4

1 In this section, the design of the `Queue` data structure is designed and implemented based on the finding that a queue *has a* list. Then, how about the statement that 'a queue *is a* list'? Please discuss the statement and design/implement the `Queue` class if the statement makes sense.

2 The implementation class `LinkedQueue` uses a `LinkedList` object for maintaining the objects. By modifying the `LinkedQueue` class or otherwise, develop another implementation class of the `Queue` interface, say `ArrayQueue`, which uses an `ArrayList` object for maintaining the objects. Experiment with the `LinkedQueue` and `ArrayQueue` by adding a large number of objects to them and remove them afterwards. Is there any difference in their performances? Appendix A of the unit may give you more insight.

3 Based on the discussion of the stack data structure presented earlier in the unit, develop a Java interface `Stack` with implementation classes. The following is a list of methods that a stack data structure possesses:

```
public void push(Object obj)
public Object pop()
public int getLength()
```

To maintain the objects added to the `Stack` object, similar to the Queue, you can choose either `LinkedList` or `ArrayList`. For example, develop the `LinkedStack` and `ArrayStack` classes by using the `LinkedList` and `ArrayList` respectively. Afterwards, write a driver program to test the classes and their efficiencies.

String manipulations

The class `java.lang.String` (or simply class `String`) is important in the Java standard software library. You have used it since the first Java program that we encountered in the course. A `String` object encapsulates a sequence of characters of type `char` for a particular textual representation. From time to time, whenever it is necessary to prepare a message to be shown on the screen, you have to prepare a `String` object with the desired contents, especially with the use of `String` concatenations.

Besides `String` concatenation, the `String` class defines many utility methods for `String` manipulations. Please use the following reading to learn some of these utility methods.

Reading

King, section 3.9, 'Java's string class', pp. 108–17

The reading reminds you that a sequence of characters in a program that are enclosed with a pair of double quotation marks is considered a `String` literal, and it is automatically converted to a `String` object by the compiler. Therefore, the following two expressions, `"A"` and `'A'` are significantly different, even though both denote a single character `A`. The former one enclosed with double quotation marks will be converted to a `String` object with the content of a single character `A` only, whereas the latter one that uses a pair of single quotation marks denotes a single primitive value `A` (or the equivalent numeric value `65`) of type `char`.

Therefore, a pair of double quotation marks and a pair of single quotations declare a `String` object and a primitive value of type `char` respectively. Furthermore, it is possible to have a pair of double quotation marks that encloses nothing for a `String` object with empty content; that is, `""`. However, a pair of single quotation marks that encloses nothing will cause a compile time error.

From the reading, you learned that the `String` class defines many utility methods. Basically, there are three types of utility method: informative, string manipulation and data conversions, as shown in Table 10.5.

Table 10.5 Frequently used methods defined by the `String` class

Category	Method declaration

Informative	`public char charAt(int index)`
	`public int compareTo(String anotherString)`
	`public int compareToIgnoreCase(String str)`
	`public boolean equals(Object anObject)`
	`public boolean equalsIgnoreCase(String str)`
	`public int indexOf(String str)`
	`public int indexOf(String str, int fromIndex)`
	`public int lastIndexOf(String str)`
	`public int lastIndexOf(String str, int fromIndex)`
	`public int length()`
	`public boolean startsWith(String prefix)`
	`public boolean endsWith(String suffix)`

String manipulation	`public String replace(char oldChar, char newChar)`
	`public String substring(int beginIndex)`
	`public String substring(int beginIndex, int endIndex)`
	`public String toLowerCase()`
	`public String toUpperCase()`
	`public String trim()`

Data conversion	`public char[] toCharArray()`
	`public static String valueOf(boolean b)`
	`public static String valueOf(char c)`
	`public static String valueOf(char[] data)`
	`public static String valueOf(double d)`
	`public static String valueOf(float f)`
	`public static String valueOf(int i)`
	`public static String valueOf(long l)`
	`public static String valueOf(Object obj)`

From Table 10.5, you should notice that the methods that are commonly used include class and non-class (that is, `static` and non-`static`) methods. For non-class methods (or instance methods), a `String` object performs the method and the results are determined based on the object itself. Class methods are mainly conversion utility methods that convert primitive values to equivalent `String` objects.

Most of the informative and string manipulation methods are mentioned in the reading. The extra methods are discussed in the following sections. Before the discussion, it is necessary to highlight the immutable feature of the `String` class.

Immutable `String` object

Similar to wrapper classes, a `String` class defines no methods for updating the sequence of characters that a `String` object encapsulates. In Table 10.5, you can see that the methods for `String` manipulation

provide a `String` object to be returned. For example, consider the following two statements:

```
String str1 = "abcdef";
String str2 = str1.toUpperCase();
```

The first statement assigns the reference of the `String` object with contents `"abcdef"` to the variable `str1`. Then, the second statement calls the `toUpperCase()` method of the `String` object referred by the variable `str1`. Instead of updating the contents of the `String` object referred by variable `str1`, a new `String` object is created and returned and is assigned to the variable `str2`. The scenario is visualized in Figure 10.18.

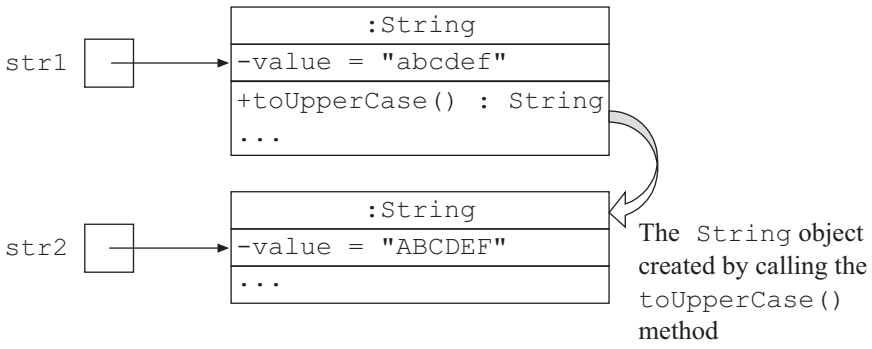


Figure 10.18 The scenario after the `toUpperCase()` method of a `String` object is called

You should notice that if the above program segment is modified to be,

```
String str1 = "abcdef";
str1.toUpperCase();
```

a new `String` object is created by the second statement and is returned. The original `String` object referred by the variable `str1` is kept unchanged. As the reference of the newly created `String` object is not maintained by any reference variable, the newly created `String` object cannot be located and used. Then, the JVM determines such a `String` object that is not referred by any reference variable, which means that such an object cannot be accessed; the JVM therefore automatically removes it from the memory.

To make the variable `str1` refer to a `String` object with all characters capitalized, the following program segment should be used instead:

```
String str1 = "abcdef";
str1 = str1.toUpperCase();
```

After executing the above program segment, the `String` object referred by the variable `str1` is `"ABCDEFG"`. Please be cautioned that it is an illusion that the contents of the original `String` object are changed. The reality is that a new `String` object with contents `"ABCDEFG"` is created and assigned to the variable `str1`. The original `String` object with contents `"abcdef"` is no longer referred by the variable `str1`. If such a

`String` object is referred by no reference variables, it will be removed automatically by the JVM. The scenario is visualized in Figure 10.19.

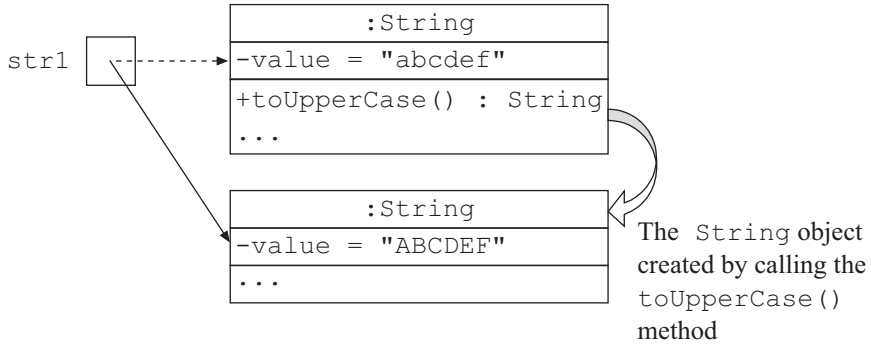


Figure 10.19 The scenario of replacing the reference of a variable after executing the `toUpperCase()` method

(In Figure 10.19, the dashed line and solid line denote the objects referred by the variable `str1` before and after executing the statement, `str1 = str1.toUpperCase();` respectively.)

In conclusion, it is not possible to change the contents of a `String` object referred by a variable. However, it is possible to replace the reference maintained by a variable so that it refers to another `String` object. It is also applicable to all non-class methods of the `String` class that return a `String` object.

String manipulating with `String` class

Most informative and string manipulation methods are discussed in the last reading. In this section, we discuss those methods that are not covered in the reading.

Methods `compareTo()` and `compareToIgnoreCase()`

In *Unit 6*, while we were discussing sorting `String` objects, the method `compareTo()` was used to compare two `String` objects to determine the lexicographical order of the two objects. Table 10.6 summarizes the behaviours of this method.

Table 10.6 The interpretation of the `compareTo()` method

The result of `str1.compareTo(str2)`	**Interpretation**
Negative	The `String` object referred by variable `str1` precedes (or is less than) the one referred by variable `str2` lexicographically.
Zero	The two `String` objects are equivalent.
Positive	The `String` object referred by variable `str1` follows (or is greater than) the one referred by variable `str2` lexicographically.

The `compareToIgnoreCase()` behaves similarly to the `compareTo()` method without considering the `String` objects. For example, if the `String` objects referred by the variables `str1` and `str2` are `"Hello"` and `"HELLO"` respectively, the result of `str1.compareTo(str2)` is positive. The result of `str1.compareToIgnoreCase(str2)` is zero.

Methods `equals()` and `equalsIgnoreCase()`

The `equals()` method is defined in the ultimate superclass `Object` class, and all classes including the `String` class must possess such a method. The `equals()` method of the `String` class returns `true` if the supplied object reference is referring to a `String` object and the sequence of the `String` object is exactly the same as that of the `String` object whose `equals()` method is being called.

The `equals()` method and its difference with the `==` operator are discussed in *Unit 7*. `String` manipulation is a little more complicated. First of all, let's remind ourselves of the core difference between the `equals()` method and `==` operator. The `equals()` method compares the two objects (the object whose `equals()` method is being called and the object supplied via the supplementary data) by executing the statements implemented by the object. Therefore, the `equals()` method usually determines the equality by comparing the attributes of the two involved objects. The `==` operator only compares the contents of the two reference variables and returns `true` if they are referring to the same object. With such information in mind, please study the program `TestStringEquals` shown in Figure 10.20 and determine its execution output.

```
// Definition of class TestStringEquals
public class TestStringEquals {

    // Main executive method
    public static void main(String args[]) {
        // Declare and initialize two String variables
        String str1 = "Hello";
        String str2 = "Hello";

        // Show the results of different comparsions
        System.out.println(str1.equals(str2));
        System.out.println(str2.equals(str1));
        System.out.println(str1 == str2);
    }
}
```

Figure 10.20 TestStringEquals.java

You should find no difficulty figuring out that the results of the first two comparisons are `true`. How about the third comparison? Compile and execute the `TestStringEquals` program. You will find that the output of the program is:

```
true
true
true
```

The result of the third comparison is `true`, which implies that the variable `str1` and `str2` are referring to the same `String` object. Such a phenomenon is due to the fact that when the compiler compiles the class definition, two identical `String` literals exist. As `String` objects are immutable, it is unnecessary and a waste of resources to have two `String` objects with the same contents. Therefore, one `String` object suffices; the compiler implicitly creates a single `String` object and its reference is assigned to the two variables `str1` and `str2` at runtime. Therefore, both `str1` and `str2` are assigned the same `String` object reference, and they practically refer to the same `String` object. This is the reason why the result of the third comparison is `true`.

You can see that you can count on the `equals()` method for `String` comparisons for most cases, unless you intentionally want to test whether two variables are referring to the same `String` object.

As the `equals()` method returns `true` if two `String` objects have the same sequence of characters and the `==` operator returns `true` if two variables are referring to the same `String` object, you can treat both the `equals()` and the `==` operator as case-sensitive comparisons. If you want to treat two `String` objects as equal if they are equivalent without the case consideration, you can use the `equalsIgnoreCase()` method.

Methods `startsWith()` and `endsWith()`

Both the methods `startsWith()` and `endsWith()` accept supplementary data of a `String` object for determining whether the supplied `String` object is the prefix or suffix of the `String` respectively. For example, the following statements,

```
str1.startsWith(str2)
str1.endsWith(str2)
```

in which `str1` and `str2` are referring to `String` objects, determine whether the `String` object referred by `str2` is the prefix or suffix of the `String` object referred by the variable `str1` respectively. You should notice that if the supplied supplementary data, the content of the variable `str2` in our case, are `null`, the methods will cause `NullPointerException` exception.

Method `replace()`

The method `replace()` accepts two supplementary data of type `char`, which denote the character to be replaced and the character for replacement; for example the following program segment:

```
String str1 = "has";
String str2 = str1.replace('h', 'w');
```

After executing the above program segment, the content of the `String` object referred by the variable `str2` is `"was"`.

Method `toCharArray()`

You occasionally need to process the sequence of characters encapsulated by a `String` object one by one. A typical way to do so is to use a looping structure and the `charAt()` method of the `String` object to obtain characters one at a time, such as the following program segment:

```
String str = ...;  // declare and initialize a String variable
for (int i=0; i < str.length(); i++) {
    char c = str.charAt(i);
    ... // process a character one at a time
}
```

Another possible way to do this is to use the `toCharArray()` to obtain an array object that maintains the sequence of the characters of the `String` object. An equivalent program segment of the above is:

```
String str = ...;  // declare and initialize a String variable
char[] charArray = str.toCharArray();
for (int i=0; i < charArray.length; i++) {
    char c = charArray[i];
    ... // process a character one at a time
}
```

If you are going to traverse the sequence of the characters maintained by a `String` object, it is preferable to use the former approach that uses the `charAt()` method to access characters one at a time. The reason is that the latter approach that uses the `toCharArray()` object will create a new array object and it could be a performance problem, especially if the `String` object contains a long sequence of characters. It can be beneficial to obtain an array object with an element of type `char` to maintain the sequence of characters of the `String` object, if it is necessary to traverse the individual characters repeatedly. Then, you can use the `toCharArray()` method to obtain such an array object.

Methods `valueOf()`

The `String` class defines several overloaded class methods `valueOf()` that accept various types of supplementary data and return equivalent `String` objects. For example, in the following statement, a `String` object `"10"` is based on a primitive value `10` of type `int`:

```
String intStr = String.valueOf(10);
```

The `valueOf()` methods of the `String` class is another approach for converting primitive values to the corresponding `String` objects, other than using the wrapper classes.

Case study: verification of Hong Kong identity card numbers

The general format of a Hong Kong identify card number is `A999999(9)`. The letter `A` denotes a letter and the digit `9` can be any arbitrary digit, `0` to `9`. The last digit enclosed in parentheses is the *check digit* derived from the leading letter and the six digits. To verify whether a given Hong Kong identity card number is valid or not, we first have to determine the algorithm for determining the check digit.

The leading letter is treated as a numeric value in the way that the letters `A`, `B` are `1, 2` and so on. The final letter `Z` is `26`.

For the seven numeric values (the leading letter and the following six digits), multiply them by `8`, `7`, `6`, `5`, `4`, `3` and `2` respectively and then sum all the products.

Obtain the remainder by dividing the above sum by `11`.

If the remainder is `0`, it is the check digit. Otherwise, the check digit is the result of subtracting the remainder from `11`. If the result is `1`, the check digit is `A`.

For example, for the Hong Kong identity card number `A123456`, the procedure for getting the check digit is:

```
   A     1     2     3     4     5     6
   1     1     2     3     4     5     6
 × 8   × 7   × 6   × 5   × 4   × 3   × 2
   8   + 7 + 12 + 15 + 16 + 15 + 12 = 85
                                   % 11
                                      8
```

As the remainder is `8`, the check digit is obtained by `11-8`, which is `3`.

The above algorithm for obtaining the check digit is implemented in the `HKIdHandler` class shown in Figure 10.21. It defines two class methods — `getIdCheckDigit()` and `isValidIdNumber()`. By supplying a String object, the method derives a check digit and verifies whether the supplied number is valid or not respectively.

```
// Definition of class HKIdHandler
public class HKIdHandler {

    // Get a check digit from a supplied HK Id number
    public static char getIdCheckDigit(String idNumber) {
        int multipler = 8;
        int sum = 0;

        // Get the sum of all multiplication of the digits and the
        // multipler
        for (int i = 0; i < idNumber.length(); i++) {
            char digit = idNumber.charAt(i);
            int value = 0;
            if (i == 0) {
                // The first alphabet
                value = digit - 'A' + 1;
            }
            else {
                // The following digits
                value = digit - '0';
            }

            sum += value * multipler;
            // Decrease the multipler
            multipler--;
        }

        // Get the remainder
        int remainder = sum % 11;

        // Get the check digit according to the remainder
        char checkDigit = '0';
        switch (remainder) {
            case 0 :
                checkDigit = '0';
                break;
            case 1 :
                checkDigit = 'A';
                break;
            default :
                checkDigit = (char) (11 - remainder + '0');
        }
        return checkDigit;
    }

    // Verify whether a supplied String object is a valid HK Id number
    public static boolean isValidIdNumber(String idNumber) {
        // Get the first alphabet and the following six digits
        String idNumberForChecking = idNumber.substring(0, 7);
        // Get the check digit
        char checkDigit = getIdCheckDigit(idNumberForChecking);
        // Return whether the check digit is consistent
        return checkDigit == idNumber.charAt(8);
    }
}
```

Figure 10.21 HKIdHandler.java

The driver program, `TestHKIdHandler` class, is shown in Figure 10.22.

```
// Definition of class TestHKIdHandler
public class TestHKIdHandler {

    // Main executive method
    public static void main(String args[]) {
        // Check whether a program parameter is provided
        if (args.length < 1) {
            System.out.println("Usage: TestHKIdHandler <ID Number>");
            System.out.println(
                "<ID Number> = A999999, for getting the check digit");
            System.out.println(
                "<ID Number> = A999999(9), for verification");
        } else {
            // Get the Id number from the program parameter
            String idNumber = args[0].toUpperCase();
            // Process the Id number according to its length
            switch (idNumber.length()) {
                case 7 :
                    char checkDigit =
                        HKIdHandler.getIdCheckDigit(idNumber);
                    System.out.println(
                        "The check digit is (" + checkDigit + ").");
                    break;
                case 10 :
                    if (HKIdHandler.isValidIdNumber(idNumber)) {
                        System.out.println(
                            args[0] + " is a valid HK Id Number.");
                    } else {
                        System.out.println(
                            args[0] + " is not a valid HK Id Number.");
                    }
                    break;
                default :
                    System.out.println("Invalid ID Number format");
                    break;
            }
        }
    }
}
```

Figure 10.22 TestHKIdHandler.java

Compile the classes and execute the `TestHKIdHandler` program with a Hong Kong identity card number as program parameter. The program derives the check digit if the supplied identity number is in the format `A999999`. Otherwise, if the format of the supplied identity number is in the format `A999999(9)`, the program will verify the supplied number. The following are several sample executions:

```
java TestHKIdHandler A123456
The check digit is (3).

java TestHKIdHandler A123456(3)
A123456(3) is a valid HK Id Number.

java TestHKIdHandler A123457(3)
A123457(3) is not a valid HK Id Number.
```

Please use the following self-test to verify your understanding of the uses of the methods mentioned above.

Self-test 10.6

1 Determine results obtained by the following expressions,

a `str.trim().replace(' ', 'A')`

b `str.replace(' ', 'A').trim()`

c `str.trim().startsWith("Hello")`

d `str.lastIndexOf(" ") - str.indexOf(" ")`

where the variable `str` is referring to a `String` object with contents `" Hello World "`.

2 Study the following program:

```
// Definition of class TestStringEquals
public class TestStringEquals {

    // Main executive method
    public static void main(String args[]) {
        // Declare and initialize two String variables
        String str1 = new String("Hello");
        String str2 = new String("Hello");

        // Show the results of different comparsions
        System.out.println(str1.equals(str2));
        System.out.println(str2.equals(str1));
        System.out.println(str1 == str2);
    }
}
```

Determine and discuss the output of the program and figure out the number of `String` objects involved throughout the program execution.

Using `StringBuffer` class for mutable string objects

Both the `String` and `StringBuffer` classes provided by the Java standard software library enable you to encapsulate a sequence of characters. The core difference between them is that `StringBuffer` objects provide the flexibility for changing the encapsulated sequence of characters, whereas the contents of a `String` object cannot be changed.

In previous units, we have been using `String` concatenation operations for constructing various outputs, such as the following statement:

```
System.out.println("The value of var is " + var);
```

The supplementary data supplied to the `println()` method are actually the reference of a `String` object obtained by `String` concatenation. Such a statement works properly, and we are happy with it for most cases.

However, if your program uses `String` concatenation intensively, it can be a performance problem. Let's look at the class `TestString` as shown in Figure 10.23.

```
// Definition of class TestString
public class TestString {

    // Main executive method
    public static void main(String[] args) {
        // Prepare an empty String object
        String str = "";
        // Obtain a String with 30000 characters
        for (int i=0; i < 30000; i++) {
            str += "*";
        }
        // Show the length of the String for verification
        System.out.println(str.length());
    }
}
```

Figure 10.23 TestString.java

Compile the class and execute it. The output `30000` will be shown on the screen after a while. You can experiment with the program by modifying the upper limit of the `for` loop so that a `String` object with a longer content is obtained. Then, you will find that the time required for program execution increases dramatically.

During the execution of the `TestString` program, 30000 `String` concatenation operations are performed, and hence 30000 `String` objects are created. Object creations are overheads to the program execution. For all the programs that we encountered in the course, such

an overhead is not very significant, because the program handles only few objects.

If a software application has to perform a lot of `String` concatenation operations, instead of using `String` objects, the Java standard software library provides the `java.lang.StringBuffer` (or `StringBuffer` for short) class that helps us resolve the problem.

Besides `String` objects, a `StringBuffer` object can be used to encapsulate a sequence of characters. The core difference between a `String` object and a `StringBuffer` object is that the sequence of characters maintained by a `StringBuffer` object can be modified, and it is therefore a mutable object.

Before discussing how to use the `StringBuffer` class to resolve `String` concatenation problems, please read the following to learn the basic usage of the `StringBuffer` class.

Reading 10.4

King, section 13.7, '`StringBuffer` class', pp. 579–83

From the reading, you learned that a `StringBuffer` object can store a string (that is, a sequence of characters) to be manipulated. Afterwards, an immutable `String` object can be obtained from the `StringBuffer` object.

A `StringBuffer` object encapsulates a sequence of characters and two core attributes — `length` and `capacity`. The capacity is the maximum number of characters it can store, and the length is the number of characters currently maintained. Therefore, the `capacity` of a `StringBuffer` object is always greater than its `length`. If an operation result exceeds the capacity of the `StringBuffer`, the value of the attribute `capacity` increases automatically. For inquiring and setting the attributes `capacity` and `length` of a `StringBuffer` object, the methods presented in Table 13.16 in the textbook can be used. The methods are `capacity()`, `ensureCapacity()`, `length()`, and `setLength()`.

String manipulation with `StringBuffer` class

To create a `StringBuffer` object, you can use any one of the constructors presented in Table 10.7.

Table 10.7 Constructors of the `StringBuffer` class

Constructor	Usage
`public StringBuffer()`	To create a `StringBuffer` object without characters, and the initial capacity is `16`.
`public StringBuffer(int length)`	To create a `StringBuffer` object with the initial capacity as specified by the parameter `length`.
`public StringBuffer(String str)`	To create a `StringBuffer` object with the supplied `String` object as its initial contents and the initial capacity exceeds the length of the initial string by `16`.

Once you are given a `StringBuffer` object, you can use the methods as presented in Table 10.8 to obtain the details of the encapsulated sequence of characters.

Table 10.8 The methods defined by the `StringBuffer` class for obtaining information about a `StringBuffer` object

Method	Usage
`public char charAt(int index)`	Gets the character at the specified `index`.
`public void getChars( int srcBegin, int srcEnd, char[] dst, int dstBegin)`	Copies a range of sequence of characters (specified by parameters `srcBegin` and `srcEnd`) to the destination `char` array (specified by the parameter `dst`) starting from the position specified by `dstBegin`.
`public int indexOf(String str)` `public int indexOf( String str, int fromIndex)`	Determines and obtains the position of the supplied `String` object in the sequence of characters encapsulated by the `StringBuffer` object.
`public int lastIndexOf(String str)` `public int lastIndexOf( String str, int fromIndex)`	Determines and obtains the last position of the supplied `String` object in the sequence of characters encapsulated by the `StringBuffer` object.
`public void setCharAt( int index, char ch)`	Sets a character, specified by `ch`, to a sequence of characters at the position specified by `index`.
`public String substring(int start)` `public String substring( int start, int end)`	Obtains a `String` object based on a range of characters encapsulated by the `StringBuffer` object.

Furthermore, you can use the string manipulation methods defined by the `StringBuffer` class to process the encapsulated sequence of characters. From the API documentation of the `StringBuffer` class, the methods that manipulate the sequence of characters and return the reference of a `StringBuffer` object are presented in Table 10.9.

Table 10.9 The methods defined by the `StringBuffer` class for manipulating the encapsulated sequence of characters

Method	Uses
`public StringBuffer append(...)`	Appends the textual representation of the supplied supplementary data to the existing sequence of characters.
`public StringBuffer delete( int start, int end)`	Deletes a range of characters from the current sequence of characters.
`public StringBuffer deleteCharAt(int index)`	Deletes a particular character specified by the index from the sequence of characters.
`public StringBuffer insert( int offset, ...)`	Inserts the textual representation of the supplied supplementary data to the current sequence of characters at the position specified by the parameter offset.
`public StringBuffer replace( int start, int end, String str)`	Replaces the range of characters of the current sequence of characters by the contents of the supplied `String` object.
`public StringBuffer reverse()`	Reverses the sequence of characters.

Most methods listed in Table 10.9 are discussed in an earlier reading. The `append()` and `insert()` methods of the `StringBuffer` class are overloaded and the ellipsis (. . .) in the method declarations indicates the parameter can be of various types. Both the `append()` and `insert()` methods may extend the sequence of characters. If the resultant sequence of characters exceeds the capacity of a `StringBuffer` object, the capacity of the `StringBuffer` object will increase automatically.

As the uses of the methods listed in Table 10.9 are quite straightforward, differences between string manipulations with `String` and `StringBuffer` classes are elaborated on in this section.

Comparing the `StringBuffer` methods that return a `StringBuffer` object and the `String` methods that return a `String`, a common difference is that methods of the `String` class return the reference of a newly created `String` object, whereas the methods of the `StringBuffer` class return the reference itself, that is, the value of the variable `this`.

To illustrate the difference between the methods of `String` class and `StringBuffer` class, two classes that encapsulate a reading are written as `TicketCounter1` and `TicketCounter2` in Figure 10.24 and Figure 10.25. They are the analogies of `String` and `StringBuffer` classes respectively.

```
// Definition of TicketCounter1
public class TicketCounter1 {
    // Attribute
    private int reading;    // The reading of the counter

    // Constructor
    public TicketCounter1(int reading) {
        this.reading = reading;
    }

    // Increase the reading by 1 and return a new TicketCounter1
    // object
    public TicketCounter1 increase() {
        return new TicketCounter1(reading + 1);
    }

    // Return the value of the attribute reading
    public int getReading() {
        return reading;
    }
}
```

Figure 10.24 TicketCounter1.java

```
// Definition of class TicketCounter2
public class TicketCounter2 {
    // Attribute
    private int reading;    // The reading of the counter

    // Constructor
    public TicketCounter2(int reading) {
        this.reading = reading;
    }

    // Increase the reading by 1 and return the reference of the
    // object itself
    public TicketCounter2 increase() {
        reading++;
        // Return the reference of this object
        return this;
    }

    // Return the value of the attribute reading
    public int getReading() {
        return reading;
    }
}
```

Figure 10.25 TicketCounter2.java

For the `TicketCounter1` class, every time the method `increase()` is called, a new `TicketCounter1` object is created with a new reading, but the attribute `reading` of the original `TicketCounter1` object is

kept unchanged. When the `increase()` method of a `TicketCounter2` object is called, however, the attribute `reading` of the object is increased by one and its reference is returned.

Let's consider the following two program segments

```
TicketCounter1 counter1 = new TicketCounter1(0);
counter1.increase().increase();
```

and

```
TicketCounter2 counter2 = new TicketCounter2(0);
counter2.increase().increase();
```

The interpretation of the statements `counter1.increase().increase()` is that the message `increase` is sent to the object referred by the variable `counter1`. As the method `increase()` returns a reference of type `TicketCounter1`, the second `increase` message is sent to the object referred by the reference returned by the first `increase()` method. Therefore, the above statement is the shorthand of the following statement segment:

```
TicketCounter temp = counter1.increase();
temp.increase();
```

For the former program segment, the scenario is visualized in Figure 10.26.

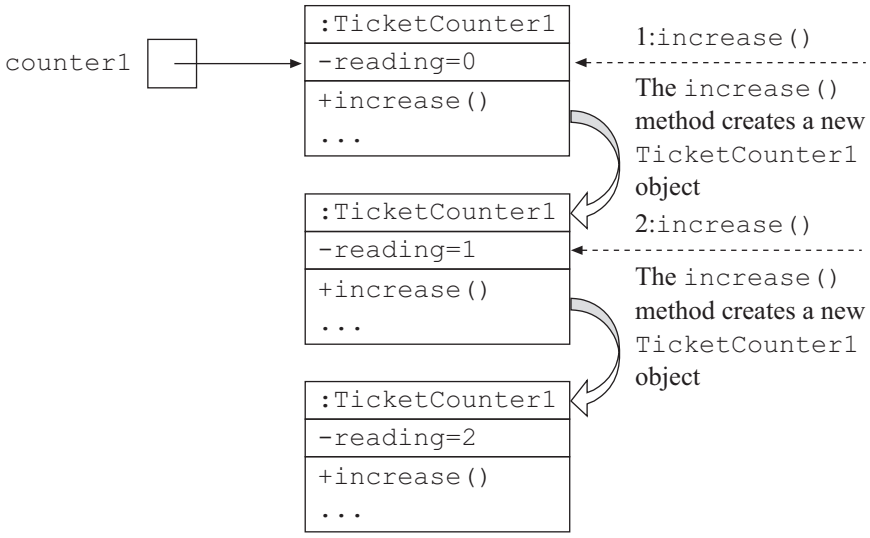


Figure 10.26 The `increase()` methods of two `TicketCounter1` objects are called

In Figure 10.26, the dashed arrows indicate the target objects to which the messages are sent, and the solid arrow denotes the object referred by the variable `counter1`. Every time the `increase()` method of a `TicketCounter1` object is called, a new `TicketCounter1` object is created with `reading` of value `1`. Then, the second increase message is sent to the newly created `TicketCounter1` object to create another new `TicketCounter1` object with `reading` of value `2`. Therefore, after the

execution of the former program segment, the `reading` of the `TicketCounter1` object is kept at `0`. It is the same for the `String` class that calling the methods of a `String` object creates a new `String` object and the original `String` object is kept unchanged and hence is immutable.

If you expect that the `TicketCounter1` object referred by the variable `counter1` is increased twice after executing the program segment, the second statement in the program segment should be modified to be:

```
counter1 = counter1.increase().increase();
```

Similarly, if you expect a `String` variable to refer to a `String` object that is the result of calling a method of the original `String` object, the statement should look like:

```
String str1 = " Hello ";
str1 = str1.trim().toUpperCase();
```

You should notice that executing the second statement of the above program segment creates two new `String` objects, and the last one is finally assigned to the variable `str1`.

The scenario of the second program segment mentioned above,

```
TicketCounter2 counter2 = new TicketCounter2(0);
counter2.increase().increase();
```

is visualized in Figure 10.27.

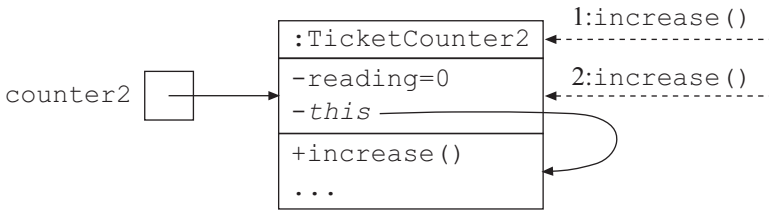


Figure 10.27 The `increase()` method of same `TicketCounter2` objects is called twice

As the `increase()` method of a `TicketCounter2` object always returns the reference itself (that is, the content of the keyword `this`), the two `increase` messages are sent to the same `TicketCounter2` object and no new objects are created. Furthermore, after executing the statement, the `reading` of the `TicketCounter2` object referred by the variable `counter2` is increased to `2`.

The `TicketCounter2` class illustrates the mechanism of calling a method of a `StringBuffer` object that returns a `StringBuffer`. Calling such methods creates no new `StringBuffer` objects, and the content of the original `StringBuffer` object is updated.

Now, we can further investigate the problem of inefficiency of `String` concatenation demonstrated by the `TestString` class (Figure 10.23). Your earlier reading mentions that the statement,

```
str = str1 + str2 + str3;
```

is equivalent to:

```
StringBuffer stringBuf = new StringBuffer();
stringBuf.append(str1).append(str2).append(str3);
str = stringBuf.toString();
```

Based on the same idea, the `String` concatenation operation used in `TestString` class,

```
str += "*";
```

is equivalent to:

```
StringBuffer stringBuf = new StringBuffer();
stringBuf.append(str).append("*");
str = stringBuf.toString();
```

Therefore, you can see that new objects are created repeatedly and substantial overhead is introduced. Here, using a `StringBuffer` object explicitly can greatly improve the performance, as illustrated in the `TestStringBuffer` class shown in Figure 10.28.

```
// Definition of class TestStringBuffer
public class TestStringBuffer {

    // Main executive method
    public static void main(String[] args) {
        // Create a StringBuffer object
        StringBuffer sb = new StringBuffer();
        // Append 30000 characters to the StringBuffer object
        for (int i=0; i < 30000; i++) {
            sb.append("*");
        }
        // Obtain a String object from the StringBuffer object
        String str = sb.toString();
        // Show the length of the String object for verification
        System.out.println(str.length());
    }
}
```

Figure 10.28 TestStringBuffer.java

Compile and execute the `TestStringBuffer` program. A `String` object with length 30000 is constructed by using a `StringBuffer` object, and the execution completes instantly. Study the `TestStringBuffer` program. You will find that a single `StringBuffer` object is manipulated to construct the sequence of

30000 characters, and a `String` object is finally obtained from the `StringBuffer` by the `toString()` method.

After manipulating the sequence of characters encapsulated by a `StringBuffer` object, you usually need to obtain a `String` object based on the `StringBuffer` object for actual use. Besides using the `toString()` method of a `StringBuffer` object to get a `String` object, as illustrated in the `TestStringBuffer` class definition (Figure 10.28), you can use the constructor of the `String` class to create a new `String` object based on an existing `StringBuffer` object:

```
public String(StringBuffer buffer)
```

For example, the following program segment uses the above constructor to create a `String` object:

```
StringBuffer sb = ...;
String str = new String(sb);
```

Please use the following self-test to check your understanding of the use of `String` class and `StringBuffer` class.

Self-test 10.7

1 Evaluate the results of the following expressions.

a `"ABCDEF".toLowerCase().substring(1,3)`

b `new Double(100).toString().length()`

c
```
new StringBuffer(" ABC ").
append(1).append(2).append(3).toString().trim()
```

2 Write an `Encoder` class that accepts a `String` object, encodes its contents and returns a resultant `String` object. For the encoding processing, all character `A`'s are replaced by `B`'s, all character `B`'s are replaced by `C`'s, and so on. Character `Z`'s are replaced by character `A`'s. The process is similar for small letters. Therefore, the encoded `String` object for the `String` object `"Hello XYZ"` is `"Ifmmp YZA"`.

Review the definition of the `PrimitiveList1` class (Figure 10.39 in Appendix E). You will find that the `toString()` method of the class relies on `String` concatenations to construct the resultant `String` object to be returned. Based on our discussion in this section, `StringBuffer` is a better choice for performing `String` concatenation. An enhanced version, `PrimitiveList2`, is written and is discussed in Appendix F of this unit. Please refer to the appendix for further details.

Summary

This is the last unit of the course. In this unit, many utility classes are introduced, such as collection classes for various data structures, wrapper classes for manipulating primitive values and string manipulations with the `String` class and the `StringBuffer` class. Furthermore, we discussed how to design and implement a new data structure if all existing collection classes fail to fulfill your specific requirements.

Whenever a collection of data items can be accessed by a single name, it is considered a data structure. Based on this definition, a single object that encapsulates its own attribute can be considered a data structure. However, the term data structure usually refers to a collection of data items or objects that can be accessed as an entity. Therefore, an array object is more appropriate to be called a data structure. Array was introduced in *Unit 5* and further elaborated on in *Unit 6*. You learned that arrays are implemented as array objects in the Java programming language. Therefore, once an array object is created, its size and element types are fixed and unchangeable.

As it is often necessary to consolidate a collection of data items or objects for further processing, the Java standard software library provides many classes in the `java.util` package for purposes that fit different requirements. Based on their features, they are categorized into `List` and `Set`. `List` and `Set` are the general types for the implementations `List` and `Set`, and the abstract superclasses of the concrete specific implementation classes, such as `ArrayList` and `HashSet`. More exactly, `List` and `Set` are Java interfaces that define the common methods exposed by a `List` type or `Set` type object, and the implementation adopted by a class is up to the designer and developer of the class. Furthermore, the Java interfaces `List` and `Set` are subsequently the subinterfaces of the general type `Collection`, which is also defined as a Java interface in the `java.util` package.

Another type of data structure that we discussed is the map data structure defined by the Java interface `Map` in the `java.util` package. Such data structures maintain the mappings between key objects and value objects. When a `Map` object is given the key object, the `Map` object can return the corresponding value object if the `Map` object maintains such a key-value object pair. A sample implementation is the `HashMap` class.

`List`, `Set` and `Map` are three data structures provided by the Java standard software library. A `List` object maintains the order among the objects so that each object can be accessed by specifying an index. Therefore, a `List` object can usually be used as a substitute for an array object. A `Set` object can be used to maintain a collection of objects with no duplications. As a result, you can repeatedly add objects to a `Set` object and it is guaranteed that the `Set` object maintains no object more than once. A `Map` object can be used if it is necessary to maintain the key-value object pairs, so that providing a key object can allow access to the corresponding value object.

The `Collection` classes can maintain any objects except primitive values. In order to solve such a problem, eight wrapper classes are defined for the eight primitive types of the Java programming language. All wrapper class objects are immutable, and each can encapsulate a value of the corresponding primitive type. Furthermore, wrapper classes define some utility methods for conversions, especially the conversion between `String` and the corresponding primitive type.

Finally, we discussed the possible string manipulations by calling the methods defined by the `String` class. Of all string manipulation operations, string concatenation is frequently used. A sample program illustrates that string concatenation is a slow operation that can be a performance problem. The reason behind such poor efficiency is that `String` objects are immutable, and objects are repeatedly created and removed by the JVM during string concatenations with `String` objects, which contribute most of the execution time. The Java standard software library provides the `StringBuffer` class that is preferable for string concatenations. Unlike a `String` object, a `StringBuffer` object maintains a sequence of characters that can be updated. If any operation with the `StringBuffer` object causes the sequence of characters to exceed the capacity of the `StringBuffer` object, its capacity will be extended automatically.

This unit mainly introduced you to some commonly used utility classes, so that some operations that you thought you might have to handle yourself are implemented by using those utility classes. As you learn more and more utility classes provided by the Java standard software library or third-party developers, you will find that you are better equipped for more complex and complicated software development with the Java programming language.

Summary of the course

The end of the last unit of the course a suitable place to summarize what you have learned in the course.

This course was probably your first computer science course. Therefore, you acquired fundamental knowledge about computers in *Unit 1* of the course. To perform operations with a computer system, you need hardware and software that refer to physical components that constitute a computer and the programs with data to be executed by the computer equipment. Hardware and software must match so that the hardware can execute the software. Then, we discussed the core components of typical computer hardware and their roles. For the software side, you learned that software is written in a particular programming language. With respect to developing software, the object-oriented approach and object-oriented programming languages were used in the course because it is the natural way of analysing a problem and handling it. Of all object-oriented programming languages, the Java programming language was chosen for this course.

In *Unit 2*, we said that problems are usually complicated nowadays and it is necessary to use a systematic approach to develop a software application to solve problems. By consolidating the experiences accumulated by software developers, the tasks to be performed throughout the life of development and using the software usually follow a common pattern. This is known as the *software development cycle* and the sequence of tasks is categorized into phases. Afterwards, we discussed key concepts in object-oriented software development and what classes and objects are in the object-oriented paradigm and their relationships. Basically, classes are blueprints for a group of objects of the same type. You write classes to define the behaviours and attributes of a type of object, and objects are created while executing the programs. By manipulating the objects, the objects mimic real-world situations for solving desired problems. Finally, you learned the approaches to analysing a problem and designing the classes in an object-oriented way.

By what you learned from *Unit 2*, you can derive classes if you are given a problem. Then, *Unit 3* equipped you with the knowledge to implement the derived classes with the Java programming language. You learned what classes, methods and attributes are, and their relationships with the derived candidate classes, by analysing the problem. The unit also provided you with fundamental knowledge of programming in the Java programming language, such as identifiers, dot notation, the eight primitive types, non-primitive types (or class type), and how to perform numeric operations.

Based on what you learned from *Unit 3*, you could write classes defining methods and attributes that correspond to object behaviours and attributes. All methods discussed before *Unit 4* specify sequences of operations to be performed, and they are executed sequentially. In *Unit 4*, you learned program constructs — branching (or selection) and repetition (or looping) — that enable program segments to be executed

conditionally and repeatedly. Then, object behaviours can be written with a kind of 'intelligence' so that it can behave differently in different situations.

As computers are getting faster, software is handling more and more data. Individual variables are no longer capable of maintaining those data. Therefore, array was introduced in *Unit 5*. Arrays in the Java programming language are implemented as objects that can be used to maintain data of the same type. We discussed how to create and handle them. With an array object, it is a common necessity to determine whether it contains a particular data item. Such an operation is known as searching, and we discussed two frequently used approaches — linear searching and binary searching — and their applications.

You learned in *Unit 6* the advanced uses and applications of what you learned from *Unit 5*, for example, building complex `boolean` expressions with `boolean` operators, nested branching constructs and looping constructs. In *Unit 5*, you learned that binary searching is much faster than sequential searching, but the data to be searched must be sorted in a particular order. Many sorting algorithms are available, and *Unit 6* discussed a simple one — insertion sorting. By using such a sorting algorithm, you can sort the items maintained by an array object. You learned the uses and applications of multi-dimensional arrays from *Unit 6*. Finally, an advanced control structure was introduced — recursion. Such a technique is especially useful for solving problems that can be resolved or converted to the same and simpler problems and the solutions to the based problems are well defined. A typical characteristic of the programs that implement recursion is a method directly or indirectly calling itself.

Unit 7 is the core unit for object-oriented programming. It first introduced the concepts of data hiding and encapsulation. Basically, classes that realize these two concepts define methods and attributes to be `public` and `private` respectively. For each attribute, `public` getter/setter methods are provided. The keywords `public` and `private` specify two different access control privileges, in which `public` members (methods or attributes) can be accessed universally and `private` members can only be accessed by objects of the same class. Constructors, which are executed implicitly during the creation of an object, were introduced in the unit. We said that class definitions could be reused by writing a new class that *extends* an existing one. The new class is known as subclass or child class, and the existing one is known as superclass or parent class. Then, it is necessary to write the specific methods or attributes for the subclass, and you can consider all methods defined in the superclass are inherited to the subclasses. Such a phenomenon is known as *inheritance* and it realizes the 'is a' relationship. If the subclass defines members that are already defined by the superclass, the members defined by the subclass are considered as overriding the ones inherited from the superclass. If an object receives a message that corresponds to a method that is defined by both the superclass and subclass, the method defined by the *subclass* is executed. In the Java programming language, a variable of a particular class type

can refer to an object of the class and all its subclasses. Then, whenever a message is sent to the object referred by such a variable, the method to be executed depends on the real type of the object. This is known as *polymorphism*. Finally, in the unit, abstract classes were introduced that define abstract ideas or general objects. They usually define `abstract` methods to be implemented by concrete subclasses that model the corresponding specific types.

With what you learned from *Unit 7*, we could then discuss how to use the classes provided by the Java standard software library, because the classes are written in the object-oriented approach. Therefore, if you are going to use them by either extending them or simply creating objects of the classes for manipulating, you need to apply your knowledge of inheritance and polymorphism. *Unit 8* to *Unit 10* introduced you to the ways, and the classes to be used, for handing particular aspects of operations.

Before *Unit 8*, all programs were handling the data supplied to them during runtime, and operation results were obtained and displayed accordingly. However, once the program terminated, all supplied data and the operation results were lost. In *Unit 8*, you first learned how to manipulate files or directories stored on a computer drive with the `java.io.File` class (or `File` class for short). Afterwards, we discussed the stream-based input/output operations supported by the classes provided in the `java.io` package. There are basically two types of stream — byte-based and character based — and two possible directions of data flow. Byte-based streams and character-based streams are suitable for binary and textual data respectively. Furthermore, exceptions were introduced in the unit, which indicated runtime errors. We discussed the use of the `try/catch` construct to handle occurred exceptions and the importance of implementing exception handling in software development.

The requirements of a software application are not only to produce correct operation results but also a user-friendly user interface that makes it easier for users to use, such as a graphical user interface. In *Unit 9*, you learned how to create GUIs with the classes in the `javax.swing` package of the Java standard software library. To create a GUI, you first have to choose the top-level container object of the GUI, which is probably a `JFrame` or a `JDialog`. Afterwards, you choose a suitable layout manager for the container object and GUI components are then added to the container. We discussed the common layout managers and common GUI components. The GUI can be shown on the screen but cannot respond to user operations such as clicking the mouse on a GUI component. Then, event handling was introduced. All user operations can be considered to be events. Whenever a user operates the GUI, such as clicking a button with the mouse pointer, an event object is created and the button is considered the event source. Events are categorized, and if the GUI has to handle a particular event category, it is necessary to register a suitable event handler object with the GUI component. The class of an event handler object must implement a suitable Java interface such as `ActionLister`. Java interfaces are variants of an abstract class

in which all its methods are `abstract`. Its subclasses do not use the `extends` clause, but the `implements` clause refers to it. Then, the event handler class has to implement all methods defined by the Java interface. For example, to write a class that can handle an `Action` category event, which is a typical event of a GUI component, it must implement the `ActionLister` interface and define a concrete `actionPerformed()` method. Finally, we discussed what Java applets are, their differences from usual GUI programs, and how to develop them.

The last unit of the course, *Unit 10*, introduced mostly utility classes, such as the `Collection` classes for maintaining collections of objects, wrapper classes for encapsulating primitive values and `String`/`StringBuffer` classes for string manipulations. The unit first discusses what data structures are and the use of `Collection` classes in the `java.util` package. Furthermore, the queue data structure was discussed to illustrate how to define a data structure. `Collection` objects can maintain any objects except primitive values. To resolve the issue, wrapper class objects can be used to encapsulate primitive values so that they can be added to `Collection` objects. String manipulations are common operations for constructing textual representations or extracting data from a sequence of characters. Before *Unit 10*, we only used `String` objects that were immutable. That is, their contents cannot be changed. Therefore, string manipulations with `String` objects create new `String` objects repeatedly and can reduce execution efficiency. `StringBuffer` can also maintain a sequence of characters like a `String` object does. The core difference is the sequence of characters encapsulated by a `StringBuffer` object can be updated, so it is more suitable for string manipulations.

The above is a summary of what you should have learned by the end of the course. Developing software with an object-oriented programming language like the Java programming language is interesting, challenging and rewarding. As you write more software applications with the Java programming language and experiment with more classes provided by the Java standard software library, you will find that you will become a more and more experienced Java developer.

See you in the advanced programming courses!

Appendix A: Determining the collection classes to be used

In this unit, we discussed several data structures: list, queue, stack, set and map. All these data structures can maintain a collection of objects. They are categorized by their behaviours or the methods they offer. For example, the common methods to be used with the various data structures are shown in Table 10.10.

Table 10.10 The common methods exposed by different data structures

Data structure	Commonly used methods
List	`public void set(int index, Object obj)` `public Object get(int index)`
Queue	`public void enqueue(Object obj)` `public Object dequeue()`
Stack	`public void push(Object obj)` `public Object pop()`
Set	`public void add(Object obj)` `public Object[] toArray()`
Map	`public void put(Object key, Object value)` `public Object get(Object key)`

For the five data structures presented in Table 10.10, `List`, `Set` and `Map` are defined as Java interfaces in the `java.util` package. For the queue and stack data structures, the methods shown are usually expected. These types define the behaviours or methods the objects must possess, and they are therefore usually defined as Java interfaces. The implication is that programmers are free to choose their own approaches to build such data structures.

In this appendix, the common approaches that can be used to build the above-mentioned data structures and their characteristics are presented so that you can choose the most suitable approach according to your requirements.

Possible implementation approaches

The baseline of a data structure is to maintain a collection of objects. Therefore, the different approaches refer to the different ways to maintain the collection of objects. The common ways for maintaining a collection of objects are array, linked list, tree and hash table.

Maintaining objects with an array object

An array object with element type `Object` can maintain a collection of objects by storing the reference of the objects to be maintained in the array elements. The scenario is illustrated in Figure 10.29.

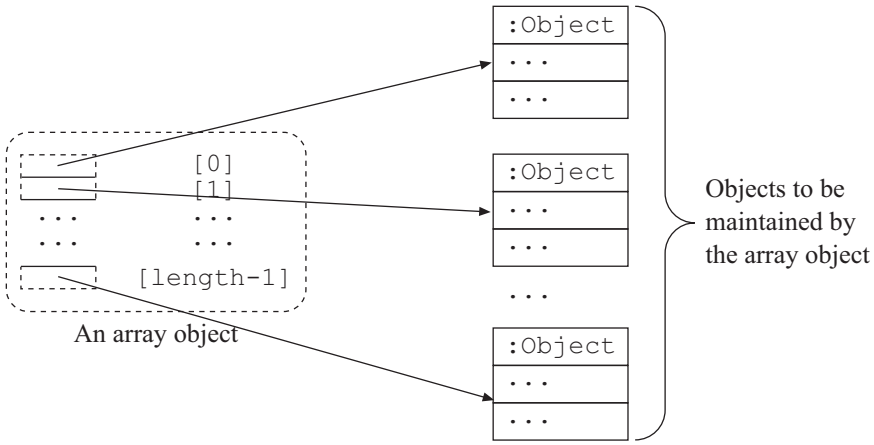


Figure 10.29 The scenario of objects maintained by an array object

In Figure 10.29, the attribute `length` of the array object is not shown. Each array element can store the reference of an object that the array object maintains.

The characteristic of maintaining objects with an array object is that any object maintained by an array object can be accessed by supplying an index or subscript and the time required to access any object is the same. The downside of using array object is the size of an array object is fixed. If a new object is to be maintained by the array object when all elements of an array object are used, a new array object with a larger size has to be created with element copying. Furthermore, rearranging the contents of the array element contents, such as shifting the indexes for the objects, it is necessary to copy the contents of the array elements one by one.

Maintaining objects with a linked list

A linked list is composed of a sequence of node objects, and each node object stores the reference of the object to be maintained and the reference of the node object that follows it. Visually, the scenario of maintaining objects with a linked list is illustrated in Figure 10.30.

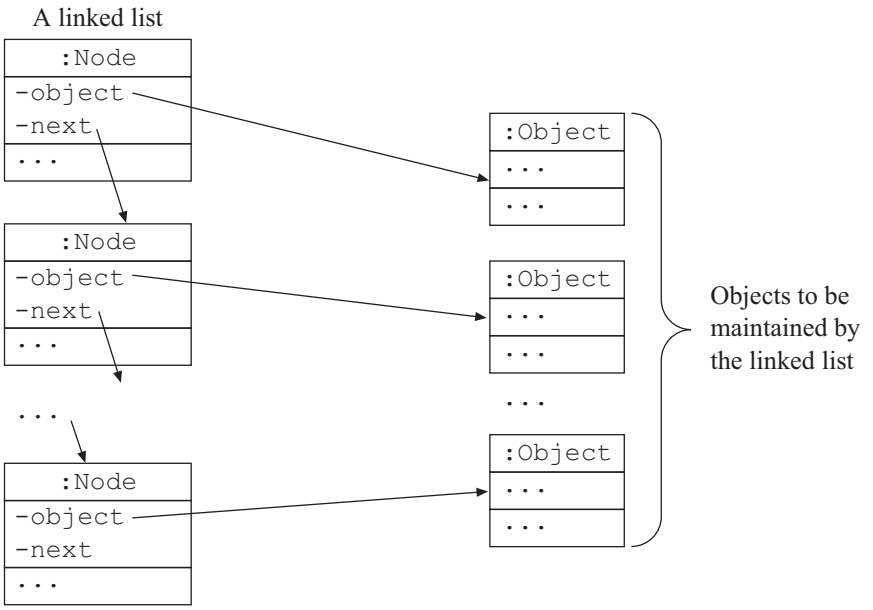


Figure 10.30 The scenario of objects maintained by a linked list

From Figure 10.30, you can see that each node object stores the reference of the object that is maintained by the list, and the reference of the next node object. Therefore, given the reference of the first node object, it is possible to traverse the linked list to locate all objects. You can imagine that the operations of adding a new object and removing an object at any location of the linked list require changing a few references of its adjacent node object. Locating a particular object in the linked list needs to search the entire linked list sequentially from the first one to the last one.

Maintaining objects with a tree

Figure 10.2 illustrates how a tree data structure can be used to maintain the directories and files stored on a computer drive. A similar structure can be used to maintain a collection of objects as illustrated in Figure 10.31.

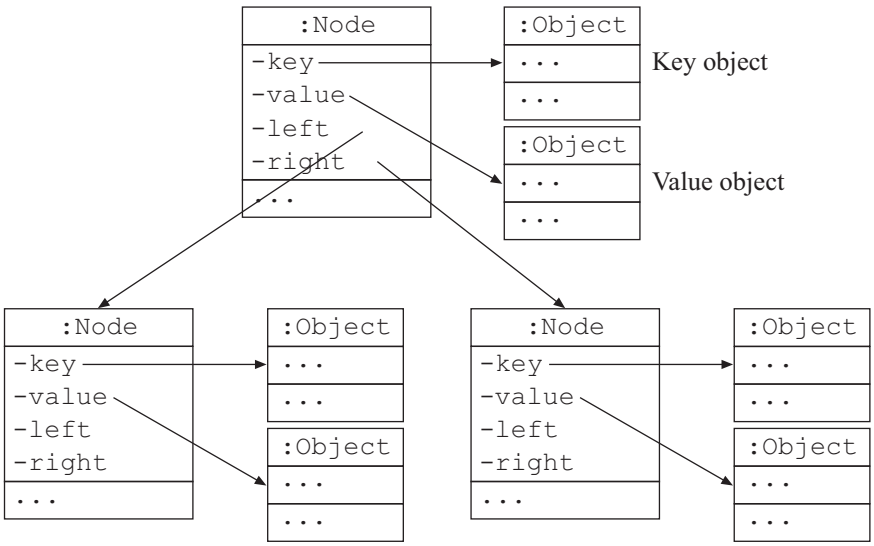


Figure 10.31 The scenario of maintaining objects with a tree data structure

In Figure 10.31, each node object refers to a key object and a value object. When we discussed the data structure tree, another tree data structure shown in Figure 10.3 illustrated a tree data structure for maintaining a sequence of numbers. The assignment of the numbers to the nodes was based on the comparison results among the numbers. Therefore, a tree data structure is suitable for maintaining the type of data items or objects that can be compared, such as `String` objects.

You can see from Figure 10.31 that each node of a tree data structure maintains the references to a key object, and a value object and the key object is used for comparison for arranging the nodes. Therefore, to locate a value object by a target key object, the target key object is compared with the key object of each node object starting from the root node object. Each comparison of the target key object with the key object stored by a node object can yield three possible results. They are:

1 the target key object equals the key object of the node object,

2 the target key object can be found on the left sub-tree of this node object, which are the node objects starting from the one referred by the `left` attribute of the node object, or

3 the target key object can be found on the right sub-tree of this node object, which are the node objects starting from the one referred by the `right` attribute of the node object.

Based on the key object comparison result, every comparison can therefore eliminate about half of the node objects to be searched. The efficiency of locating a key object is similar to that of the binary searching we discussed in *Unit 5*. As a result, a tree data structure is suitable for maintaining key-value object pairs; that is, implementing the `Map` object.

What if the key object and the value object are referring to the same object in all node objects of a tree? Then, as a tree data structure is good at locating a target key object among all node objects it maintains, such a tree is suitable to determine whether the tree already maintains an object. If the tree maintains no such object, the object can be added to the tree. Then, the tree is practically a `Set` object.

Maintaining objects with a hash table

Before introducing what a hash table is, you first have to understand what a hashing function is.

A hashing function is a mathematical function that computes a numeric value based on the given data. For example, there is a hashing function that can generate a numeric value based on a `String` object. The numeric values obtained by the function are `12`, `6` and `15` for the `String` object `"Hello"`, `"World"` and `"OUHK"` respectively.

A hash table is a data structure that defines a hashing function and has a storage that can be accessed by a value, such as an array object. For example, a hash table that uses the above sample hashing function has an array object with size 30 as its storage. The `String` objects `"Hello"`, `"World"` and `"OUHK"` are first processed by the hashing function to obtain the value `12`, `6` and `15` respectively. Then, these three `String` objects are stored at the storages `12`, `6` and `15` accordingly. The scenario is visualized in Figure 10.32.

String object	Hashing function result
"Hello"	12
"World"	6
"OUHK"	15

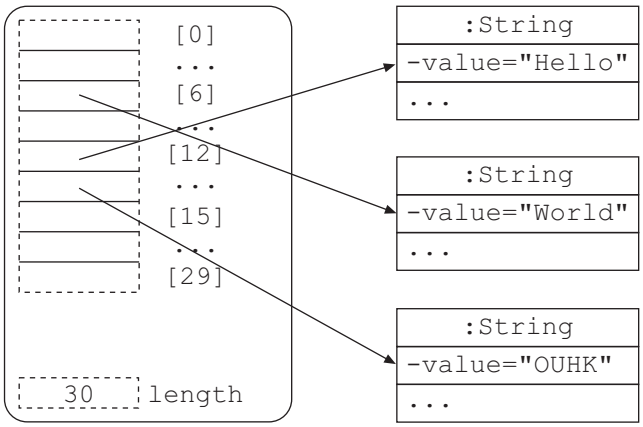


Figure 10.32 The scenario of maintaining objects with a hash table

To locate a `String` object in the hash table, the hashing function is applied on the `String` object first. If the `String` object is `"Hello"`, the value obtained from the hashing function is `12`, and it is then possible to access the storage immediately with value `12` to determine whether such `String` object exists. The discussion here uses `String` objects for illustration. Actually, the hashing function and the storage of the hash table can accept objects of any type.

You can see that maintaining objects with a hash table enables a target object to be searched by applying a mathematical function to it and accessing the storage directly. As a result, a hash table enables efficient searching of objects, and it is therefore suitable for implementing `Set` data structures. If the object supplied to the hashing function is the key object and the object maintained by the storage is the value object, a hash table practically becomes a `Map` data structure.

Common operations with a data structure

To choose the suitable approach for maintaining objects, you should first determine the operations you are going to perform with the data structure. The common operations are:

1 adding/removing an object,

2 accessing an object with an index or key object, and

3 determining the existence of an object.

In previous sections, we discussed four approaches to maintaining objects: array, linked list, tree and hash table. Table 10.11 gives you the characteristics of each approach in performing the above types of operation.

Table 10.11 The matrix of operation performance by different approaches for different operations

Operation	Array	Linked list	Tree	Hash table
Adding/removing object	Slow. Array creation or element copying may be required	Fast. Only a few references of the linked list need changes, especially at the front and at the end of the list.	Slow. Need searching and the tree may need restructuring. The objects added to the tree are implicitly sorted.	Depends! Fast if the storage is available. Otherwise, it is slow if the storage to be used is occupied. An extra mechanism is required to find new storage to maintain the object.
Accessing an object with an index or key object	Fast. The desired element can be accessed immediately with a subscript.	Slow. Searching from the beginning of the list is required.	Fast. Searching is performed by comparing the key object with the target object.	Fast. Hashing function is applied on the key object and the desired storage is accessed right away.
Determining the existence of an object	Slow. Sequential searching is required.	Slow. Sequential searching is required.	Fast. Searching is performed by comparing the key object with the target object for determination.	Fast. Hashing function is applied on the key object and the desired storage is accessed right away for determination.

The performance of different approaches for different operations given in Table 10.11 provides a relative sense of the general effectiveness among the approaches. The actual performance depends on the number of objects to be maintained and the characteristics of the objects to be maintained. For example, if you are going to maintain a few objects, all

operations by all approaches can complete immediately. However, if your program is going to manage thousands of objects, performance becomes critical. Table 10.11 can give you some insight into the different approaches.

The implementation classes provided by the `java.util` package

Each implementation class in the `java.util` package has two aspects — the set of behaviours such a class possesses, and the approach it uses to implement those behaviours. Table 10.12 is a summary of the set of behaviours and approaches of all collection classes provided in the `java.util` package.

Table 10.12 The approaches and behaviours of the implementation classes in the `java.util` package

		Approaches of implementation			
		Array	Linked List	Tree	Hash table
Behaviour	List	`ArrayList` `Vector`	`LinkedList`		
	Set			`TreeSet`	`HashSet`
	Map			`TreeMap`	`HashMap` `Hashtable`

In Table 10.12, behaviour concerns the external behaviour of a `Collection` object and the approaches of implementation concern the way the external behaviour of a `Collection` object is implemented. With the table, you should first determine the necessary external behaviour; that is, `List`, `Set` or `Map`. Then, based on the operations you are going to perform with the `Collection` object, you can further figure out the suitable implementation class.

The two pairs of implementation classes, `ArrayList`/`Vector` and `HashMap`/`Hashtable`, possess the same behaviour and use the same implementation approaches. There are differences between the two implementations, but these are outside the scope of the course. For the time being, you can choose either one of the two arbitrarily. For further details of the implementation classes, please refer to the API documentation.

While you are choosing an implementation class and your only concern is the execution performance, you can experiment with their performances by executing the program with the same set of objects and operations with different implementation classes. In order to switch the implementation class easily, you should declare the variable with the suitable Java interface type instead of an implementation class. For example, the following declaration

```
List list = ...;
...
```

is preferable to the following one:

```
ArrayList list = new ArrayList();
...
```

The reason is if you are going to change the implementation class, the only change to the former one is

```
List list = new ArrayList();
```

whereas the latter one concretely declares the list data structure must be an `ArrayList` object. Then, it is necessary to change all variables of type `ArrayList` to the new one, such as `LinkedList`.

Appendix B: Implementation of an integer queue

We discussed the design and implementation of a queue data structure that can manage a collection of objects in which the handling of the objects follows the first-in-first-out pattern. The implementation we discussed, the `LinkedQueue` class (shown in Figure 10.16) can manage objects of any type except primitive values. If a `LinkedQueue` object is used to model a queue of primitive values of type `int`, wrapper objects have to be used. However, the use of wrapper classes will introduce overheads to the program execution, such as object creations and method calls for primitive value retrievals.

If efficiency is critical and the usage of `LinkedQueue` object with wrapper class objects cannot fulfill the requirements, a possible solution is to write an implementation for an integer queue. For example, a sample implementation, `IntegerQueue` class, is shown in Figure 10.33.

```
// Definition of class IntegerQueue
public class IntegerQueue {
    // The capacity of the queue
    private int capacity;
    // The starting subscript of the queue in the array
    private int start = 0;
    // The ending subscript of the queue in the array
    private int end = 0;
    // The count of integers maintained by the queue
    private int count = 0;
    // The array object for maintaining the number
    private int[] storage;

    // Constructor
    public IntegerQueue(int capacity) {
        // Store the capacity and create the array object
        this.capacity = capacity;
        storage = new int[capacity];
    }

    // Add an integer at the end of the queue
    public void enqueue(int value) {
        if (count < capacity) {
            // If there are still rooms for storage, store the integer,
            // increase the count and adjust the end of the queue
            storage[end] = value;
            count++;
            end = (end + 1) % capacity;
        }
        else {
            // Otherwise, show error message
            System.err.println("Error: The queue is full");
        }
    }
```

```
    // Remove and return the number at the start of the queue
    public int dequeue() {
        int result = -1;
        if (count > 0) {
            // If there are numbers, get the number, decrease the count,
            // and adjust the start of the queue
            result = storage[start];
            count--;
            start = (start + 1) % capacity;
        }
        else {
            // Otherwise, show error message
            System.err.println("Error: No number in the queue");
        }
        // Return number
        return result;
    }
}
```

Figure 10.33 IntegerQueue.java

The `IntegerQueue` class is a simple implementation for maintaining a queue of integers. It uses an array object to maintain the integers. The size of the array object is determined while executing the constructor, and the array size will not expand automatically. The definition does not use wrapper class objects for encapsulating the integers. An array object is used to maintain the integers. Therefore, its performance is better than using a `List` object with `Integer` objects.

A driver program `TestIntegerQueue` is written for testing the `IntegerQueue` class. Please refer to the course CD-ROM or website for the definition.

Appendix C: A Collection class for a dedicated type

In *Unit 2*, while we were discussing how to analyse the payroll calculation problem, a candidate class `StaffList` was identified for maintaining the list of `Staff` objects. By what you have learned in this unit, you can simply use any implementation class of the Java interface `List` to play the role of the `StaffList` class, such as:

```
List staffList = ...;
...
staffList.set(0, new Staff(...));
```

Using a `List` object is simpler, but deriving a `Collection` class for a dedicated type can provide you with extra advantages. For example, a `StaffList` class is actually defined for maintaining `Staff` objects. A possible implementation is shown in Figure 10.34.

```
// Resolve classes in java.util package
import java.util.*;

// Definition of class StaffList
public class StaffList extends ArrayList {
    // Set the Staff object to the specified location
    public void setStaff(int index, Staff staff) {
        set(index, staff);
    }

    // Get the Staff object from the specified location
    public Staff getStaff(int index) {
        return (Staff) get(index);
    }
}
```

Figure 10.34 StaffList.java

The `StaffList` class defines two methods for illustration; more methods might be required for practical use. The two methods, `setStaff()` and `getStaff()`, simply call the corresponding inherited methods `set()` and `get()` for setting a `Staff` object into storage and getting a `Staff` object from storage respectively.

The added value of the `StaffList` class is that the object supplied to the `addStaff()` method must be a `Staff` object. (If the `Staff` type is `abstract` or is a Java interface, the object supplied to the method must be an object of any concrete subclass of the superclass `Staff`.) Otherwise, the compiler can detect any discrepancy and give you compile-time errors. For example, the following statement will cause a compile-time error:

```
StaffList staffList = new StaffList();
...
staffList.setStaff(0, new Date());
```

The compiler can determine that the object supplied to the `addStaff()` method is not a `Staff` object at compile-time. Otherwise, if the staff list is implemented by a general `List` object, such as

```
List staffList = ...;
...
staffList.set(0, new Date());
```

the compiler will accept the above program segment because the `set()` method of a List object can accept objects of any type. It will still cause an error, but the error only occurs at runtime when the object is obtained by calling the `get()` method, such as:

```
Staff staff = (Staff) staffList.get(0);
```

Then, the type of object obtained from the `List` object is not `Staff` and the casting operation will cause a runtime error or exception. As the `setStaff()` method of the `StaffList` guarantees the object added to the list must be `Staff` objects, the statement in the `getStaff()` method

```
return (Staff) get(index);
```

securely casts the type of object to `Staff` type and is returned. Furthermore, the statement calling the `getStaff()` method does not need casting. For example, the statement that calls the `getStaff()` method needs no casting operation:

```
StaffList staffList = new StaffList();
...
staffList.setStaff(0, new Staff(...));
...
Staff staff = staffList.getStaff();
```

Developing a specific collection class for dedicated classes can improve the reliability of your software. However, it can be tedious to write such specific collection classes, especially for a software application that involves a lot of candidate classes. You are the designer and the developer of the software, and it is up to you to choose the approach or compromise between the two.

Appendix D: A List of boolean values — `BitSet`

An implementation: `BitSet`

From the first reading in this unit, you learned that a `BitSet` can be considered an `ArrayList` object that maintains a sequence of `boolean` values and each `boolean` value can be specified by an index. Such a class is suitable for maintaining a large collection of two-state values. In *Unit 6*, the class `Enrolment` uses a single value of type `int` to represent the course enrolled in by using one bit out of the 32 bits to denote a course. With the same principle, you can use a value of type `long` to represent 64 different two-state values. If you want to maintain more than 64 two-values, you may find `BitSet` useful.

The `Enrolment` class discussed in *Unit 6* is modified to use a `BitSet` to represent the courses enrolled. The complete class definition is shown in Figure 10.35.

```
// Resolve BitSet class in java.util package
import java.util.BitSet;

// Definition of class Enrolment
public class Enrolment {
    // Constant variables for different courses
    public static final int COURSE_M205   = 0;
    public static final int COURSE_M221   = 1;
    public static final int COURSE_M245   = 2;
    public static final int COURSE_M246   = 3;
    public static final int COURSE_M261   = 4;
    public static final int COURSE_MST204 = 5;
    public static final int COURSE_MST207 = 6;
    public static final int COURSE_MT201  = 7;
    public static final int COURSE_MT210  = 8;
    public static final int COURSE_MT258  = 9;
    public static final int COURSE_MT260  = 10;
    public static final int COURSE_MT268  = 11;
    public static final int COURSE_MT269  = 12;
    public static final int COURSE_S271   = 13;
    public static final int COURSE_T202   = 14;
    public static final int COURSE_T222   = 15;
    public static final int COURSE_T223   = 16;
    public static final int COURSE_T225   = 17;
    public static final int COURSE_TM222  = 18;

    // Immutable array object maintaining course codes
    public static final String[] courseCodes = {
        "M205", "M221", "M245", "M246", "MT261",
        "MST204", "MST207", "MT201", "MT210", "MT258",
        "MT260", "MT268", "MT269", "S271", "T202",
        "T222", "T223", "T225", "TM222"
    };

    // Immutable array object maintaining course names
    public static final String[] courseNames = {
```

```
            "Fundamentals of Computing",
            "Mathematical Methods",
            "Probability and Statistics",
            "Elements of Statistics",
            "Mathematics for Computing",
            "Mathematical Models and Methods",
            "Mathematical Methods, Models and Modelling",
            "Computing Fundamentals with Java",
            "Computing Fundamentals",
            "Programming and Database",
            "Computer Architecture and Operating Systems",
            "Commercial Information Processing",
            "Commercial Information Systems and Programming",
            "Discovering Physics",
            "Analogue and Digital Electronics",
            "Electronics Principles and Digital Design",
            "Microprocessor-based Computers",
            "Analogue Circuits",
            "The Digital Computer"
        };

        // The variable representing the courses a student is taking,
        // and the enrolment list initially contains no course.
        private BitSet coursesTaking = new BitSet(19);

        // Add a course to the Enrolment
        public void addCourse(int course) {
            coursesTaking.set(course);
        }

        // Remove a course from the Enrolment
        public void removeCourse(int course) {
            coursesTaking.clear(course);
        }

        // Toggle a course in the Enrolment
        public void toggleCourse(int course) {
            coursesTaking.flip(course);
        }

        // Determine whether the course is chosen
        public boolean contains(int course) {
            return coursesTaking.get(course);
        }

        // Show the chosen courses in this Enrolment
        public void showCourses() {
            int courseCount = 0;
            for (int i = 0; i < 19; i++) {
                if (contains(i)) {
                    courseCount++;
                    System.out.println(courseCount + ": " +
                        courseNames[i] + " [" +
                        courseCodes[i] + "]");
                }
            }
        }
    }
```

Figure 10.35 Enrolment.java

Compared with the `Enrolment` class discussed in *Unit 6*, the definition shown in Figure 10.35 is easier to understand, because the numbers representing the courses are consecutive integers and the methods `set()`, `clear()`, `flip()` and `get()` are more self-explanatory than using bitwise operators.

However, `BitSet` is not a data structure for maintaining objects. Therefore, it is not considered a `Collection` object. To maintain a collection of objects with no duplications, the `java.util` package of the Java standard software library provides you two implementation classes — the `HashSet` and `TreeSet`.

Appendix E: Wrapper classes

Wrapper classes are not needed in the collection of JDK1.5 and this is only for your reference. In the previous sections, we said that collection classes enable us to store or maintain a group of elements. As the element type of each storage unit of a collection object is `Object`, which is the ultimate superclass of all classes in the Java programming language, each storage unit of a collection object can maintain any object by storing its reference. However, a collection object cannot maintain primitive type values, as they are not reference types and are not subclasses of the class `Object`.

To resolve the problem, the Java standard software library defines wrapper classes for encapsulating primitive values.

Wrapper classes as immutable objects of primitive type

A reference variable, say `obj` of type `Object`, can store or maintain the reference of any object, as all classes in the Java programming language are subclasses of `Object`. For example

```
Object obj = new ClassName();
```

creates an object and the reference of the created object is assigned to the variable `obj`. However, it is not possible to assign a primitive value to the variable `obj`. That is, the following statement will cause an error:

```
Object obj = 1;
```

To enable the reference variable `obj` to refer to a primitive value, a possible way is to write a class that simply encapsulates a primitive value and provides a method for getting the value. For illustration, a class `MyInteger` is shown in Figure 10.36.

```
// Definition of class MyInteger
public class MyInteger {
    // The primitive (int) value stored
    private int value;

    // The constructor that stores the supplied integer
    public MyInteger(int value) {
        this.value = value;
    }

    // Get the stored integer
    public int intValue() {
        return value;
    }
}
```

Figure 10.36 MyInteger.java

With the class `MyInteger`, you can now create a `MyInteger` object to encapsulate a primitive value of type `int`, such as:

```
Object obj = new MyInteger(1);
```

The variable `obj` cannot maintain a primitive value of type `int` but it can maintain the reference of any object, which is a `MyInteger` object in this case. The scenario can be visualized in Figure 10.37.

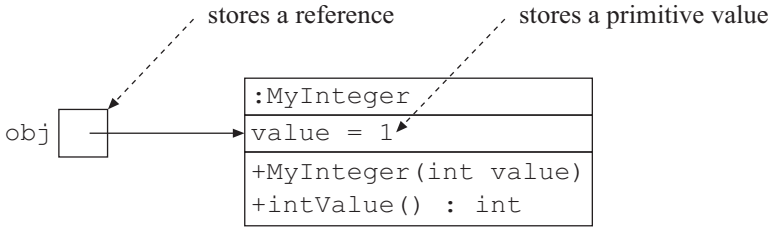


Figure 10.37 A reference variable referring to a `MyInteger` object that maintains a primitive value

As it is sometimes necessary to encapsulate primitive values, the Java standard software library provides classes for all primitive types, and each object encapsulates a single primitive value using the above-mentioned idea. Such classes are known as wrapper classes. Please use the following the reading to learn about using wrapper classes.

Reading

King, Section 13.3 ‘Wrapper classes’, pp. 562–64

From the reading, you know that there are eight wrapper classes for the eight primitive types. For most of the classes, except primitive `char` and `int`, the wrapper class names are the same names of the primitive type with a capital letter first. For `char` and `int`, the corresponding wrapper classes are `Character` and `Integer` respectively.

The core purpose of wrapper classes is to encapsulate primitive values so that these values can be maintained by a reference variable. The implication is that collection classes (discussed earlier in this unit) can maintain primitive values. A sample program that illustrates this usage is discussed in the next section.

Most wrapper classes define more than one constructor, and each must define a constructor that looks like:

```
public WrapperClassName(PrimitiveType value)
```

For example, a possible way to create an `Integer` object with a value of type `int` is:

```
Integer intObject = new Integer(10);
```

Wrapper classes not only enable you to encapsulate a primitive value, they define many useful utility methods for various purposes, such as enabling the conversions between primitive values and textual representations. Appendix E of the unit provides you with further details on performing conversion with wrapper class objects.

Wrapper classes are final

All wrapper classes define no methods for updating the encapsulated primitive value. Therefore, all wrapper class objects are considered immutable objects, and their encapsulated value can be read only after a wrapper class object is created. You may consider creating subclasses of the wrapper classes so that setter methods are defined to update the encapsulated value. However, all wrapper classes are `final` and it is therefore impossible to extend the classes by inheritance.

Instead of updating the encapsulated value, you can only have a wrapper class object with a new primitive value by creating a new wrapper class object by supplying a value based on the existing wrapper class object.

Self-test 10.5

1 The `Boolean` wrapper class does not define a `parseBoolean()` method that accepts a `String` object. Suggest a way to obtain a primitive value of type `boolean` from a `String` object in addition to the approach of creating a `Boolean` object explicitly. For example, if the content of a `String` object is `"true"`, the obtained `boolean` value is `true`. Otherwise, a `boolean` value of type `false` is obtained.

2 Two variables `intObj1` and `intObj2` are referring to two `Integer` objects that encapsulate values `1` and `2` of type `int`. Suggest the required statements to obtain the sum of the encapsulated values and the `String` concatenation of the two objects, so that the results are `3` and `"12"` respectively.

Conversion with wrapper class utility methods

If you read the API documentation of the wrapper classes, you will find that the six wrapper classes for numeric primitive types — `byte`, `double`, `float`, `int`, `long` and `short` — are subclasses of the abstract class `java.lang.Number` (or `Number`, for short). The class hierarchy of the wrapper classes is shown in Figure 10.38.

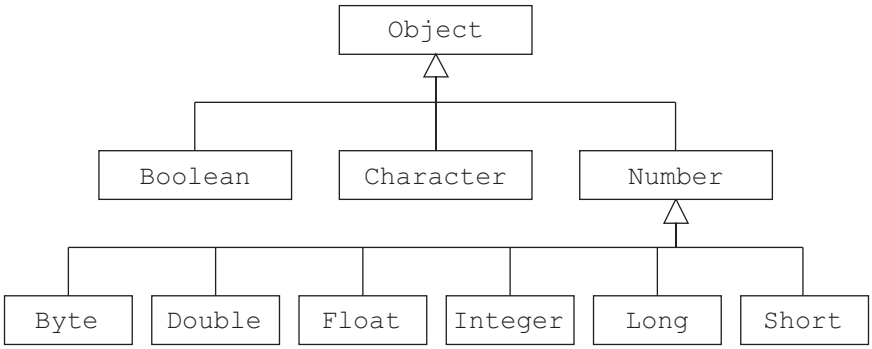


Figure 10.38 Class hierarchy of all wrapper classes

The `Number` class defines methods, which are:

```
public byte byteValue()
public abstract double doubleValue()
public abstract float floatValue()
public abstract int intValue()
public abstract long longValue()
public short shortValue()
```

Even though four out of six methods are abstract, all subclasses of the `Number` class — `Byte`, `Double`, `Float`, `Integer`, `Long`, and `Short` — define all six methods for returning an equivalent value of the suitable primitive type. You should notice that rounding and truncation might be applied for getting the result to be returned.

To illustrate how to use collection classes for maintaining primitive values and the different values returned by the six methods defined by the `Number` class, a class `PrimitiveList1` is written as shown in Figure 10.39.

```
// Resolve ArrayList class
import java.util.ArrayList;

// Definition of class PrimitiveList1
public class PrimitiveList1 extends ArrayList {
    // The string for line delimiter
    private static final String HORIZONTAL_LINE =
        "=========================================";

    // Return the textual representation of the collection of object
    public String toString() {
        // The variable referring the String object to be return
        String result = "";

        // For each element in the list, append it to the String
        // object to be returned
        for (int i = 0; i < size(); i++) {
            Object element = get(i);
            result += "Item " + i + " = " + element + "\n";

            // Check if the element is a Number object
            // (object of any subclass of Number class)
            if (element instanceof Number) {
                // Append the equivalent values of different numeric
                // primitive types
                Number number = (Number) element;
                result += "byte = " + number.byteValue() + "\n";
                result += "double = " + number.doubleValue() + "\n";
                result += "float = " + number.floatValue() + "\n";
                result += "int = " + number.intValue() + "\n";
                result += "long = " + number.longValue() + "\n";
                result += "short = " + number.shortValue() + "\n";
            }

            // Add a line delimiter
            result += HORIZONTAL_LINE + "\n";
        }

        // Return the resultant textual representation
        return result;
    }
}
```

Figure 10.39 PrimitiveList1.java

To test the `PrimitiveList1` class, a driver program `TestPrimitiveList1` is written as shown in Figure 10.40.

```
// Definition of class TestPrimitiveList1
public class TestPrimitiveList1 {

    // Main executive method
    public static void main(String args[]) {
        // Create a PrimitiveList1 object
        PrimitiveList1 primitiveList = new PrimitiveList1();

        // Add primitive values to the PrimitiveList object
        // by wrapper class objects
        primitiveList.add(new Boolean(true));
        primitiveList.add(new Byte((byte) 65));
        primitiveList.add(new Character('A'));
        primitiveList.add(new Double(3.1415));
        primitiveList.add(new Float(1000.5F));
        primitiveList.add(new Integer(100));
        primitiveList.add(new Long(10000L));
        primitiveList.add(new Short((short) 50));

        // Show the textual representation of the
        // PrimitiveList1 object
        System.out.println(primitiveList);
    }
}
```

Figure 10.40 TestPrimitiveList1.java

Compile the classes and execute the `TestPrimitiveList1` program. The following output will be shown on the screen:

```
Item 0 = true
==========================================
Item 1 = 65
byte = 65
double = 65.0
float = 65.0
int = 65
long = 65
short = 65
==========================================
Item 2 = A
==========================================
Item 3 = 3.1415
byte = 3
double = 3.1415
float = 3.1415
int = 3
long = 3
short = 3
==========================================
Item 4 = 1000.5
byte = -24
```

```
double = 1000.5
float = 1000.5
int = 1000
long = 1000
short = 1000
==========================================
Item 5 = 100
byte = 100
double = 100.0
float = 100.0
int = 100
long = 100
short = 100
==========================================
Item 6 = 10000
byte = 16
double = 10000.0
float = 10000.0
int = 10000
long = 10000
short = 10000
==========================================
Item 7 = 50
byte = 50
double = 50.0
float = 50.0
int = 50
long = 50
short = 50
==========================================
```

With the use of the six methods defined by the `Number` class, you can perform conversion operations that are similar to casting. For example, the following two statements are practically equivalent:

```
int i = (int) 3.14;
int i = new Double(3.14).intValue();
```

The first statement simply uses casting for the conversion, and the second one first creates a `Double` object that encapsulates the value `3.14` and the `intValue()` method is used to get the converted value of type `int`. The first statement is mostly used because it is simpler and involves no object creation.

Wrapper classes not only provide conversion among numeric types but also support conversion between primitive values and textual representation. A primitive value, say a value `123456` of type `int`, is represented internally using four bytes, but the textual representation needs six characters to represent it. To convert a primitive value to its corresponding textual representation, the `toString()` method of the corresponding wrapper class is used. For example, to convert the primitive value `123456` of type `int` to its textual representation, the following statement can be used:

```
String intStr = new Integer(123456).toString();
```

Therefore, the statement for converting a primitive value to a `String` object is:

```
new WrapperClass(primitiveValue).toString()
```

As well as the above method, an even simpler way to get the textual representation of a primitive value is to use `String` concatenation. In *Unit 7*, we said that if either side of the `+` operator is a `String` object, a `String` concatenation operation will be carried out. For the non-`String` operand, its `toString()` method will be executed implicitly for getting a textual representation for the concatenation operation. If the non-`String` operand is a primitive value, the `toString()` method of the corresponding wrapper class will be executed instead. Therefore, the following shorthand has been used in previous units for converting primitive values to get their corresponding `String` objects:

```
"" + 123456
```

As the right-hand side operand is non-`String` and is a primitive value, the `toString()` method of the corresponding wrapper class, `Integer`, will be called for `String` concatenation. Therefore, the above statement is equivalent to

```
"" + new Integer(123456).toString()
```

and the result of the above `String` concatenation is therefore `"123456"`. As a result, the general format of getting an equivalent `String` object of a primitive value is:

```
"" + primitiveValue
```

For a conversion the other way round — to get a primitive value from a `String` object — we can use the wrapper classes as well. According to the API documentation of the wrapper classes, all classes except the `Character` class define an overloaded constructor that accepts a `String` object for initializing the value to be encapsulated. For example, the `Integer` class defines an overloaded constructor to be:

```
public Integer(String s)
```

It is therefore possible to create an `Integer` object by the following statement:

```
String intStr = "123456";
Integer intObject = new Integer(intStr);
```

Once an `Integer` object is available, it is possible to call its `intValue()` to obtain the primitive value it encapsulates, such as:

```
int value = intObject.intValue();
```

Then, the value `123456` is assigned to the variable `value`. By combining the above statement, the `String` object `"123456"` can be converted to its equivalent primitive by the following statement:

```
new Integer("123456").intValue();
```

You should notice that the constructors of the six numeric wrapper classes that accept a `String` object as supplementary data cause a `NumberFormatException` if the supplied `String` object does not contain a proper number. Therefore, the above statement is usually enclosed in a `try/catch` construct that handles the `NumberFormatException` exception. A typical statement block for getting a primitive value from a `String` object is therefore:

PrimitiveType `var = ...;  // initialize the variable var`
```
try {
    var = new WrapperClassName(StringExpression).PrimitiveTypeValue();
} catch (NumberFormatException e) {
    ... // perform remedial action if the String object does not
        // contain a proper value
}
```

For example, to obtain a primitive value of type `int` from a `String` object, the following statement block can be used:

```
String intStr = ...;
int value = 0;
try {
    value = new Integer(intStr).intValue();
} catch (NumberFormatException e) {
    System.out.println("The variable intStr does not contain "
        "a proper number of type int");
}
```

If the `String` object referred by the variable `intStr` contains a proper number of type `int`, an equivalent value of type `int` is assigned to the variable `value`. Otherwise, the `catch` block of the above program block will be executed and the value of the variable `value` will be kept unchanged at `0`.

The above approach is also applicable to the `boolean` primitive type, and the `Boolean` wrapper class defines an overloaded constructor that accepts a `String` object. If the contents of the `String` object are non-`null` and are case-insensitively equal to `"true"`, the created `Boolean` object encapsulates a `boolean` value `true`. Otherwise, no exception will be thrown and the `Boolean` object encapsulates a `boolean` value `false`. Therefore, after executing the following two statements,

```
boolean b1 = new Boolean("TRUE").booleanValue();
boolean b2 = new Boolean("YES").booleanValue();
```

the contents of the variable `b1` and `b2` are `true` and `false` respectively.

Another way to obtain a numeric primitive value from a `String` object is to use the class method `parse`*PrimitiveType*`()` of the six numeric wrapper classes. For example, we have been using the following statement to get a primitive value of type `int` from a `String` object

```
Integer.parseInt(StringExpression)
```

such as:

```
String intStr = ...;
int value = Integer.parseInt(intStr);
```

Like the former approach that creates a wrapper class object with a supplied `String` object, the `parse`*PrimitiveValue*`()` method causes a `NumberFormatException` exception if the supplied `String` object does not contain a proper number. Therefore, a suitable `try/catch` construct has to be used as well, such as:

```
String intStr = ...;
int value = 0;
try {
    value = Integer.parseInt(intStr);
} catch (NumberFormatException e) {
    System.out.println("The variable intStr does not contain "
        "a proper number of type int");
}
```

Basically, the two above-mentioned approaches are practically equivalent, but the latter is shorter and is therefore used more frequently.

Appendix F: An enhanced implementation of `PrimitiveList1`

The `toString()` method of the `PrimitiveList` class we discussed and showed in Figure 10.39 used `String` concatenations to construct the final `String` object to be returned. As we said that `String` concatenation could be a substantial overhead, it is preferable to use a `StringBuffer` object instead. To illustrate such an enhancement, `PrimitiveList2` is written and shown in Figure 10.41.

```
// Resolve ArrayList class
import java.util.ArrayList;

// Definition of class PrimitiveList2
public class PrimitiveList2 extends ArrayList {
    // The string for line delimiter
    private static final String HORIZONTAL_LINE =
        "=========================================";

    // Return the textual representation of the collection of object
    public String toString() {
        // The variable referring the StringBuffer object for obtaining
        // the String object to be returned
        StringBuffer result = new StringBuffer();

        // For each element in the list, append it to the StringBuffer
        // object to be returned
        for (int i = 0; i < size(); i++) {
            Object element = get(i);
            result.append("Item ").append(i).append(" = ").
                append(element).append("\n");

            // Check if the element is a Number object
            // (object of any subclass of Number class)
            if (element instanceof Number) {
                // Append the equivalent values of different numeric
                // primitive types
                Number number = (Number) element;
                result.append("byte = ").
                    append(number.byteValue()).append("\n");
                result.append("double = ").
                    append(number.doubleValue()).append("\n");
                result.append("float = ").
                    append(number.floatValue()).append("\n");
                result.append("int = ").
                    append(number.intValue()).append("\n");
                result.append("long = ").
                    append(number.longValue()).append("\n");
                result.append("short = ").
                    append(number.shortValue()).append("\n");
            }
```

```
            // Add a line delimiter
            result.append(HORIZONTAL_LINE).append("\n");
        }

        // Return the resultant textual representation
        return result.toString();
    }
}
```

Figure 10. 41 PrimitiveList2.java

The driver program, `TestPrimitiveList2` class, can be found on the course CD-ROM and the course website. Compile the classes and execute the `TestPrimitiveList2` program. You will get the same output as in the `TestPrimitiveList1` program.

After you have experimented with both the `TestPrimitiveList1` and `TestPrimitiveList2` programs, you will find that they take roughly the same time to complete the executions. However, if the `toString()` methods of the `PrimitiveList1` or `PrimitiveList2` are called repeatedly, the efficiency of the `PrimitiveList2` becomes more noticeable.

Programs using `String` concatenations are shorter, more natural and easier to read at the expense of possibly introducing significant performance problems. String concatenations using `StringBuffer` object are much faster, but the programs are less readable. If you are the one to make the decision to choose either `String` or `StringBuffer`, you can use concatenation with `String` objects in program segments that are not executed repeatedly. Otherwise, you should use `StringBuffer` so that you do not have to search and modify the program segments in the future.

Suggested answers to self-test questions

Self-test 10.1

1 As the number of days for a particular month of a year should be accessible by the month, such as 0 for January and 11 for December, both an array object and a list data structure are preferable. Practically, as the number of months in a year is fixed, an array object is a better choice than a list data structure, because using an array object is simpler than using a list data structure.

2 Student data for a class are usually sorted alphabetically and are accessible by the class number or student number. Therefore, a list data or array object is preferable. However, as the number of students for different classes and the number of students for a class may vary in an academic year, a list data structure is preferable, as the size of an array object in the Java programming language cannot be changed.

3 The preferable way to maintain a collection of website addresses depends on the way you handle those address. If you are going to sort the addresses by the website name or the hyperlink address so that you can access a particular Web address more easily, it is preferable to maintain the addresses by a list data structure. Whenever a new address is added to the list data structure, it is inserted in an appropriate position in the list so that the order of the list is maintained. However, if you are going to categorize the website addresses according to their nature, the addresses will be arranged in a hierarchical style, such as “Computer > Software > Programming language > Java” and “Computer > Hardware > Processors”. Then, it is preferable to maintain the Web page addresses in a tree data structure.

Self-test 10.2

1 The following table summarizes the differences between an array object and a `List` object.

Aspect	A `List` object	An array object
Element type	The storage of a `List` object can maintain references of objects and can therefore maintain any objects in the Java programming language. Primitive types can also be stored and Java automates the conversion from a primitive value (e.g. an integer 3) to an object containing the value (e.g. an `Integer` object containing 3). The automation also exists in the opposite direction.	The type of array element can be primitive or non-primitive, and an array object can therefore natively maintain a collection of primitive values. It is possible to restrict the type of elements to be maintained by an array object. For example, an array object with element type of class `Date` can only store the reference referring to `Date` object.
Efficiency	The time to access a particular maintained object depends on the implementation class being used. It may take longer to access a particular element maintained by a list object.	All elements maintained by an array object can be accessed in any order by using the array subscript or index, and the time to access any element is the same.
Extensibility	The capacity of a `List` object can extend automatically.	The size of an array object is determined at the time it is created. There is no way to resize an array object, and the workaround is to create another array object with a larger size and copy the contents from the original array object to the new one.

2 SortParameters.java

```
// Resolve classes in java.util package
import java.util.*;

// Definition of SortParameters
public class SortParameters {

    // Main executive method
    public static void main(String args[]) {
        // Create a List object
        List <String> paramList = new ArrayList<String> ();

        // Add the program parameters to the List object
        for (int i=0; i < args.length; i++) {
            paramList.add(args[i]);
        }

        // Sort the objects maintained by the List object
```

```
        Collections.sort(paramList);

        // Show the objects maintained by the List object
        for (int i=0; i < paramList.size(); i++) {
            System.out.println(i + " : " + paramList.get(i));
        }
    }
}
```

Self-test 10.3

1 **TestArrayList.java**

```
// Resolve classes in java.util package
import java.util.*;

// Definition of class TestArrayList
public class TestArrayList {

    // Main executive method
    public static void main(String[] args) {
        // Check program parameter
        if (args.length < 1) {
            // Show usage
            System.out.println("Usage: java TestArrayList <name>");
        } else {
            // Prepare a List object (a ArrayList object)
            List<Object> objList = new ArrayList<Object>();

            // Add element to the List object
            objList.add(0, "Hello ");
            objList.add(1, args[0]);
            objList.add(2, ". It is now ");
            objList.add(3, new Date());
            objList.add(4, ".");

            // Show the objects maintained by the List one by one
            Iterator iterator = objList.iterator();
            while (iterator.hasNext()) {
                System.out.print(iterator.next());
            }
            System.out.println();
        }
    }
}
```

2 TestHashMap.java

```
// Resolve classes in java.util package
import java.util.*;

// Definition of class TestTreeMap
public class TestHashMap {

    // Main executive method
    public static void main(String[] args) {
        // Check program parameters
        if (args.length < 1) {
            // Show usage message
            System.out.println("Usage: java TestTreeMap <region>");
        } else {
            // Create the mapping between the region codes and the
            // region full names
            Map<String, String> mapping = new HashMap<String, String>();
            mapping.put("HK", "Hong Kong");
            mapping.put("KLN", "Kowloon");
            mapping.put("NT", "New Territories");

            // Show the contents of the Map objects
            Set<String> keys = mapping.keySet();
            Iterator iterator = keys.iterator();
            int i = 0;
            while (iterator.hasNext()) {
                i++;
                Object key = iterator.next();
                System.out.println(i + " : " + key +
                    " - " + mapping.get(key));
            }

            // Get the region full name from the Map object
            String region = (String) mapping.get(args[0]);
            // Show the result
            if (region == null) {
                System.out.println(
                    "The supplied region code is invalid");
            } else {
                System.out.println(
                    "The region code ("
                        + args[0]
                        + ") is referring to \""
                        + region + "\".");
            }
        }
    }
}
```

Compile and execute the `TestHashMap` program. You will find that the output of the program is as follows:

```
1 : KLN – Kowloon
2 : NT - New Territories
3 : HK - Hong Kong
```

The reason is that the `Map` object uses a `Set` object to maintain the key objects and a `Set` object does not maintain the order among the key objects. Therefore, the order of the key objects visited by the `Iterator` object is not necessarily the same as the order of the key/value object pairs added to the `Map` object.

Self-test 10.4

1 It makes sense to say 'a queue is a list' because a queue can be viewed as a specific type of list in that objects are appended to the end of the list and objects are removed from the beginning of the list. Based on such an idea, it is possible to implement a queue class, say `LinkedQueue`, which extends the `LinkedList`.

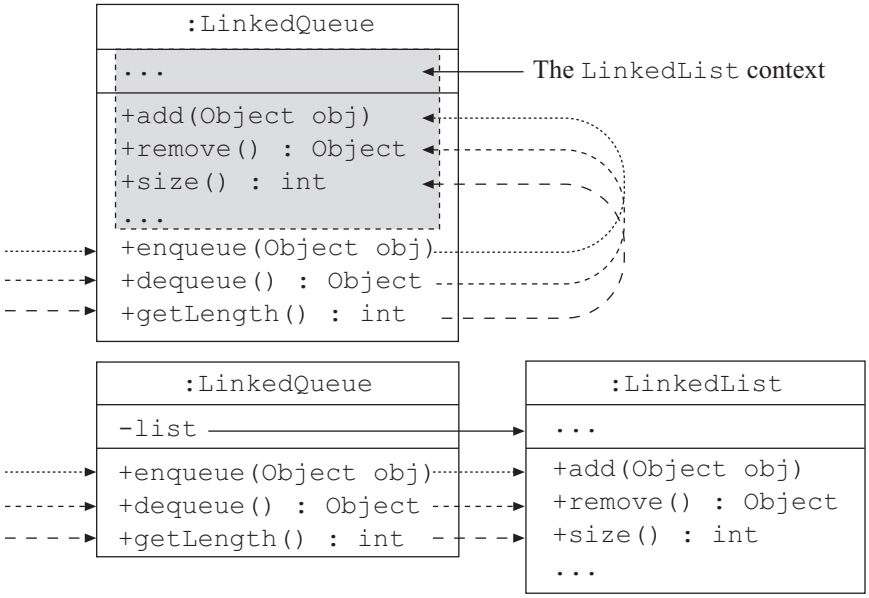
LinkedQueue.java

```
// Resolve classes in java.util package
import java.util.*;

// Definition of class LinkedQueue
public class LinkedQueue extends LinkedList implements Queue {
    // Add an object to the queue
    public void enqueue(Object obj) {
        add(obj);
    }

    // Obtain and remove an object from the queue
    public Object dequeue() {
        return remove(0);
    }

    // Get the length of the queue
    public int getLength() {
        return size();
    }
}
```

However, the methods of a list which violate the valid behaviour of a queue need to be disabled. Examples are `add()` and `remove()` which can be used to add/remove an element at any list position but cannot be used on a queue since it is first-in-first-out and adding/removing an element at an arbitrary position is not allowed. As a result, 'a queue has a list' is a better implementation.

By comparing the definitions of the `LinkedQueue` class presented here and the one shown in Figure 10.16, you can figure out how to implement the `LinkedQueue` using different approaches. A significant difference is that it is not necessary to have the attribute

`list` to refer to a `LinkedList` object, and the method calls are not prefixed with "`list.`". From another perspective, the calls to the methods `add()`, `remove()` and `size()` are actually prefixed with "`this.`". The scenarios of the two implementations of the class `LinkedQueue` can be visualized as the following diagram.



(Dashed arrows of different styles in the diagram indicate the paths of message passing for different methods. The solid arrow indicates the `LinkedList` object is referred by the attribute `list` of the second `LinkedQueue` object.)

In the diagram, the upper `LinkedQueue` object in the dialog is the implementation used in this self-test. The behaviours of the `LinkedQueue` object are implemented by calling the methods provided by the `LinkedList` context. Therefore, the methods are implicitly prefixed by "`this.`". The lower `LinkedQueue` object is the one created by the `LinkedQueue` class shown in Figure 10.16. It has a `LinkedList` object and all behaviours of the `LinkedQueue` object are delegated to the associated `LinkedList` object.

2 **ArrayQueue.java**

```
// Resolve classes in java.util package
import java.util.*;

// Definition of class ArrayQueue
public class ArrayQueue implements Queue {
    // Attribute
    private List list = new ArrayList();

    // Add an object to the queue
    public void enqueue(Object obj) {
        list.add(obj);
    }
```

```
    // Obtain and remove an object from the queue
    public Object dequeue() {
        return list.remove(0);
    }

    // Get the lenght of the queue
    public int getLength() {
        return list.size();
    }
}
```

To test the performance of the two implementations, `LinkedQueue` and `ArrayQueue`, a driver program, `TestQueue` class, is written below.

TestQueue.java

```
// Definition of class TestQueue
public class TestQueue {

    // Main executive method
    public static void main(String args[]) {
        // Check for program parameter
        if (args.length < 1) {
            // If no program parameter is provided, prompt the user
            System.out.println("Usage: java TestQueue 1|2");
            System.out.println("Where 1 - LinkedQueue, 2 - ArrayQueue");
        }
        else {
            // Prepare the Queue object accordingly
            Queue queue = null;
            if (args[0].equals("1")) {
                queue = new LinkedQueue();
            } else if (args[0].equals("2")) {
                queue = new ArrayQueue();
            }

            if (queue != null) {

                // Add objects to the queue object
                for (int i=0; i < 50000; i++) {
                    queue.enqueue(new Double(Math.random() * 1000.0));
                }
                // Show the number of objects maintained by the Queue
                System.out.println("There are " +
                    queue.getLength() + " objects");

                // Remove the objects from the queue
                for (int i=0; i < 50000; i++) {
                    queue.dequeue();
                }
                // Show the number of objects maintained by the Queue
                System.out.println("There are " +
                    queue.getLength() + " objects");
            }
        }
    }
}
```

The `TestQueue` program adds and removes 50000 elements to a `Queue` object. Execute the program and provide a program parameter `1` or `2`. The implementation `LinkedQueue` and `ArrayQueue` is to be used respectively. You will find that the `LinkedQueue` implementation takes much less time than the `ArrayQueue` does. For the reason behind it, please refer to Appendix A of the unit.

3 **Stack.java**

```
// Definition of interface Stack
public interface Stack {
    // Add an object to the Stack
    public void push(Object obj);
    // Obtain and remove an object from the Stack
    public Object pop();
    // Inquire the number of objects it maintains
    public int getLength();
}
```

LinkedStack.java

```
// Resolve classes in java.util package
import java.util.*;

// Definition of class ArrayStack
public class LinkedStack implements Stack {
    // Attribute
    private List list = new LinkedList();

    // Add an object to the Stack
    public void push(Object obj) {
        list.add(obj);
    }

    // Obtain and remove an object from the Stack
    public Object pop() {
        return list.remove(list.size() - 1);
    }

    // Get the lenght of the Stack
    public int getLength() {
        return list.size();
    }
}
```

ArrayStack.java

```
// Resolve classes in java.util package
import java.util.*;

// Definition of class ArrayStack
public class ArrayStack implements Stack {
    // Attribute
    private List list = new ArrayList();

    // Add an object to the Stack
    public void push(Object obj) {
        list.add(obj);
    }

    // Obtain and remove an object from the Stack
    public Object pop() {
        return list.remove(list.size() - 1);
    }

    // Get the lenght of the Stack
    public int getLength() {
        return list.size();
    }
}
```

TestStack.java

```
// Definition of class TestStack
public class TestStack {

    // Main executive method
    public static void main(String args[]) {
        // Check for program parameter
        if (args.length < 1) {
            // If no program parameter is provided, prompt the user
            System.out.println("Usage: java TestStack 1|2");
            System.out.println("Where 1 - LinkedStack, 2 - ArrayStack");
        }
        else {
            // Prepare the Stack object accordingly
            Stack stack = null;
            if (args[0].equals("1")) {
                stack = new LinkedStack();
            } else if (args[0].equals("2")) {
                stack = new ArrayStack();
            }

            if (stack != null) {
```

```
                // Add objects to the Stack object
                for (int i=0; i < 50000; i++) {
                    stack.push(new Double(Math.random() * 1000.0));
                }
                // Show the number of objects maintained by the Stack
                System.out.println("There are " +
                    stack.getLength() + " objects");

                // Remove the objects from the Stack
                for (int i=0; i < 50000; i++) {
                    stack.pop();
                }
                // Show the number of objects maintained by the Stack
                System.out.println("There are " +
                    stack.getLength() + " objects");
            }
        }
    }
}
```

Self-test 10.5

1 If the String object is referred by a variable `str`, you can use the expression

```
str.equals("true")
```

to obtain a `boolean` value of `true` or `false` accordingly.

2 The expression for obtaining the sum is:

```
intObj1.intValue() + intObj2.intValue()
```

To obtain the `String` object with contents `"12"`, either one of the following expressions can be used:

```
intObj1.toString() + intObj2.toString()
intObj1.toString() + intObj2
intObj1 + intObj2.toString()
"" + intObj1 + intObj2
```

Self-test 10.6

1 a `HelloAWorld`

b `AHelloAWorldA`

c `true`

d `12`

2 The output of the program is:

```
true
true
false
```

The results of the first two expressions are obvious. For the last expression, `str1 == str2`, the reference of the two variables are compared. Therefore, the result depends on whether the two variables are referring to the same `String` object. As the variables `str1` and `str2` are initialized with the expression, `new String("Hello")`, two `String` objects are created and are assigned to the variable `str1` and `str2` respectively. Therefore, the two variables are not referring to the same `String` object and the expression, `str1 == str2`, is therefore `false`.

Throughout the entire execution of the program, three String objects are involved. The String object `"Hello"` exists twice in the definition of the class. As we said, the compiler will only create one `String` object for two or more identical `String` objects in class definition. Therefore, an implicit `String` object is created by the compiler. Two other `String` objects are created by the expression, `new String("Hello")`, at runtime. As a result, there are three `String` objects with contents `"Hello"` involved.

Self-test 10.7

1 a `bc`

b `5`

c `ABC 123`

2

Encoder.java

```
// Class definition of Encoder
public class Encoder {
    // Encode the input String object
    public static String encode(String inStr) {
      // The result String
      String outStr = null;

      if (inStr != null) {
         // Initialize the variable to be returned
        outStr = "";

         // Process each character
        for (int i = 0; i < inStr.length(); i++) {
            // Get a character
           char inChar = inStr.charAt(i);

            // Prepare the encoded character
           char outChar = inChar;

            // If the character is a character
           if (Character.isLetter(inChar)) {
               // Encode the character
              switch (inChar) {
                 case 'z' :
                    outChar = 'a';
```

```
                        break;
                    case 'Z' :
                        outChar = 'A';
                        break;
                    default :
                        outChar = (char) (inChar + 1);
                }
            }

             // Append the encoded character to the result String
            outStr += outChar;
        }
    }

    // Return the result String
    return outStr;
  }
}
```