MT201

Unit 5

Arrays

Course team

Developer: Herbert Shiu, Consultant

Designer: Dr Rex G Sharman, OUHK

Coordinator: Kelvin Lee, OUHK

Members: Dr Reggie Kwan, OUHK

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

In *Unit 3*, you started learning how to write class definitions in the Java programming language. A class definition specifies attributes and behaviours. To enable the objects to behave differently in different situations, we discussed the ways to specify conditions, branching statements and looping statements in *Unit 4*.

You can write many useful programs to solve various problems based on such programming techniques. However, sooner or later, you will find that your programs usually have to manipulate a lot of data of the same type. For example:

1. a personal financial system that keeps all your earnings and expenditures in each month of a year
2. phonebook software that keeps the information of your friends
3. a student record system that keeps the marks of all students in a subject.

In the above situations, we can use an array to consolidate a group of data or objects of the same type as a single entity. The data or objects that are consolidated by an array are the *elements* of the array. You can consider the elements as being numbered so that you can access any one of them by providing its number (or index). It is possible to access the elements in the sequence of the numbers or without any particular order, depending on the requirements of the software.

Even though an array can help you group all your friends' information in a phonebook software application, how can you find all friends whose birthdays are in March? This is a typical searching problem. In this unit, we discuss the ways to find an element that fits some criteria in an array.

Objectives

At the end of this unit, you should be able to:

1 *Describe* why arrays are needed.

2 *Apply* simple arrays of primitive types in Java programs.

3 *Apply* simple arrays of non-primitive types in Java programs.

4 *Apply* searching algorithms on arrays.

Why are arrays needed?

The data to be processed in many problems are of the same type(s). For example, the savings account information of each bank customer of a particular bank is similar, and we can use a savings account object to store information for each customer. If a different variable is declared to store the information of a customer, it is difficult to manage and process the information when the number of customers is large. We may end up with a large number of variables `adaSavAC`, `benSavAC`, `johnSavAC`, each with a different name referring to a different savings account object.

As another example, instead of using the `switch/case` structure that was discussed in *Unit 4*, you can declare 12 variables with different variable names to store the number of days in the months. An example is shown in Figure 5.1.

jan	feb	mar	apr	may	jun	jul	aug	sep	oct	nov	dec
31	28	31	30	31	30	31	31	30	31	30	31

Figure 5.1 Storing 12 values by 12 variables

However, using 12 variables is not only tedious, but it is difficult to write programs that use these 12 variables for processing. For example, if you want to verify whether a day is a valid day in the month, you can do so with the following program segment:

```
boolean valid = false;
if (month == 1) { valid = (day <= jan); }
if (month == 2) { valid = (day <= feb); }
if (month == 3) { valid = (day <= mar); }
if (month == 4) { valid = (day <= apr); }
if (month == 5) { valid = (day <= may); }
if (month == 6) { valid = (day <= jun); }
if (month == 7) { valid = (day <= jun); }
if (month == 8) { valid = (day <= aug); }
if (month == 9) { valid = (day <= sep); }
if (month == 10) { valid = (day <= oct); }
if (month == 11) { valid = (day <= nov); }
if (month == 12) { valid = (day <= dec); }
```

After executing the above program segment, the variable `valid` stores a `boolean` value that specifies whether it is a valid day in the month or not. It is tedious and error-prone to use 12 variables to represent the data. Could you pinpoint the error in the above program segment that might lead to a wrong answer?[1]

Is there a handy way to keep the number of days in the month of a year but that is simple to use? A possible way is to create a class, say `DaysOfMonthKeeper`, with methods `setDay()` and `getDay()`.

---

1 The variable to be used for a variable `month` with the value 7 should be `jul` instead of `jun`.

The `setDay()` method can be used to set the number of days in a month; the `getDay()` method returns the number of days in the month according the value of the parameter. That is:

DaysOfMonthKeeper
getDay(month : int) : int setDay(month : int, day : int)

Then, we can greatly simplify the above program segment to become:

```
DaysOfMonthKeeper keeper = new DaysOfMonthKeeper();
boolean valid = (day <= keeper.getDay(month));
```

It is now the programmer of the class `DaysOfMonthKeeper` who determines how to store the 12 values. A `DayOfMonthKeeper` object may have 12 attributes for the number of days in the months. Whenever the `getDay()` method of the `DayOfMonthKeeper` object is called, the number of days of the specified month is returned.

You can see that if we can have an object that retrieves and assigns a value (or data) according to a number, it can greatly simplify the way to write programs that need to access a collection of data, especially those of the same type that can be numbered.

It is tedious to write the class `DaysOfMonthKeeper` to keep a collection of 12 numbers, just as it is a nightmare to write another class to keep a collection of student names in a class.

Fortunately, the Java programming language supports the above features using *array*; it is actually not necessary to write the class `DaysOfMonthKeeper`. For example, the previous program segment that determined whether the value of variable day is a valid day in the month can be enhanced to be:

```
int[] days = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
boolean valid = (day <= days[month - 1]);
```

The above program segment, which uses an array, is equivalent to the previous one. The first statement declares a variable `days` and an array object that contains the number of days in the months of a year. The second statement declares the variable `valid`, and the expression that involves the array determines whether the value of the variable `day` is a valid day in the month. (We will discuss how the values are numbered starting from zero and how it therefore has to perform the calculation `month – 1` before getting the value.) The program is much shorter and simpler. In the following sections in this unit, we discuss in detail the ways to use arrays in the Java programming language.

Creating and using arrays

An array enables you to access a collection of data or objects of the same type with a single variable. In the previous section, we looked at an example in which a group of data of the same type (the number of days in 12 months) can be accessed by the variable `days`.

Please use the following reading to learn how to use arrays in the Java programming language. Afterwards, we elaborate on the ideas presented in the reading.

Reading

King, Section 5.1, pp. 182–87

The following list highlights some concepts involved in the use of arrays:

1 An array is implemented as an object in the Java programming language.

2 The data stored in an array are known as the *elements* of the array, or simply array elements.

3 All elements of an array must be of the same type.

The way to use arrays in the Java programming language usually involves the following steps:

1 declaring a variable that can refer to an array object

2 creating the array object and assigning its reference to the variable

3 accessing the elements in the array using the variable and specifying a subscript; furthermore, it is common to pass the reference of the array object as supplementary data of a method call (via method parameter) so that the entire array object can be accessed as a whole.

The following sections discuss the above steps in detail so that you can have a better understanding of how things work.

Declaring array variables

From the reading, you know that it is necessary to use a variable to refer to an array object in the program. We have said that the format of declaring a variable in the Java programming language is:

Type Variable-name;

For example, to declare a variable named count with type `int` is:

```
int count;
```

You can use a similar format to declare the type of an array. The question here is how to represent the type of an array. Before we proceed to the discussion of such variable declarations, you have to thoroughly understand the principles of arrays in the Java programming language.

Arrays are implemented as objects in the Java programming language. That is, during execution of a software application written in the Java programming language, arrays are objects in the JVM memory that occupy memory spaces and have their attributes and behaviours. Such an object is known as an array object, and the data it can store are the array elements.

If there is an array object, we have to use a variable that stores its reference so that it is possible to access its attributes and behaviours. Such a variable is known as an array variable.

Figure 5.2 visualizes the scenario:

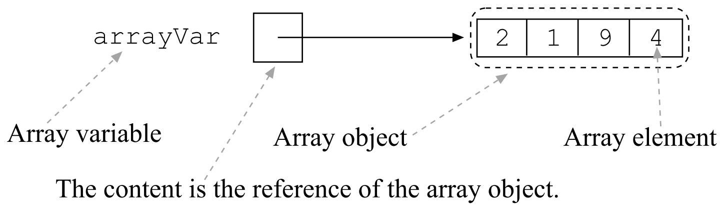


Figure 5.2 A typical scenario of using an array

The above diagram shows an array variable, `arrayVar`, and an array object of four array elements. The array object is surrounded by dashes. The contents of the array elements are `2`, `1`, `9` and `4` respectively.

In the Java programming language, the array element type determines the type of the array. For example, the above array object can store data of type `int`, or in other words it is an array object with array elements of type `int`. The way to specify the type is:

```
int[]
```

The pair of square brackets denotes that it is an array, and the type `int` represents that the array element type is `int`. The general format to represent the type of array is:

Type`[]`

where the *Type* in the above format is the type of the array element. As a result, to declare an array variable, we use the following format:

Type`[]` *Variable-name;*

For example, to declare a variable named `days` that can refer to an array with element type `int`, you can use the following declaration statement:

```
int[] days;
```

Then, the type of the variable is an array of `int` and the element type of the array object that can be referred to by such a variable is `int`.

The following declaration statement is another example of declaring an array variable:

```
String[] names;
```

The above statement declares a variable named `names` and its type is array of `String`, which means such a variable can refer to an array object with array elements of type `String`.

A single declaration statement can declare more than one variable of the same type. For example, the following statement

```
int i, j;
```

declares two variables named `i` and `j` and both are of type `int`. Similarly, the Java programming language allows you to declare two variables of the same array type, such as

```
double[] incomes, expenditures;
```

declares two variables `incomes` and `expenditures`. Both variables are of the same array type, `double`.

The Java programming language allows you to use another format to declare array variables. The format is:

Type Variable-name`[];`

For example:

```
int days[];
String names[];
```

The pair of square brackets can be placed after the variable name instead of immediately following the array element type. You can choose the one you prefer and stick to it while you are writing programs in the Java programming language. In other words, make it your personal convention.

If you declare more than one variable in the same declaration statement, the location of the square brackets does matter. For example, the following statement

```
int[] a, b;
```

declares two variables `a` and `b` of type array of `int`. However, the following declaration statement

```
int a[], b;
```

declares two variables `a` and `b`, but the type of variable `a` is array of `int` and the type of variable `b` is simply `int`. Using the convention in the textbook, you are not recommended to mix variables of array type with those of non-array type in a single declaration statement.

Creating array objects

Array variables are of non-primitive types, and they can only refer to an array object. When an array variable is declared, no array object is immediately available. Therefore, it is necessary to create the actual array object. You learned in *Unit 3* that the keyword new is used to create an object in the Java programming language. For example, to create a `TicketCounter` object, you used the following statement:

```
new TicketCounter()
```

The type of object to be created.

The statement part that follows the keyword `new` specifies the type of the object to be created. You use a similar format to create an array object. Before discussing the full format, let's see how to specify the type of an array object:

Type`[`*number of elements*`]`

The *Type* specifies the array element type, and the *number of elements* specifies how many elements the array object maintains (that is, the array size). It must be non-negative[2].

For example, the type `int[12]` specifies an array with array element type `int` and the array object can maintain (or store) 12 elements. As a result, the complete statement to create an array object whose array element type is `int` that can maintain 12 elements is:

```
new int[12]
```

You can imagine that an array object with 12 elements and the element type `int` is created in the memory of the Java virtual machine (JVM), as shown in the broken line in Figure 5.3.

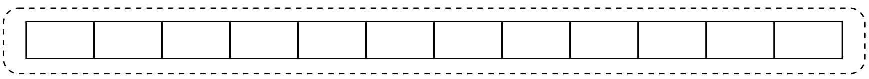


Figure 5.3 An imaginary array object in JVM

[2] Normally the size of an array object to be created is positive. However, it is possible to create an array object of size zero, and it is usually created to fit a program segment requirement indicating that no element has to be processed. Furthermore, an array object of zero size occupies memory in JVM. We will come across a circumstance in which a zero size array is created, when we discuss parameter passing from the command line later in this unit.

You also learned in *Unit 3* that the keyword `new` not only creates the object of the specified type but also returns the reference of the newly created object. Therefore, in order to keep the reference of an array object so that you can subsequently access it and its elements, the keyword `new` is usually used with an `=` operator that assigns the reference of the newly created array object to an array variable. For example,

```
int[] days;
days = new int[12];
```

The first statement in the above program segment declares a variable named days of type array of `int`. That is:

The second statement creates an array object with 12 elements; each element is of type `int`. After the above two statements are executed, the scenario is as shown in Figure 5.4.

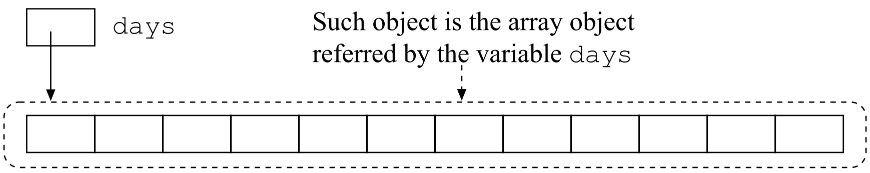


Figure 5.4 An array variable is initialized with an array object

As a form of shorthand, the two statements above can be merged into a single statement:

```
int[] days = new int[12];
```

The above statement clearly shows an important observation. The declaration of an array variable does not need the array size, just the array element type. But creating an array object needs both pieces of information. Therefore, a variable of type array of `int` can refer to array objects with any array size, provided that the element type is `int`. For example, the variable `days` can refer to an array object of 12 elements of type `int` or to an array object of 10 elements of type `int`.

The elements of all newly created array objects are automatically initialized. Table 5.1 shows the initial values of each element for different array element types:

Table 5.1 Initial values of array elements for different element types

Element type	Element initial value
`byte`	`(byte) 0`
`short`	`(short) 0`
`char`	`(char) 0`
`int`	`0`
`long`	`0L`
`float`	`0.0`
`double`	`0.0`
`boolean`	`false`
All non-primitive types	`null`

In a few words, elements of all primitive types other than `boolean` are initialized as zero values, and elements of `boolean` type are initialized as `false`. All non-primitive elements are initialized as `null`.

The scenario after the above-mentioned program segment is executed is shown in Figure 5.5:

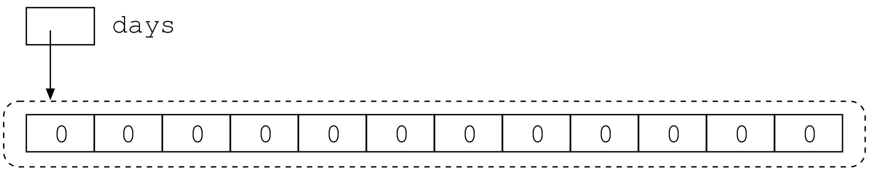


Figure 5.5 An array object of 12 elements with initial values

Accessing elements in arrays

Once you have created an array object and have referred to it using a suitable array variable, it is possible to access its elements. It is necessary to provide a *subscript* (or index) to access a particular array element. The format is:

Array-variable[*subscript*]

In the Java programming language, the subscript in the pair of square brackets can be any expression, and its type is `int`. All array subscripts start with zero. That is, the first element is identified by subscript zero. For example,

```
days[0]
```

specifies the first element of the array object referred to by the variable `days`. You can visualize it in Figure 5.6.

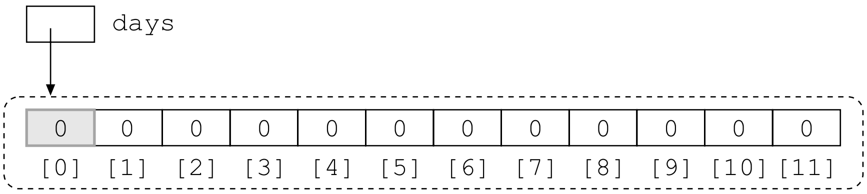


Figure 5.6 The interpretation of the expression days[0]

The notation `[`*subscript*`]` (e.g. `[3]`) used in the above diagram and hereafter denotes the subscript of the array element.

As the expression `days[0]` specifies the first element of the array object and the element type is `int`, you can treat `days[0]` as a usual variable of type `int`. Therefore, you can have the following statements:

```
days[0] = 31;
int jan = days[0];
```

The valid subscripts for an array object with 12 elements range from `0` to `11`. If your program intends to access the 13th element of the array object, it is a runtime error and the program will usually terminate with error messages. The subscript ranges from `0` to `n – 1` for an array object with n elements.

We mentioned that the array subscript is not necessarily a fixed number. It can be any expression with a result of type `int`. A common expression is a single variable. For example,

```
days[i]
```

If the value of the variable `i` is 3, `days[i]` is identical to `days[3]` and that refers to the fourth element of the array object that is referred to by array variable `days`.

If the expression is not simply a variable, it is evaluated first to determine the subscript value; the element of the array object with that subscript is accessed afterwards. For example,

```
days[month – 1]
```

If the value of the variable `month` is `1`, `month – 1` is evaluated first and gives `0`. Then, `days[month – 1]` is resolved to be `days[0]`.

The pair of square brackets `[]` is an operator in the Java programming language and it takes precedence over other operators. For example, in the following statement

```
int totalOfTwoMonths = days[i] + days[i + 1];
```

the two expressions in the pairs of brackets, `i` and (`i + 1`) are evaluated first. If the value of the variable `i` is `5`, the two expressions are evaluated to be `5` and `6` respectively. Then the sixth element (with subscript `5`) and seventh element (with subscript `6`) of the array object referred to by variable days are accessed, and the element values are obtained. Then, the `+` operator executes that gets the sum of `days[5]` and `days[6]`. Finally, the `=` operator is executed to assign the result to the variable `totalOfTwoMonths`.

Now you have learned some basic uses of arrays in the Java programming language—enough for you to implement arrays in real applications. The following is a sample use of array. You will realize that array is a crucial feature of the language.

Example program — a simple stack

A stack is a common tool used by software developers for solving various problems. But first of all, do you know what a stack is?

A stack is like a pile of dishes. If you are getting the dishes one by one, the last dish you place on the pile of dishes is the first dish you get from it. We usually specify such a phenomenon as ‘last-in-first-out’ (LIFO). Therefore, a stack is considered to be an object that can store some data. At any time, if a piece of data is retrieved from the stack, the last element that was stored by the stack object is returned first and is removed from the stack.

In this section, we discuss how to design and implement a stack. For simplicity, the stack we’re going to implement stores numbers of type `int` only. We place one number over another in diagrams, and this entity consisting of numbers is called a stack. Once you have learned how to implement a stack, you can develop stacks for storing data of other data types.

First of all, we have to analyse what the attributes and behaviours of a stack are. Basically, there are only two behaviours, `push` and `pop`. The `push` behaviour places a number on top of the stack, which is possibly empty. For example, if the stack currently stores a number `10` as shown in Figure 5.7:

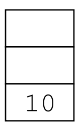


Figure 5.7 Current stack contents

If the stack object executes its behaviour `push` with supplementary data `20`, it places the number `20` on top of the existing number `10`. The scenario is shown in Figure 5.8:

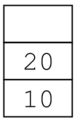


Figure 5.8 Pushing a number 20 to the stack

If it performs its `pop` behaviour now, the topmost number (the latest number that is stored by the stack object) is returned as the feedback and is removed from the stack. Therefore, at that moment, the number `20` is returned. The scenario of the stack object is shown in Figure 5.9:

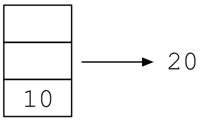


Figure 5.9 Popping a number from the stack

If the stack performs its `pop` behaviour again, the number `10` is returned as the feedback and is removed from the stack. Then, the stack object becomes empty as shown in Figure 5.10.

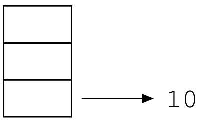


Figure 5.10 Popping another number from the stack

The design of a stack class must include two behaviours — `push` and `pop`. The `push` behaviour needs supplementary data as the number to be stored in the stack; there is no feedback for this behaviour. The `pop` behaviour takes no supplementary data but it returns, as feedback, the last number popped. Therefore, the class, says `IntegerStack`, is preliminarily designed as shown in Figure 5.11.

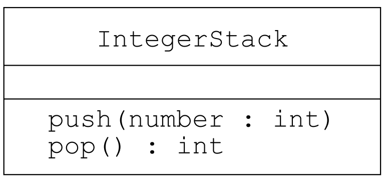


Figure 5.11 The IntegerStack class with methods only

The `IntegerStack` object has to store a sequence of numbers, and an array is the preferable way to do so. Therefore, the `IntegerStack` class should have an array object with element type `int` to store the numbers, and there should be an attribute for referring to the array object. As a result, the design of the `IntegerStack` class is enhanced as shown in Figure 5.12.

IntegerStack
storage : int[]
push(number : int) pop() : int

Figure 5.12 The IntegerStack class with an array variable as an attribute

When an `IntegerStack` object performs the behaviour `push` for the first time, the number is stored in the first element of the array:

```
storage[0] = number;
```

For the second execution of `push`, the number should be stored in the second element of the array. That is:

```
storage[1] = number;
```

At any time, you might need to know the total numbers already stored in the array object, or equivalently the number of used array elements. For such practical reasons, it is preferable to have an attribute, say `top`, to store the number of used array elements and indicate the current subscript of the array object, which is equivalently the top of the stack that can store a newly added number. Therefore, the class design is enhanced as shown in Figure 5.13.

IntegerStack
storage : int[] top : int
push(number : int) pop() : int

Figure 5.13 The complete design of the IntegerStack class

The template of the IntegerStack class definition is:

```
public class IntegerStack {
    private int[] storage;
    private int top;
    public void push(int number) { ...... }
    public int pop() { ...... }
}
```

As an array variable declaration only declares a variable that can refer to an array object, we need to create a *real* array object to store the data when an `IntegerStack` object is created. Therefore, the attribute declaration of `storage` is defined and initialized to refer to an array object. The size of the array object is arbitrarily chosen to be six. The attribute declaration becomes:

```
    private int[] storage = new int[6];
```

Since the stack should be empty when nothing has been pushed into it, the attribute `top` is initialized to 0. After the two changes, the class definition becomes:

```
public class IntegerStack {
    private int[] storage = new int[6];
    private int top = 0;
    public void push(int number) { ...... }
    public int pop() { ...... }
}
```

Notice the value of the attribute `top` is initialized to `0` and we do not rely on Java to give it a default value of `0`. It is bad programming practice to omit the above explicit initialization. Therefore, when you create an `IntegerStack` object, the scenario can be visualized in Figure 5.14.

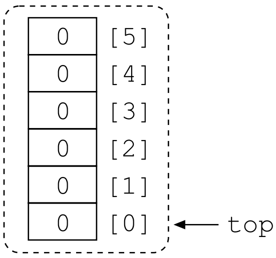


Figure 5.14 The initial scenario of the IntegerStack object

The value of the attribute `top` is zero. You can visualize in the above diagram that the attribute `top` indicates that the array element to be used in the next `push` operation is the array element with subscript `0`. (The array elements are presented in such a way that `storage[5]` is at the top and `storage[0]` is at the bottom to mimic stack operations from top to bottom.)

After the `IntegerStack` object performs its `push` behaviour with supplementary data `10`, it is expected that the scenario will become like the one shown in Figure 5.15.

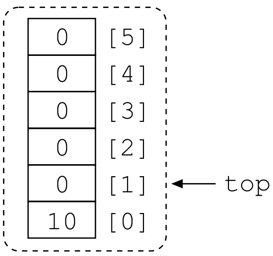


Figure 5.15 The array object after performing the push behaviour with data 10

Therefore, the value of the variable `top` is changed from `0` to `1`. That is, its value is increased by `1`. The corresponding statements for the `push` behaviour are therefore

```
storage[top] = number;  // number is the value to be pushed into stack
top++;
```

and the method declaration is

```
public void push(int number) {
    storage[top] = number;
    top++;
}
```

You can see that the value of the attribute `top` always specifies the subscript of the array element that will store the next new number. As a result, the number to be returned by the `pop` behaviour is `storage[top – 1]`. Furthermore, it is expected that after the `pop` behaviour is executed, the scenario should become like the one shown in Figure 5.16.

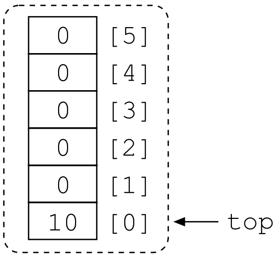


Figure 5.16 The array object after the IntegerStack object performs the pop behaviour

Therefore, the value of the attribute `top` is decreased by `1`. It is practically not necessary to reset the value of the array element, `storage[0]`, to `0` because it will be overridden in the next `push` behaviour.

A possible way to implement the behaviour `pop` is:

```
top--;
return storage[top];
```

The first statement decreases the value of the attribute `top` by one, and the second returns the value of the array element with the value of attribute `top` as subscript.

The complete definition of `pop()` method is therefore

```
public int pop() {
    top--;
    return storage[top];
}
```

As a result, the complete definition of the `IntegerStack1` class is written as shown in Figure 5.17.

```
// Definition of the class IntegerStack (version 1)
public class IntegerStack1 {
    // Attributes
    // The storage for the numbers in the stack using an array with
    // 6 elements
    private int[] storage = new int[6];
    // The attribute for the subscript of the element that can store
    // the newly added number
    private int top = 0;

    // Behaviours

    // The behaviour to push a new number
    public void push(int number) {
        // Show debug message
        System.out.println("DEBUG: Push " + number);

        // Store the number
        storage[top] = number;
        // Increase the subscript for storing the next number
        top++;
    }

    // The behaviour to pop the last number
    public int pop() {
        // Show debug message
        System.out.println("DEBUG: Pop");

        // Decrease the subscript for the last number
        top--;
        // Return the last number
        return storage[top];
    }
}
```

Figure 5.17 IntegerStack1.java

The above class definition is the first implementation version of the stack. We have other implementations in this section.

Now, we can write a driver program that sets up the environment to examine the `IntegerStack1` class. For example, the following class `IntegerStack1` that is defined to test an `IntegerStack1` object is shown in Figure 5.18.

```
// The class definition of TestIntegerStack1 for setting up the
// environment and test the IntegerStack1 object
public class TestIntegerStack1 {

    // Main executive method
    public static void main(String args[]) {
        // create an IntegerStack1 object and use variable stack
        // to refer to it
        // Modify the following line to examine other implementations
        IntegerStack1 stack = new IntegerStack1();

        // Push two numbers to the stack object
        stack.push(10);
        stack.push(20);

        // Pop numbers from the stack and display them
        System.out.println(stack.pop());
        System.out.println(stack.pop());
    }
}
```

Figure 5.18 TestIntegerStack1.java

In this section, we have three driver programs, `TestIntegerStack1`, `TestIntegerStack2` and `TestIntegerStack3`. (The definitions of class `TestIntegerStack2` and `TestIntegerStack3` will be shown very soon.) They are written to examine the first implementation of the stack design, which is the class `IntegerStack1`. If you want to examine the other implementations, you have to modify the driver programs to change the implementation to be used. For example, change the statement to

```
IntegerStack2 stack = new IntegerStack2();
```

to execute the driver program for an `IntegerStack2` object.

For the above `TestIntegerStack1`, compile the two class definitions, `IntegerStack1` and `TestIntegerStack1` and execute the `TestIntegerStack1`. The output of the program is:

```
DEBUG: Push 10
DEBUG: Push 20
DEBUG: Pop
20
DEBUG: Pop
10
```

The outputs of the program confirm that the last number pushed to the `IntegerStack1` object is the first one to be popped from it and displayed.

You can modify the `main()` method of the `TestIntegerStack1` to test the capability of an array object. For example, another `main()` method is defined in the class `TestIntegerStack2` and shown in Figure 5.19.

```
// The class definition of TestIntegerStack2 for setting up the
// environment and test the IntegerStack object
public class TestIntegerStack2 {

    // Main executive method
    public static void main(String args[]) {
        // create an IntegerStack1 object and use variable stack
        // to refer to it
        IntegerStack1 stack = new IntegerStack1();

        // Push seven numbers to the stack object
        stack.push(10);
        stack.push(20);
        stack.push(30);
        stack.push(40);
        stack.push(50);
        stack.push(60);
        stack.push(70);

        // Pop numbers from the stack and display them
        System.out.println(stack.pop());
        System.out.println(stack.pop());
        System.out.println(stack.pop());
        System.out.println(stack.pop());
        System.out.println(stack.pop());
        System.out.println(stack.pop());
        System.out.println(stack.pop());
    }
}
```

Figure 5.19 TestIntegerStack2.java

The `main()` method of the `TestIntegerStack2` class intends to push seven numbers to the `IntegerStack1` object and display them by executing the `pop()` method seven times. You can successfully compile the program because the class definition is properly written. However, when you execute it, you will get the following output:

```
DEBUG: Push 10
DEBUG: Push 20
DEBUG: Push 30
DEBUG: Push 40
DEBUG: Push 50
DEBUG: Push 60
DEBUG: Push 70
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 6
        at IntegerStack1.push(IntegerStack1.java:19)
        at TestIntegerStack2.main(TestIntegerStack2.java:18)
```

The above output indicates that there is a runtime error `ArrayIndexOutOfBoundsException`. Such a runtime error occurs because an array object does not have the element specified by the subscript, which is 6 as shown in the error message. If you study the source file `IntegerStack1.java` at line 19, the statement is:

```
storage[top] = number;
```

When the above statement is executed, the subscript of the array, the value of the variable `top` in this instance, is `6` and the statement intends to access `storage[6]`. As the array object of the `IntegerStack1` object has six members with subscripts from `0` to `5`, the subscript `6` is out of bounds for the array object and hence the runtime error is displayed.

Let's test another `main()` method as defined in the `TestIntegerStack3` class shown in Figure 5.20.

```
// The class definition of TestIntegerStack3 for setting up the
// environment and test the IntegerStack object
public class TestIntegerStack3 {

    // Main executive method
    public static void main(String args[]) {
        // create an IntegerStack object and use variable stack
        // to refer to it
        IntegerStack1 stack = new IntegerStack1();

        // Push two numbers to the stack object
        stack.push(10);
        stack.push(20);

        // Pop numbers from the stack and display them
        System.out.println(stack.pop());
        System.out.println(stack.pop());
        System.out.println(stack.pop());
    }
}
```

Figure 5.20 TestIntegerStack3.java

For the above `main()` method, two numbers are pushed into the `IntegerStack1` object, but it intends to pop three numbers from it. Compile and execute it; you will get the following outputs:

```
DEBUG: Push 10
DEBUG: Push 20
DEBUG: Pop
20
DEBUG: Pop
10
DEBUG: Pop
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: -1
        at IntegerStack1.pop(IntegerStack1.java:32)
        at TestIntegerStack3.main(TestIntegerStack3.java:18)
```

The output indicates that the `IntegerStack1` object can successfully execute the `pop()` method the first two times. With respect to the runtime error message

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: -1
        at IntegerStack1.pop(IntegerStack1.java:32)
        at TestIntegerStack3.main(TestIntegerStack3.java:18)
```

the first line suggests that there is an `ArrayIndexOutOfBoundException` runtime error as it intended to access the array element with subscript `–1`, which is an invalid subscript and hence resulted in the runtime error. The error message in the second line shows the reversed execution sequence that led to the statement where the runtime error occurred.

```
        at IntegerStack1.pop(IntegerStack1.java:32)
```

The above message is the second line of the runtime error message; it shows that the runtime error occurred while the program was executing the `pop()` method of the `IntegerStack1` object and the runtime errors occurred at line 32 of `IntegerStack1.java` source code, which is the following statement:

```
        return storage[top];
```

Then, the third line of the runtime error message

```
        at TestIntegerStack3.main(TestIntegerStack3.java:18)
```

indicates that the `pop()` method of the `IntegerStack1` object is called from the `main()` method of the `TestIntegerStack3` class at line 18 of the `TestIntegerStack3.java` source code. The statement is:

```
        System.out.println(stack.pop());
```

The above two experiments confirmed that the subscript range of an array object with six elements is from `0` to `5` inclusive. If an array element is accessed with a subscript that is out of range, a runtime error will occur.

In order to safeguard an array object being accessed with an invalid subscript, it is preferable to use an `if` or `if/else` statement to verify whether the subscript is valid before accessing the intended array element. For example, you can use the following `if/else` statement in the `push()` method to make sure the subscript is valid before accessing the array element:

```
// The behaviour to push a new number
public void push(int number) {
    // Show debug message
    System.out.println("DEBUG: Push " + number);

    // Verify whether all array elements have been used
    if (top < 6) {
        // Store the number
        storage[top] = number;
        // Increase the subscript for storing the next number
        top++;
    }
    else {
        // Show message to prompt the user
        System.out.println("The stack is full");
    }
}
```

Similarly, you might expect the following `pop()` method can be used to ensure the validity of the subscript:

```
// The behaviour to pop the last number
public int pop() {
    // Show debug message
    System.out.println("DEBUG: Pop");

   // Verify whether the stack is empty
    if (top > 0) {
        // Decrease the subscript for the previous number
        top--;
        // Return the last number
        return storage[top];
    }
    else {
        // Show message to prompt the user
        System.out.println("The stack is empty");
    }
                  }
```

Unfortunately, the above method declaration gives a compilation error because the `pop()` method is declared that it must return a value of type `int`. If the condition `top > 0` is `false`, the `return` statement in the method will not be executed and no value is returned. The compiler can determine such a problem, and you are prompted with a compile time error. Therefore, a proper way to define the `pop()` method is:

```
    // The behaviour to pop the last number
    public int pop() {
        // Show debug message
        System.out.println("DEBUG: Pop");

        // A local variable result is declared to store the value
        // to be returned
        int result = -1;

        // Verify whether the stack is empty
        if (top > 0) {
            // Decrease the subscript for the last number
            top--;
            // Store the last number to local variable result
            result = storage[top];
        }
        else {
            // Show message to prompt the user
            System.out.println("The stack is empty");
        }
        return result;
    }
```

The above definition of the `pop()` method first declares a local variable `result` for storing the potential value to be returned, and it is initialized to be `–1`. If the attribute top is greater than `0`, which means that the array object of the `IntegerStack1` object contains at least one number available, the `if` part of the `if/else` statement will execute and assign the last pushed number to the local variable `result`. At the end of the method, the value of the local variable `result` is returned.

If the condition `top > 0` is `false`, it means that no number is available. Then, the `else` part of the `if/else` statement is executed and the message `"The stack is empty"` is displayed. At the end of the method, the value of the local variable `result`, which was initialized to be `–1`, is returned. Software developers usually use an unusual return value to indicate abnormal situations or operations. In our example, `–1` is chosen to be the value to indicate that all elements of the array that the `IntegerStack1` object had are unused and no valid value is available.

Another advantage of using a local variable, the variable `result` in the mentioned `pop()` method, is that it adheres to the single-entry-single-exit principle, which is a recommended programming style.

Based on the above discussions, Figure 5.21 shows the complete definition of the second implementation of the Stack design, the `IntegerStack2 class`.

```
// Definition of the class IntegerStack (version 2)
public class IntegerStack2 {
    // Attributes
    // The storage for the numbers in the stack using an array with
    // 6 elements
    private int[] storage = new int[6];
    // The attribute for the subscript of the element that can store
    // the newly added number
    private int top = 0;

    // Behaviours

    // The behaviour to push a new number
    public void push(int number) {
        // Show debug message
        System.out.println("DEBUG: Push " + number);

        // Verify whether all array elements have been used
        if (top < 6) {
            // Store the number
            storage[top] = number;
            // Increase the subscript for storing the next number
            top++;
        }
        else {
            // Show message to prompt the user
            System.out.println("The stack is full");
        }
    }

    // The behaviour to pop the last number
    public int pop() {
        // Show debug message
        System.out.println("DEBUG: Pop");

        // A local variable result is declared to store the value
        // to be returned
        int result = -1;

        // Verify whether the stack is empty
        if (top > 0) {
            // Decrease the subscript for the last number
            top--;
            // Store the last number to local variable result
            result = storage[top];
        }
        else {
            // Show message to prompt the user
            System.out.println("The stack is empty");
        }
        return result;
    }
}
```

Figure 5.21 IntegerStack2.java

Based on the second implementation `IntegerStack2`, another `IntegerStack3` class definition with slightly shorter `push()` and `pop()` methods can be written. The complete definition of the third implementation is shown in Figure 5.22.

```
// Definition of the class IntegerStack (version 3)
public class IntegerStack3 {
    // Attributes
    // The storage for the numbers in the stack using an array with
    // 6 elements
    private int[] storage = new int[6];
    // The attribute for the subscript of the element that can store
    // the newly added number
    private int top = 0;

    // Behaviours

    // The behaviour to push a new number
    public void push(int number) {
        // Show debug message
        System.out.println("DEBUG: Push " + number);

        // Verify whether all array elements have been used
        if (top < 6) {
            // Store the number and increase the subscript for
            // storing the next number afterwards
            storage[top++] = number;
        }
        else {
            // Show message to prompt the user
            System.out.println("The stack is full");
        }
    }

    // The behaviour to pop the last number
    public int pop() {
        // Show debug message
        System.out.println("DEBUG: POP");

        // A local variable result is declared to store the value
        // to be returned
        int result = -1;

        // Verify whether the stack is empty
        if (top > 0) {
            // Decrease the subscript for the last number and
            // Store the last number to local variable result afterwards
            result = storage[--top];
        }
        else {
            // Show message to prompt the user
            System.out.println("The stack is empty");
        }
        return result;
    }
}
```

Figure 5.22 IntegerStack3.java

The differences between the `push()` and `pop()` methods defined in the two implementations are shown in Table 5.2.

Table 5.2 The comparison of the implementations IntegerStack2 and IntegerStack3

Second implementation	Third implementation
`storage[top] = number;` `top++;`	`storage[top++] = number;`
`top--;` `result = storage[top];`	`result = storage[--top];`

The implementations of `push()` and `pop()` methods in the second implementation (`IntegerStack2`) are equivalent to the third implementation (`IntegerStack3`). In *Unit 4*, you learned that the increment operator `++` and decrement operator `--` could be used to increase and decrease the value of a variable by one, respectively.

For the class `IntegerStack3`, the `++` operator is placed after the attribute `top`, which means that the increment operation will be executed after the value of the attribute `top` is used as the subscript. Therefore, the single statement in the third implementation of `IntegerStack3` is equivalent to the two statements in the second implementation.

By contrast, the `--` operator is placed before the attribute `top` in the `pop()` method for the third implementation. The interpretation is that the value of the attribute `top` is decreased by one and the result, after decrement, is used as the subscript of the array.

Both the increment operator and decrement operator can be placed before or after a variable of any primitive types other than `boolean`. If these two operators are used alone, the placing of the operators does not matter. That is,

```
i++;
```

is equivalent to

```
++i;
```

If the operator is placed before a variable, it is known as the pre-increment operator or pre-decrement operator. The value of the associated variable is either increased or decreased by one, and the result is used afterwards. However, if the operator is placed after a variable, it is known as post-increment operator or post-decrement operator. In this instance, the value of the variable is used before the value is increased or decreased by one.

Suppose that the value of the variable `i` was originally `5`. Table 5.3 shows the values of variable `i` and `j` in various combinations of `++`/`--` operators with `=` operator, so that you can examine the differences between the operators.

Table 5.3 Comparing the different increment and decrement operators based on an initial value of i = 5

		Values of the variables after executing the statement	
Statement	Equivalent statements	i	j
`j = ++i;`	`i = i+1;` `j = i;`	6	6
`j = i++;`	`j = i;` `i = i+1;`	6	5
`j = --i;`	`i = i-1;` `j = i;`	4	4
`j = i--;`	`j = i;` `i = i-1;`	4	5

The `++` and `--` operators by themselves, such as those used in the increment part of a `for` loop, are a handy way to increase or decrease the value of a numeric variable. However, if they are used as a sub-expression as in the third implementation of the stack design, it may cause trouble for some 'green' programmers. Furthermore, statements that involve such operations as sub-expressions are harder to understand and debug. It is recommended you use them with care and in very simple expressions only.

The second implementation (`IntegerStack2`) and the third implementation (`IntegerStack3`) of the stack design are equivalent. Let's examine the class `IntegerStack3` to see whether it can work properly even under abnormal situations, such as executing either `push()` or `pop()` methods too many times.

Now, we can revisit the `main()` method of `TestIntegerStack2` that results in runtime errors for the `IntegerStack1` implementation. By modifying the statement that creates the stack object, we can examine the third implementation `IntegerStack3`.

```
public static void main(String args[]) {
    // create an IntegerStack3 object and use variable stack
    // to refer to it
    IntegerStack3 stack = new IntegerStack3();

    // Push seven numbers to the stack object
    stack.push(10);
    stack.push(20);
    stack.push(30);
    stack.push(40);
    stack.push(50);
    stack.push(60);
    stack.push(70);

    // Pop numbers from the stack and display them
    System.out.println(stack.pop());
    System.out.println(stack.pop());
    System.out.println(stack.pop());
    System.out.println(stack.pop());
    System.out.println(stack.pop());
    System.out.println(stack.pop());
    System.out.println(stack.pop());
}
```

With such a `main()` method, you will get the following output:

```
DEBUG: Push 10
DEBUG: Push 20
DEBUG: Push 30
DEBUG: Push 40
DEBUG: Push 50
DEBUG: Push 60
DEBUG: Push 70
The stack is full
DEBUG: Pop
60
DEBUG: Pop
50
DEBUG: Pop
40
DEBUG: Pop
30
DEBUG: Pop
20
DEBUG: Pop
10
DEBUG: Pop
The stack is empty
-1
```

Let's investigate what happens during the program execution. The program execution starts with the `main()` method of the `TestIntegerStack2` class. The first statement to be executed is:

```
IntegerStack3 stack = new IntegerStack3();
```

A new `IntegerStack3` object is created, and its reference is assigned to the local variable `stack`. That is, the variable `stack` is referring to an `IntegerStack3` object.

Then, the `push()` method of the `IntegerStack3` object is repeated seven times:

```
stack.push(10);
stack.push(20);
stack.push(30);
stack.push(40);
stack.push(50);
stack.push(60);
stack.push(70);
```

The first six executions of the `push()` method succeed. Before the `push()` method of the `IntegerStack3` object is called for the seventh time, the array object of the `IntegerStack3` object is shown as in Figure 5.23.

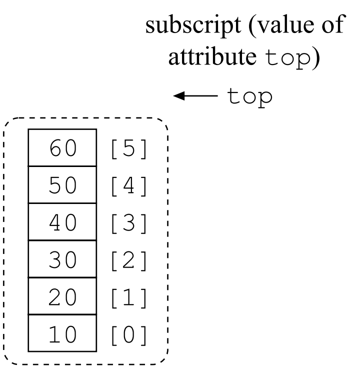


Figure 5.23 The scenario of the array object after executing the push() method six times

Notice the value of `top` is `6` in Figure 5.23. When the `IntegerStack3` object starts executing its `push()` method with parameter `70`, the condition `top < 6` is verified and the result is `false`. Therefore, the else part is executed, and the message `"The stack is full"` is displayed.

Afterwards, the flow of control (the execution sequence of the program) returns to the `main()` method of the `TestIntegerStack2` class. Then, the `pop()` method of the `IntegerStack3` object is called. For the first six method calls, the numbers `60`, `50`, `40`, `30`, `20`, and `10` are returned and displayed. For the seventh method call of the `pop()` method, the value of the attribute `top` is `0` and the `IntegerStack3` object can be visualized in Figure 5.24.

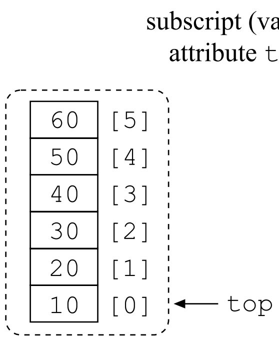


Figure 5.24 The scenario of the array object after executing the pop() method the first six times

(Please be reminded that the values of the array objects are not reset to zero after each `pop()` method call, and the array object still contains those numbers.)

When the `pop()` method of the `IntegerStack3` object is executed for the seventh time, the value of the attribute `top` is zero and the condition `top > 0` is evaluated to be `false`. Then, the `"The stack is empty"` message is shown on the screen. Furthermore, a value of `–1` is returned to where the `pop()` method was called, which is the following statement in the `main()` method:

```
System.out.println(stack.pop());
```

Therefore, a value of `–1` is finally shown on the screen. Please try to trace the program again to make sure that you thoroughly understand the operation of the `IntegerStack3` object.

Please use the following self-test to test your understanding of the use of arrays in the Java programming language.

Self-test 5.1

1 Modify the definition of the class `IntegerStack3` so that it can store ten elements at most.

2 Software developers often need another data structure — queue. It provides an `enqueue()` method that adds a number to it, and a `dequeue()` method that extracts and removes a number from it. The queue operates at a 'first-in-first-out' (FIFO) pattern. That is, the earliest number that is added to it is the first number to be returned and removed from the queue.

Please design and implement an `IntegerQueue` that stores integer numbers and operates in the FIFO pattern. Assume that your `IntegerQueue` object can store six integers at most.

Hint: You can design the class based on the `IntegerStack` class. The `enqueue()` and `dequeue()` methods of a queue correspond to the `push()` and `pop()` methods of a stack respectively. A template for the `IntegerQueue` class definition is:

```
public class IntegerQueue {
    // Attributes

    // Behaviours
    public void enqueue(int number) { ...... }
    public int dequeue() { ...... }
}
```

Furthermore, you can use the following program to test your `IntegerQueue` class:

```
public class TestIntegerQueue {
    public static void main(String args[]) {
        IntegerQueue queue = new IntegerQueue();

        queue.enqueue(10);
        queue.enqueue(20);

        System.out.println(queue.dequeue());
        System.out.println(queue.dequeue());
    }
}
```

In the previous self-test, the first question asks you to modify the `IntegerStack3` class so that it can store up to ten numbers. If you have completed the question, you will find that it is necessary to modify two parts in the definition of the class `IntegerStack3`. The modified program segments are

```
private int[] storage = new int[6];
```

and

```
if (top < 6) {
```

The above two program segments are the only statements that concern the size of the array. Therefore, the numbers in these two statements must match. If the array size (the specified size when the array is created) is larger than the number in the condition, some elements in the array will never be used. However, if the number in the condition is larger than that in the specified array size, runtime errors will occur when executing the `push()` method after all of the array elements are used.

Therefore, it is preferable to have a standardized way to determine the number of elements that an array has. Array is an object in the Java programming language and it is designed to have an attribute `length`. As a result, you can imagine that whenever you create an array, you can access its attribute `length` to determine its size at runtime. For example, executing the following statement

```
int[] storage = new int[6];
```

the scenario can be visualized in Figure 5.25.

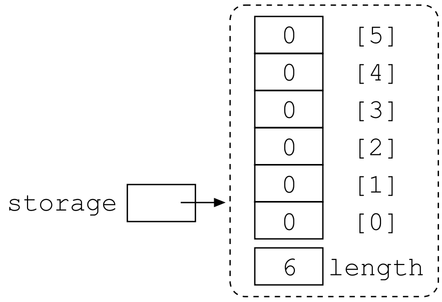


Figure 5.25 The array object with the attribute length shown

As a result, the following expression can be used to determine the number of elements at runtime,

```
storage.length
```

and the condition can be modified as

```
if (top < storage.length) {
    ......
}
```

so that the condition is verified against the number of elements at runtime. The only statement that determines the number of the integers the array object can store is:

```
private int[] storage = new int[6];
```

If it is necessary to modify the array size to, say `10`, the only modification is:

```
private int[] storage = new int[10];
```

Please use the following self-test to modify the definition of the `IntegerQueue` class that you created in the previous self-test to examine the attribute `length` of an array object.

Self-test 5.2

Please modify the definition of the `IntegerQueue` class that you wrote in Self-test 5.1 so that it uses the attribute `length` of the array in the class definition.

If you review all the implementations of the `IntegerStack3` class, you'll find that the array sizes are predetermined at compile time. You might wonder whether it is possible to resize the array so that its size can grow whenever necessary. We said that the length attribute of an array object determines the number of array elements at runtime. However, you cannot assign a new value to it, say

```
storage.length = 7;
```

that intends to change the array size. The above statement causes a compile time error because the attribute `length` is read-only. You cannot change it — you can just access it to retrieve its value.

However, it is possible to create another array object with more array elements. For example, the `push()` method can be written as:

```
// The behaviour to push a new number
public void push(int number) {
    // Show debug message
    System.out.println("DEBUG: Push " + number);

    // Verify whether all array elements have been used
    if (top == storage.length) {
        // Create a new array object with larger size
        storage = new int[storage.length + 1];
    }

    // Store the number to the stack
    storage[top++] = number;
}
```

The values inside the square brackets (examples are bolded in the above `push()` method) specify that the number of array elements is not necessarily a fixed number. It can be any expression that gives a result of `int`. As a result, the statement that creates a new array object whose size is based on the expression is valid.

If the `condition top == storage.length` is `true`, which implies all elements of the array object are used, the `if` part of the if statement is executed. The first statement in the if part creates a new array object that can store one more element than the existing one, and the reference of the array is assigned to the array variable `storage`. Finally, the number specified by the parameter `number` is stored in the array.

Figure 5.26 shows the complete definition of the fourth implementation of the stack design.

```
// Definition of the class IntegerStack (version 4)
public class IntegerStack4 {
    // Attributes
    // The storage for the numbers in the stack using an array with
    // 6 elements
    private int[] storage = new int[6];
    // The attribute for the subscript of the element that can store
    // the newly added number
    private int top = 0;

    // Behaviours

    // The behaviour to push a new number
    public void push(int number) {
        // Show debug message
        System.out.println("DEBUG: Push " + number);

        // Verify whether all array elements have been used
        if (top == storage.length) {
            // Create a new array object with larger size
            storage = new int[storage.length + 1];
        }

        // Store the number to the stack
        storage[top++] = number;
    }

    // The behaviour to pop the last number
    public int pop() {
        // Show debug message
        System.out.println("DEBUG: Pop");

        // A local variable result is declared to store the value
        // to be returned
        int result = -1;

        // Verify whether the stack is empty
        if (top > 0) {
            // Decrease the subscript for the last number and
            // Store the last number to local variable result afterwards
            result = storage[--top];
        }
        else {
            // Show message to prompt the user
            System.out.println("The stack is empty");
        }
        return result;
    }
}
```

Figure 5.26 IntegerStack4.java

This implementation `IntegerStack4` should be free of compile-time errors. Modify the `TestIntegerStack2` so that it uses an `IntegerStack4` object. The program pushes seven numbers to it. Then, you will get the following output:

```
DEBUG: Push 10
DEBUG: Push 20
DEBUG: Push 30
DEBUG: Push 40
DEBUG: Push 50
DEBUG: Push 60
DEBUG: Push 70
DEBUG: Pop
70
DEBUG: Pop
0
DEBUG: Pop
0
DEBUG: Pop
0
DEBUG: Pop
0
DEBUG: Pop
0
DEBUG: Pop
0
```

No runtime error occurs, but all the numbers other than the last one are zero. Why? Before reading the following paragraphs for the explanation, please take a few minutes to review the new version of the `push()` method. Can you figure out why such outputs were obtained?

Let's see why the modified `push()` method gives the wrong output. When the first six numbers have been pushed to the `IntegerStack4` object, its array object is shown in Figure 5.27.

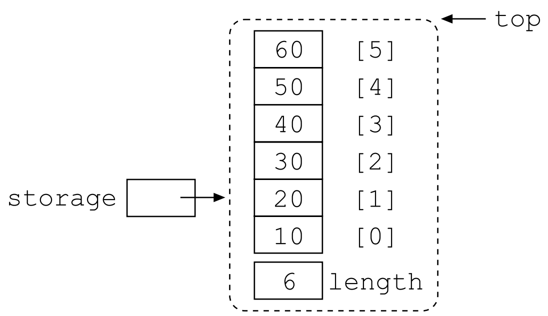


Figure 5.27 The array object of the IntegerStack object after it executes the push() method for the first six times

When the `push()` method is executed for the seventh time, the current value of the attribute `top` is `6` and the condition `top == storage.length` is true. The if part of the `if` statement is executed. The first statement in the if part

```
storage = new int[storage.length + 1];
```

creates a new array object with seven elements, as shown in Figure 5.28.

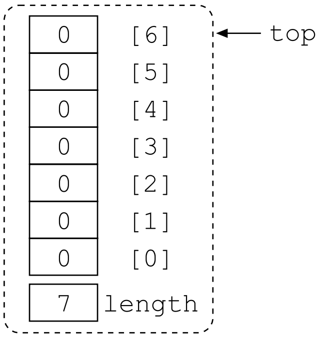


Figure 5.28 The newly created array object with seven elements

Its reference is assigned to the attribute `storage` as shown in Figure 5.29.

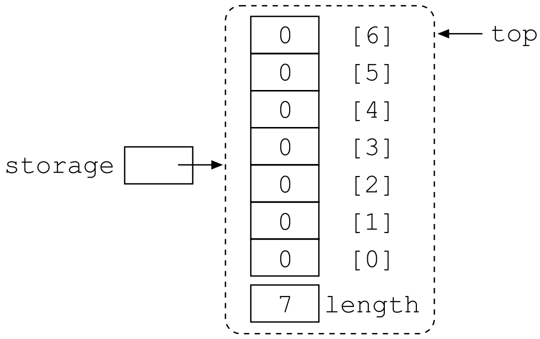


Figure 5.29 Assigning the reference of a newly created array object with seven elements to the storage attribute

You can see that the statement effectively creates a new array object with initial element values of `0`; the attribute `storage` is updated to refer to this. The original array with contents `10`, `20`, `30`, `40`, `50` and `60` is therefore lost.

After the `if` statement, the following statement is executed

```
storage[top++] = number;
```

and it assigns the number `70` to the element with subscript `6` of the array object as shown in Figure 5.30,

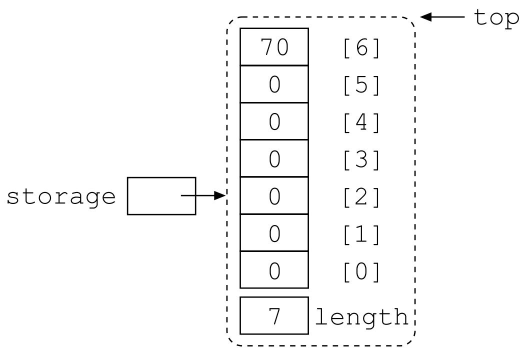


Figure 5.30 The array object of the IntegerStack stack after the value 70 is stored in the array object by the push() method

the attribute `top` is increased to `7`.

Afterwards, the subsequent statements in the `main()` method pop the numbers from the `IntegerStack4` object. The first popped number is `70`; the others are all zeroes.

This version of the `push()` method illustrates another property of arrays in the Java programming language — arrays cannot be resized. You can remedy the problem by creating a new array object with greater capacity and *copying* the array element contents in the original array object to the new one. That issue is discussed in the next section.

The different implementations of the stack design illustrate some core concepts about arrays in the Java programming language. As a conclusion, the following list is provided as a quick reference:

1 Arrays are objects in the Java programming language.

2 Array size is fixed when the array object is created.

3 Each element is accessed by specifying a subscript in the format, *Array-variable*`[`*subscript*`]`. All array subscripts start with zero. For example, in an array object with `n` elements, the subscript ranges from `0` to `n - 1`.

4 Accessing the element with a subscript that is out of the valid range will result in a runtime error.

5 The read-only attribute `length` of an array object stores its array size; that is, the number of array elements the array object has.

Manipulating arrays with loops

You have learned that you can use a `println()` method to display the contents of any expression. That is:

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

The contents of the variable `var` are displayed on the screen no matter what type of variable it is. However, if the variable is an array type that is referring to an array object, the above statement does not display the element contents of the array object; it simply displays something that is not useful to us, such as:

```
[Ljava.lang.String;@1ba34f2
```

Therefore, we have to display the array element contents one by one. It is a repeated operation and it is therefore preferable to do it with looping statements.

In *Unit 4*, we said that it is preferable to use `for` loops for repetitions with a predetermined number of times, so we usually use a `for` loop to manipulate an array object. For example:

```
for (int i=0; i < var.length; i++) {
    ...... // statement that manipulates the element array[i]
}
```

We can use an iteration to access a single array element of an array object. For example, to display the element contents of an array object referred by variable `var`, we can use the following program segment:

```
for (int i=0; i < var.length; i++) {
    System.out.println(var[i]);
}
```

The above `for` loop displays each array element on a new line on the screen. If you want to display all the array elements (especially numbers) that are properly aligned on the screen, you can use the escape character (`\t`) that denotes the tab character. For example:

```
for (int i=0; i < var.length; i++) {
    System.out.print(var[i] + '\t');
}
System.out.println();
```

In the `for` loop, the method `print()` is used instead of `println()` so that the next output is shown on the same line on the screen. Finally, you can use a `println()` method without a parameter to ensure the next output shown on the screen is on a new line.

The following reading gives you some basic ideas about accessing array elements sequentially and randomly with loops. Afterwards, some uses mentioned in the reading are elaborated on further. We provide other examples to illustrate the use of arrays with looping structures in this section.

> ***Reading 5.2***
>
> King, Sections 5.3 and 5.4, pp. 194–98

In the previous section, we mentioned that an array object could not be resized. The only way to use a new array object with a larger size is to create a new array object with a larger size and copy the contents from the original one to the new one. The software library that comes with the Java runtime environment (JRE) enables you to copy the array contents with a single method call. However, the process of using a `for` loop to do so is simple. It is used here for illustration.

To copy the element contents from the existing array object to the new one, there are two array objects to be accessed at the same time. You need an extra local variable to refer to the newly created array object. Then, you can use a loop to copy the element contents one by one. According to this procedure, the `push()` method can be modified as:

```
// The behaviour to push a new number
public void push(int number) {
    // Show debug message
    System.out.println("DEBUG: Push " + number);

    // Verify whether all array elements have been used
    if (top == storage.length) {
        // Create a new array with larger size
        int[] newArray = new int[storage.length + 1];
        // Use a loop to copy the element contents from the
        // original array to the new one.
        for (int i=0; i < storage.length; i++) {
            newArray[i] = storage[i];
        }
        // Update the attribute storage, so that this IntegerStack
        // object is referring to the new array object
        storage = newArray;
    }
    // Store the number and increase the subscript for
    // storing the next number afterwards
    storage[top++] = number;
}
```

The above `push()` method is part of the fifth version of the stack design and defined in the class `IntegerStack5`.

Modify the `TestIntegerStack2` so that it uses an `IntegerStack5` object. Execute it so that it pushes seven numbers to the `IntegerStack5` object. When the `push()` method is executed for the seventh time, the condition (`top == storage.length`) is `true` indicating all array elements of the array object that is referred by the attribute `storage` have been used. Then, the `if` part of the if statement is executed.

The first statement creates a new array that is larger by one than that referred to by the attribute `storage` and is temporarily referred to by local variable `newArray`. The scenario is shown in Figure 5.31.

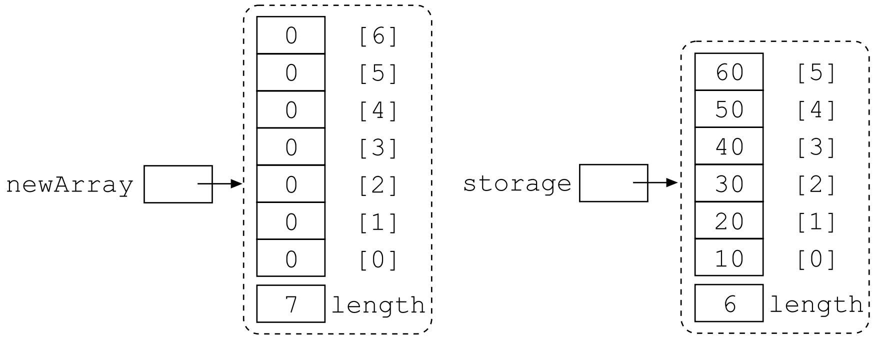


Figure 5.31 The two array objects, the existing one and the newly created one with a larger capacity

Then, the `for` loop starts, and each iteration copies an element's content from the existing array object (referred by `storage`) to the new one (referred by `newArray`). For example, in the first iteration, the value of the loop variable `i` is `0`. The statement

```
newArray[i] = storage[i];
```

copies the element contents from the first element of the array object referred to by `storage` to the first element of the array object referred to by `newArray`. Then, the scenario is shown as in Figure 5.32.

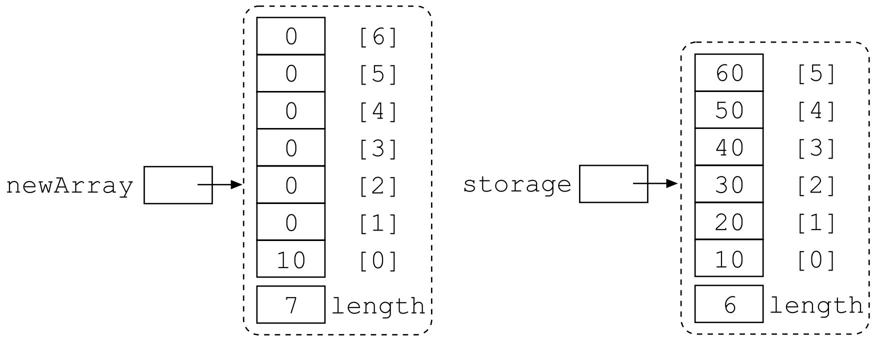


Figure 5.32 Copying the first array element value from the original one to the first array element of the new one

The expression `storage.length` determines the number of iterations. Therefore, there are six iterations, and each copies an element's content. After the `for` loop, the scenario of the array objects is shown in Figure 5.33.

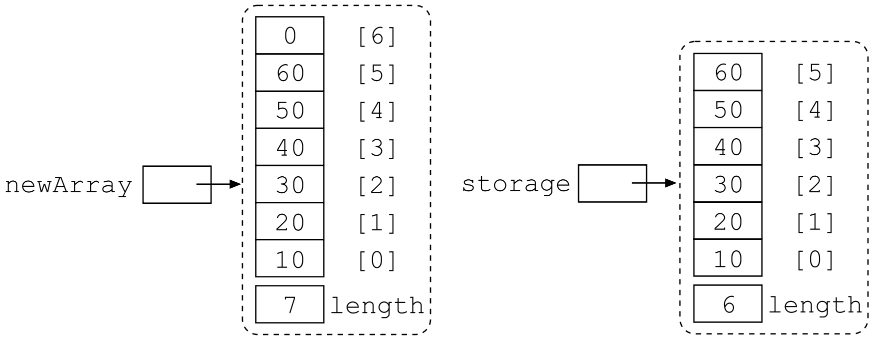


Figure 5.33 The two array objects after all values of the original array object have been copied to the new one

After executing the `for` loop, the following statement is executed

```
storage = newArray;
```

to assign the contents of the local variable `newArray` to the attribute `storage`. As the contents of an array variable are the reference of an array object, the contents of the attribute `storage` are updated to refer to the same array object that is referred to by `newArray`. Then, the original array object that the attribute `storage` referred to is lost. The scenario is shown in Figure 5.34.

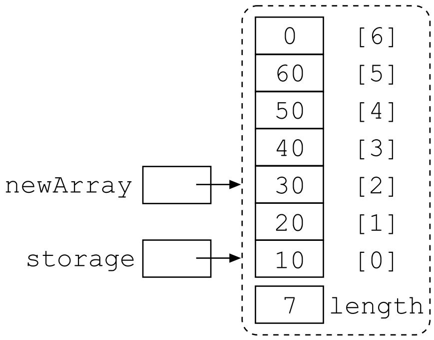


Figure 5.34 The scenario after the attribute storage is updated to refer to the new array object

The last statement in the method

```
storage[top++] = number;
```

stores the value stored in the parameter `number`, `70`, to the array object referred to by `storage`. At this moment, both local variable `newArray` and the attribute `storage` refer to the same array object with seven array elements. The number `70` is stored in the seventh element of the array object, and the attribute `top` is increased by one as shown in Figure 5.35.

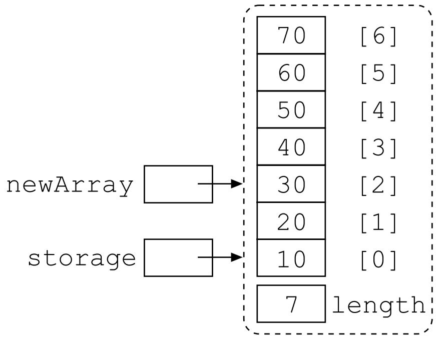


Figure 5.35 The array object after the value 70 is stored

After the numbers are pushed to the `IntegerStack5` object, the numbers are popped from it and displayed. The output is:

```
DEBUG: Push 10
DEBUG: Push 20
DEBUG: Push 30
DEBUG: Push 40
DEBUG: Push 50
DEBUG: Push 60
DEBUG: Push 70
DEBUG: Pop
70
DEBUG: Pop
60
DEBUG: Pop
50
DEBUG: Pop
40
DEBUG: Pop
30
DEBUG: Pop
20
DEBUG: Pop
10
```

For the complete listing of the definition for class `IntegerStack5`, please refer to the file in the CD-ROM. This implementation can store any number of integers and is only limited by the memory available in the JVM.

Self-test 5.3

The definition of the `IntegerStack5` class can store any number of integers. However, it is found that the time required to push the integers to an `IntegerStack5` object increases significantly with the number of integers. For example, Table 5.4 shows the required times for pushing different numbers of integers to the stack object.

Table 5.4 The time required for pushing different numbers of integers to a stack object

Number of integers	Time required
10000	0.69 seconds
30000	12.52 seconds
100000	227.27 seconds

The time required varies on different machines, but the trend clearly indicates that the time required increases dramatically with the number of integers. Why? Is there any way to improve the performance?

Hint: Please review the operations in the `push()` method and pinpoint the statement that can be executed for a smaller number of times.

The following are two examples that need both arrays and looping statements. You will then realize that arrays and looping statements are two indispensable tools in the Java programming language.

Example program — finding the minimum and maximum in a numeric array

In *Unit 4*, you came across an example of finding the minimum and maximum of a series of numbers. A class `MinMaxFinder` was designed and implemented to solve the problem. Now, you are going to implement this class using an array.

The design of the class `MinMaxFinder` in *Unit 4* was:

```
MinMaxFinder
---------------------------
currentMin : int
currentMax : int
---------------------------
testNumber(number : int)
getMinimum() : int
getMaximum() : int
```

If the `MinMaxFinder` is implemented using an array, it is not necessary to keep a current minimum value (`currentMin`) and a current maximum value (`currentMax`). Instead, you can use an array to store all the numbers to be tested, and the minimum value and maximum value are determined on request.

As a result, the `testNumber()` method no longer tests the number but just stores it, and it is therefore preferable to change its name to `storeNumber()` to reflect its real intention. Furthermore, as it is not necessary for a `MinMaxFinder` object to have attributes of current minimum and maximum because the minimum and maximum values are determined 'on the fly', the method names `getMinimum()` and `getMaximum()` are changed to `findMinimum()` and `findMaximum()` respectively.

Therefore, the class design is shown in Figure 5.36.

MinMaxFinder
storage : int[]
storeNumber(number : int) findMinimum() : int findMaximum() : int

Figure 5.36 The initial design of the MinMaxFinder class using an array

When the numbers are stored in the array object, you need an attribute to keep track of the current number of numbers that have been stored, say `total`, as shown in Figure 5.37.

MinMaxFinder
storage : int[] total : int
storeNumber(number : int) findMinimum() : int findMaximum() : int

Figure 5.37 The revised design of MinMaxFinder class

The definition of the `storeNumber()` method can be:

```
public void storeNumber(int number) {
    storage[total++] = number;
}
```

The above definition simply stores the number passed to the method via the parameter `number` in the array object referred to by the `storage` attribute, and the value of the `total` attribute is increased by one afterwards. However, we know that the array size is fixed, and accessing the element with an out-of-bound subscript of an array is a runtime error. Therefore, it is necessary to prevent such an occurrence either by expanding the array as in the definition of `IntegerStack5` class or using an `if/else` statement to safeguard it. For simplicity, the latter approach is used here. The definition of the `storeNumber()` method becomes:

```
public void storeNumber(int number) {
    if (total < storage.length) {
        storage[total++] = number;
    }
    else {
        System.out.println("Too many numbers");
    }
}
```

The `if/else` statement effectively safeguards the runtime error of accessing array elements with invalid subscripts. If the message `"Too many numbers"` is shown on the screen while you are executing the program, you know that all array elements have been used and the array size should be increased in the class definition.

To find the minimum value and maximum value in the array, it is a repeated operation on the array object. We therefore use a loop to do so. As usual, a `for` loop is used:

```
public int findMinimum() {
    int minimum;
    for (int i = 0; i < total; i++) {
        if (storage[i] < minimum) {
            minimum = storage[i];
        }
    }
    return minimum;
}
```

In the definition of the `findMinimum()` method, a local variable `minimum` is declared to store the running minimum value. All numbers that have been stored in the array object are tested with the running minimum value one by one. If the array element is less than the running minimum value, the running minimum value is updated to be that of the array element.

The method definition seems perfect, but it can give a compile time error. When the `condition storage[i] < minimum` is verified for the first time, the variable `minimum` is not yet initialized. The compiler determines this and therefore prompts you with a compile-time error. You should remind yourself that a local variable is not given a default value implicitly, whereas array elements, instance variables and class variables are initialized automatically. Therefore, the initialization of Java is not consistent, and you should explicitly initialize all variables in your class definitions.

The method definition can be modified to remedy the problem as:

```
public int findMinimum() {
    int minimum = Integer.MAX_VALUE;
    for (int i = 0; i < total; i++) {
        if (storage[i] < minimum) {
            minimum = storage[i];
        }
    }
    return minimum;
}
```

The above definition initializes the local variable `minimum` to be `Integer.MAX_VALUE` (the maximum value an integer can have in Java). If no array element is used yet, the entire `for` loop is skipped and the value `Integer.MAX_VALUE` is returned. You can use other values for such purposes, say `–1`, to signify such a scenario. The `Integer.MAX_VALUE` is chosen here so that it behaves like the one in *Unit 4*, because the `getMinimum()` method of the class `MinMaxFinder` presented in *Unit 4* returns such a value if the `testNumber()` method has not been executed.

Similarly, you can implement the `findMaximum()` as:

```
public int findMaximum() {
    int maximum = Integer.MIN_VALUE;
    for (int i = 0; i < total; i++) {
        if (storage[i] > maximum) {
            maximum = storage[i];
        }
    }
    return maximum;
}
```

The complete definition of the class `MinMaxFinder` using an array is shown in Figure 5.38.

```
// Definition of the class MinMaxFinder
public class MinMaxFinder {
    // Attributes
    private int storage[] = new int[1000];
    private int total;

    // Behaviours

    // Store the number to the array
    public void storeNumber(int number) {
        // Verify whether all array elements have been used
        if (total < storage.length) {
            // If there is still room, store the number
            // and update the amount of number stored
            storage[total++] = number;
        }
        else {
            // Show message to user
            System.out.println("Too many storage");
        }
    }

    // Find and return the minimum value in the array
    public int findMinimum() {
        // Declare a local variable for the running minimum
        int minimum = Integer.MAX_VALUE;

        // Verify each array element one by one
        for (int i = 0; i < total; i++) {
            // If the array element is less than the running minimum
            if (storage[i] < minimum) {
                // Store it to the running minimum
                minimum = storage[i];
            }
        }

        // Return the running maximum
        return minimum;
    }

    // Find and return the maximum value in the array
    public int findMaximum() {
        // Declare a local variable for the running maximum
        int maximum = Integer.MIN_VALUE;

       // Verify each array element one by one
        for (int i = 0; i < total; i++) {
            // If the array element is greater than the running minimum
            if (storage[i] > maximum) {
                // Store it to the running maximum
                maximum = storage[i];
            }
        }

        // Return the running maximum
        return maximum;
    }
}
```

Figure 5.38 MinMaxFinder.java

The above class definition initially creates an array object with a size of 1000 elements. To test the class, a `TestMinMaxFinder` class is defined based on the one with the same name in *Unit 4* by changing the method names of the `MinMaxFinder` object. The definition of the `TestMinMaxFinder` class is shown in Figure 5.39.

```
// The class definition of TestMinMaxFinder for setting up the
// environment and test the MinMaxFinder object
public class TestMinMaxFinder {

    // The main executive method
    public static void main(String args[]) {
                     // Create the MinMaxFinder object
                     MinMaxFinder finder = new MinMaxFinder();

                     // Sending messages with numbers to the object so that
                     // the numbers are test one by one
                     finder.storeNumber(15);
                     finder.storeNumber(20);
                     finder.storeNumber(10);

                     // Sending messages to the object to get the current
                     // minimum and maximum number that is hold by it
                     int minNumber = finder.findMinimum();
                     int maxNumber = finder.findMaximum();

                     // Display the result to the screen
                     System.out.println("Minimum number = " + minNumber);
                     System.out.println("Maximum number = " + maxNumber);
    }
}
```

Figure 5.39 TestMinMaxFinder.java

Example program — calculating the sum and average of a sequence of numbers

Here's an example that uses both an array and a looping structure to determine the sum and average of a sequence of numbers.

Like the previous example, you need an object that can help you determine some information from a sequence of numbers. In the previous example, the two pieces of information obtained were the minimum and maximum values. You want to get the sum and average in this example.

As in the `MinMaxFinder`, you use an array to store the numbers for calculation, and you need the `storeNumber()` method of `MinMaxFinder` to store the number. Its definition is:

```
// Store the number to the array
public void storeNumber(int number) {
    // Verify whether all array elements are used
    if (total < storage.length) {
        // If there is still room, store the number
        // and update the amount of numbers stored
        storage[total++] = number;
    }
    else {
        // Show message to user
        System.out.println("Too many numbers");
    }
}
```

To determine the sum of all numbers in the array, you need a variable to store the cumulative total while accessing the array elements one by one. In the beginning, such a variable is initialized to be zero. According to this design, the method is implemented as:

```
// Find and return the sum of all numbers in the array
public int findSum() {
    // Declare a local variable for the cumulative sum
    int sum = 0;
    // Iterate each array element and add it to the
    // cumulative sum
    for (int i = 0; i < total; i++) {
        sum += storage[i];
    }
    // Return the sum
    return sum;
}
```

The average of the numbers is obtained by dividing the sum of all numbers by the amount (number) of the involved numbers. As a result, the method is defined as:

```
public int findAverage() {
    int sum = 0;
    for (int i = 0; i < total; i++) {
        sum += storage[i];
    }
    return sum / total;
}
```

You can see that all statements except the last one are the same as in the `findSum()` method. It is preferable to reuse existing code as much as possible, because there is a single point of modification if enhancement is required. The method `findAverage()` can be modified as:

```
public int findAverage() {
    return findSum() / total;
}
```

The above method definition first calls the `findSum()` method of the object itself to get the sum of all numbers, determines the average by a division operation and finally returns the result. It works, but the result is wrong. Why? Please think for a while before proceeding.

The expressions on both sides of the `/` operator are both of type `int`, which means that it is an integer division. The division result of type `int` ignores the fractional part of the quotient. It is preferable to have the fractional part of the division result, so you need the result to be of type `double`. Therefore, you have to change the integer division to a non-integer division, which can be done by converting the expression on each side of the division operator to type `double`. Therefore, the method `findAverage()` can be modified as

```
public double findAverage() {
    return (double) findSum() / total;
}
```

or

```
public double findAverage() {
    return findSum() / (double) total;
}
```

The return type of the `findAverage()` method is modified to be double so that it matches the return type. The casting operation, `(double)` in this instance, takes precedence over the `/` operator and is right associative. Therefore, the expression that follows `(double)` is converted to type `double` before taking part in the division. The division then becomes a non-integer division. The result will be of type `double`, which contains the fractional part. However, you should notice that the following statements

```
public double findAverage() {
    return (double) (findSum() / total);
}
```

will always give a result of type `double` with no fractional part, because the division expression enclosed in the parentheses is an integer division and gives integer results. Then, the casting operator `(double)` is applied on the integer division result, but the fractional part has been lost in the integer division, which is the reason it will never give a result with a non-zero fractional part.

The `findAverage()` method can now give an accurate average of the numbers. However, you should bear in mind that whenever there is a division operator, there is a risk of a runtime error if the right expression of the division operator is zero, because any number divided by zero is undefined. To resolve such a problem, an `if/else` statement is used to make sure the right expression is non-zero. Therefore, the `findAverage()` method is further enhanced to be:

```
// Find and return the average of all numbers in the array
public double findAverage() {
    // Declare a local variable for the average to be returned
    double average = 0.0;
    // Verify there is at least one number
    if (total > 0) {
        average = (double) findSum() / total;
    }
    else {
        // If there is no number, just return value for
        // not-a-number
        average = Double.NaN;
    }
    // Return the average
    return average;
}
```

The `if/else` statement checks if there is at least one number stored in the array. Then, the average is determined by dividing the sum of all numbers by the number of numbers in the array. Otherwise, a special value `Double.NaN` is returned. `Double.NaN` is a predefined value in the `Double` class of the JRE software library. It indicates that the value is 'not-a-number'.

The methods, `storeNumber()`, `findSum()` and `findAverage()` are now ready to be assembled. Figure 5.40 shows the complete definition of the class `SumAvgFinder`.

```
// Definition of the class SumAvgFinder
public class SumAvgFinder {
    // Attributes
    private int storage[] = new int[1000];
    private int total;

    // Behaviours

    // Store the number to the array
    public void storeNumber(int number) {
        // Verify whether all array elements are used
        if (total < storage.length) {
            // If there is still room, store the number
            // and update the amount of numbers stored
            storage[total++] = number;
        }
        else {
            // Show message to user
            System.out.println("Too many numbers to be stored");
        }
    }

    // Find and return the sum of all storage in the array
    public int findSum() {
        // Declare a local variable for the cumulative sum
        int sum = 0;
        // Iterate each array element and add it to the
        // cumulative sum
        for (int i = 0; i < total; i++) {
            sum += storage[i];
        }
        // Return the sum
        return sum;
    }

    // Find and return the average of all storage in the array
    public double findAverage() {
        // Declare a local variable for the average to be returned
        double average = 0.0;
        // Verify there is at least one number
        if (total > 0) {
            average = (double) findSum() / total;
        }
        else {
            // If there is no number, just return value for
            // not-a-number
            average = Double.NaN;
        }
        // Return the average
        return average;
    }
}
```

Figure 5.40 SumAvgFinder.java

Another class `TestSumAvgFinder` is written to test the `SumAvgFinder` class as shown in Figure 5.41.

```
// The class definition of TestSumAvgFinder for setting up the
// environment and test the SumAvgFinder object
public class TestSumAvgFinder  {

    // The main executive method
    public static void main(String args[]) {
      // Create the SumAvgFinder object and is referred by
      // local variable finder
      SumAvgFinder finder = new SumAvgFinder();

      // Sending messages with numbers to the object so that
      // the numbers are tested one by one
      finder.storeNumber(15);
      finder.storeNumber(19);
      finder.storeNumber(22);

      // Display the result to the screen
      System.out.println("Sum = " + finder.findSum());
      System.out.println("Average = " + finder.findAverage());
    }
}
```

Figure 5.41 TestSumAvgFinder.java

Compile the class definitions and run the program. You will get the following result:

```
Sum = 56
Average = 18.666666666666668
```

Please use the following self-test to practise the skills that use looping statements with arrays.

Self-test 5.4

Write a class named `OddEvenCounter` that counts the number of odd integers (by method `countOdd()`) and even integers (by method `countEven()`) in a sequence of integers. You should write another class `TestOddEvenCounter` that examines an `OddEvenCounter` object with numbers `1`, `4`, `7`, `10`, `12`, `16` and `19`.

Hint: You can use a `%` operator to obtain the remainder of an integer division. For a variable `n`, if the result of `n % 2` is 0, the value of variable `n` is even. Otherwise, it is odd.

Searching

You can now have a collection of data of the same type using arrays, but it is often necessary to determine:

- whether a particular data item exists in the collection or
- whether some data fulfill particular conditions.

For example, you might have a sequence of student scores stored in an array, and you need to count the numbers of passes and fails. Another example is that you want to determine whether your charity ticket won any prize if you are given a list of drawn ticket numbers. Such an operation is known as searching.

There are two common ways to search for data in an array, which are named after the pattern of accessing the array elements. The simplest way is to verify each array element one by one (sequentially) to determine whether the desired data are found. It is like reading the job advertisements one by one to see whether you fit their requirements. If the data are sorted in a particular way, they can be searched more efficiently by checking the values of the data examined, such as locating a company name in the Yellow Pages.

The following sections discuss how the two searching methods can be implemented if an array stores all the data to be searched.

Linear searching

If you imagine that all data to be searched are arranged one after another, you can check each data item one by one to see whether it is that one you are looking for. Such a pattern of searching is known as *linear* or *sequential* searching.

Suppose that you are looking for a number in an array of integers that is referred to by the variable `numbers`. You need a variable of type Boolean, say `found`, to indicate whether the target number is found, and it is initially set to be `false`. The array elements are verified one by one. If an array element is the target number you are looking for, the variable `found` is updated to be `true`. After all array elements are checked or verified, the value of the variable `found` indicates whether the desired number exists in the array.

Based on the above design, you can implement a method named `contains()` as:

```
// Determine whether the array contains the number
public boolean contains(int target) {
    // Declare a local variable to determine whether the desired
    // number has been found
    boolean found = false;

    // Verify each array element to see whether it is the
    // desired number
    for (int i=0; i < total; i++) {
        if (numbers[i] == target) {
            // If it is the desired number, update found to
            // show a desired number is found
            found = true;
        }
    }

    // Return the searching result
    return found;
}
```

The parameter `target` stores the number to be searched for, and the numbers to be searched are stored in the array that is referred to by attribute `numbers`. The above method `contains()` returns either `true` or `false` to indicate whether the desired number is found in the array.

Another implementation of the searching is to return the count of occurrences of the target number in the array. A zero value of count indicates the target number is not found. Otherwise, the count indicates the number of times the target number appears in the array.

Therefore, you can have another method `count()` that performs similarly, but the return value is different:

```
// Count the occurrences of the desired number in the array
public int count(int target) {
    // Declare a variable for the number of occurrences so far
    int result = 0;

    // Verify each array element to see whether it is the
    // desired number
    for (int i=0; i < total; i++) {
        if (numbers[i] == target) {
            // If it is the desired number, increase the number
            // of occurrences
            result++;
        }
    }

    // Return the number of occurrences
    return result;
}
```

In the above `count()` method, a local variable `result` is declared for the number of occurrences of the target number in the array and is initially set to be zero. Each array element of the array is verified one by one. If the target number is found for an array element, the variable `result` is increased by one. After the `for` loop, the value of the variable `result` is returned.

Like the previous `SumAvgFinder` class definition, you need a `storeNumber()` method to store a number in the array. By combining all methods, `storeNumber()`, `contains()` and `count()`, you have the definition of the class `LinearSeacher` class as shown in Figure 5.42.

```
// Definition of class LinearSearcher
public class LinearSearcher {
    // Attributes
    private int numbers[] = new int[1000];
    private int total;

    // Behaviours

    // Store the number to the array
    public void storeNumber(int number) {
        // Verify whether all array elements are used
        if (total < numbers.length) {
            // If there is still room, store the number
            // and update the amount of number stored
            numbers[total++] = number;
        }
        else {
            // Show message to user
            System.out.println("Too many numbers");
        }
    }

    // Determine whether the array contains the number
    public boolean contains(int target) {
        // Declare a local variable to store whether the desired
        // number has been found
        boolean found = false;

        // Verify each array element to see whether it is the
        // desired number
        for (int i=0; i < total; i++) {
            if (numbers[i] == target) {
                // If it is the desired number, update found to
                // show a desired number is found
                found = true;
            }
        }

        // Return the searching result
        return found;
    }

    // Count the occurrences of the desired number in the array
    public int count(int target) {
        // Declare a variable for the number of occurrences so far
        int result = 0;

        // Verify each array element to see whether it is the
        // desired number
        for (int i=0; i < total; i++) {
            if (numbers[i] == target) {
                // If it is the desired number, increase the number
                // of occurrences
                result++;
            }
        }

        // Return the number of occurrences
        return result;
    }
}
```

Figure 5.42 LinearSearcher.java

Now you have to design a class, say `TestLinearSearcher`, to examine the `LinearSearcher` class. To make it more interactive, the `main()` method of the `TestLinearSearcher` class is written to enable the user to enter a sequence of integers and then enter the desired numbers to be searched for in the array.

First of all, it is necessary to create an object of the `LinearSearcher` class.

```
LinearSearcher searcher = new LinearSearcher();
```

To enable the user to enter the numbers, we use a dialog to get the input from the user and the input value is stored in the `LinearSearcher` object. A loop is used to repeat getting the numbers until a special value `–1` is entered.

```
do {
  // Get a number from user as a String object
  String inputNumber =
      JOptionPane.showInputDialog(
           "Enter a number [-1 to end]");

  // Derive the integer from the String object
  number = Integer.parseInt(inputNumber);

  // If the number is not -1, put it to the searcher
  // object
  if (number != -1) {
      searcher.storeNumber(number);
  }
} while (number != -1);
```

If the user enters `–1`, the above loop is terminated. Another loop is started that accepts a number from the user and searches for it using the `searcher` object.

```
// A do/while loop to get a number from the user and
// search for it using the searcher object
do {
  // Get a number from user as a String object
  String inputNumber =
      JOptionPane.showInputDialog(
           "Enter a number to search [-1 to end]");

  // Derive the integer from the String object
  number = Integer.parseInt(inputNumber);

  // If the number is not -1, search it with the searcher
  // object and prepare a String to be displayed in a
  // dialog
  if (number != -1) {
      String output =
          "Found = " + searcher.contains(number) +
          "\nOccurrence = " + searcher.count(number);
      JOptionPane.showMessageDialog(null, output);
  }
```

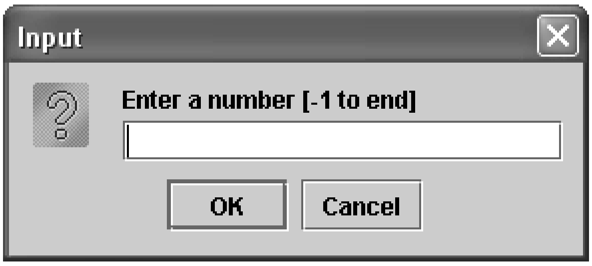
```
} while (number != -1);
```

Figure 5.43 shows the complete definition of the `TestLinearSearcher` class.

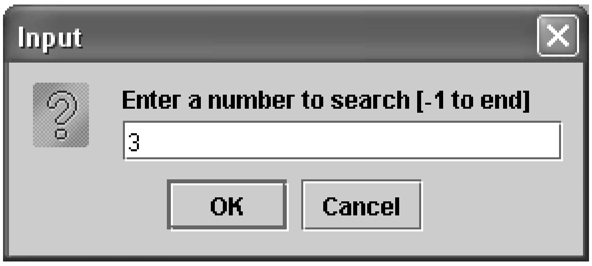
```
// Import statement for the JOptionPane
import javax.swing.JOptionPane;

// Definition of the class TestLinearSearcher
public class TestLinearSearcher {

    // Main executive method
    public static void main(String args[]) {
        // Create a LinearSearcher object and is referred by variable
        // searcher
        LinearSearcher searcher = new LinearSearcher();

        // A variable for storing the number entered by the user
        int number;

        // A do/while loop to get numbers from users and put it
        // to the searcher object
        do {
            // Get a number from user as a String object
            String inputNumber =
                JOptionPane.showInputDialog(
                    "Enter a number [-1 to end]");

            // Derive the integer from the String object
            number = Integer.parseInt(inputNumber);

            // If the number is not -1, put it to the searcher
            // object
            if (number != -1) {
                searcher.storeNumber(number);
            }
        } while (number != -1);

        // A do/while loop to get a number from the user and
        // search it by the searcher object
        do {
            // Get a number from user as a String object
            String inputNumber =
                JOptionPane.showInputDialog(
                    "Enter a number to search [-1 to end]");

            // Derive the integer from the String object
            number = Integer.parseInt(inputNumber);

            // If the number is not -1, search it with the searcher
            // object and prepare a String to be displayed in a
            // dialog
            if (number != -1) {
                String output =
                    "Found = " + searcher.contains(number) +
                    "\nOccurrence = " + searcher.count(number);
                JOptionPane.showMessageDialog(null, output);
            }
        } while (number != -1);

        // Terminate the program explicitly
        System.exit(0);
    }
}
```

Figure 5.43 TestLinearSearcher.java

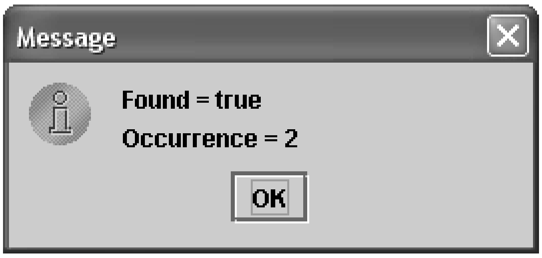
Compile and execute the source code. You'll be prompted with the following dialog:



For example, you enter the numbers `"11"`, `"5"`, `"3"`, `"8"`, `"5"`, `"3"`, `"12"` and `"-1"` individually by entering the number in the text field followed by **<Enter>** or clicking **<OK>**. The array object of the `LinearSearch` object contains the numbers `11`, `5`, `3`, `8`, `5`, `3` and `12`. Afterwards, you are prompted with the following dialog for entering the desired number to be searched for:



You can now search for the numbers by entering them in the dialog. For example, if the number `3` is entered, the following dialog shows you the result:



You can then press **<Enter>** or click **<OK>**. The previous input dialog is shown again for entering the next number to be searched for. To terminate the program, enter `-1` as the input.

The linear searching is simple and works well for small numbers of numbers. As each number is searched one by one, the time required to search for a target number increases with the number of numbers in the array object. Therefore, it is not an efficient way to search for a target number among a huge number of numbers.

The following section discusses another searching method that is much more efficient, although it is a little more complicated.

Binary searching

If you were given a list of students in your class in which the names were sorted in ascending order and you wanted to know your class number from the list, what would you do with the student list? Do you have to go through the whole list from the beginning to the end?

Probably, you will look at about middle of the list to see whether your name alphabetically precedes the one there. If so, you are sure that your name will not appear in the second half of the list, and you only need to be concerned with the first half of the list. Then, you can locate the name in the middle of the first half and see if your name precedes it or not. You can then determine whether your name will be in the first quarter or the second quarter of the list. By repeating the operation every time you verify a name in the list, you can narrow the searching scope by half until the searching scope contains only one student name. You can then either read your student number from the list or confirm that your name does not appear in the list.

For a list of sorted data, the above approach can be used to repeatedly separate the searching scope into two halves and verify whether the target data exist in the first half or in the second half, until there is only one element left in the searching scope. It is a common searching mechanism that is usually known as binary searching.

In the software library of the JRE, there are implementations of the binary searching mechanism. However, you can learn its detailed operations by designing and implementing your own method that uses binary searching.

First of all, let's look at the actual operation with respect to some real numbers. For example, there are eight sorted numbers, `1`, `23`, `43`, `56`, `76`, `84`, `93` and `97`, and it is necessary to determine whether the target number `84` is in the list. For the actual operation, you use an array object to store all these numbers as shown in Figure 5.44.

Subscript	`[0]`	`[1]`	`[2]`	`[3]`	`[4]`	`[5]`	`[6]`	`[7]`
Value	`1`	`23`	`43`	`56`	`76`	`84`	`93`	`97`
	L							U

Figure 5.44 The initial scenario when the searching starts

The letters L and U in the above table indicate the lower bound (lower limit) and the upper bound (upper limit) subscript (searching scope) respectively. Therefore, L and U are given the initial values `0` and `7` respectively, which means that all the numbers are taken into account.

Now, you can locate the data in the middle of the list for verification. The subscript of the data in the middle can be considered the average of the two searching scope bounds; that is, `(7 + 0) / 2` which is `3` by integer division. The letter M in Figure 5.45 indicates the array element with the middle subscript in the searching scope.

Subscript	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
Value	1	23	43	56	76	84	93	97
	L			M				U

Figure 5.45 The subscript for the middle item is determined

You can then verify the condition, (target number ≤ middle element). If the result is `true`, you can conclude that the target number is in the searching scope of the lower to middle bounds. Otherwise — if the result is `false` — the searching scope is from the middle subscript (bound) plus one to the upper bound. It can be visualized in Figure 5.46.

Subscript	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
Value	1	23	43	56	76	84	93	97
	L			M				U
Next searching scope for condition results	true				false			

Figure 5.46 The searching scopes for the two condition results

In our case, the condition (`84 ≤ 56`) is `false` and the searching scope is narrowed from subscript `4` to `7`. Then, the first half can be ignored and the lower bound of the searching scope is updated to `4`. The scenario is shown in Figure 5.47. The shaded region represents the array elements that are out of the searching region.

Subscript	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
Value	1	23	43	56	76	84	93	97
					L	M		U
Next searching scope for condition results					true		false	

Figure 5.47 The scenario after the first comparison

As shown in Figure 5.47, the middle subscript is again the average of the two bounds, which is `(4 + 7) / 2` that gives `5`. The condition is verified again (target number (`84`) ≤ middle data (`84`)). By human operation, we know that the target number is found, but a computer must continue the predefined steps until the searching scope contains one number. For this condition, the result is `true` and the upper searching bound is updated to be the middle subscript `5` and the searching scope covers data with subscripts from `4` to `5` as shown in Figure 5.48.

Subscript	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
Value	1	23	43	56	76	84	93	97
					LM	U		
Next searching scope					true	false		

Figure 5.48 The scenario after the second comparison

The middle subscript is determined by `(4 + 5) / 2`, which is `4`. The condition, target number (`84`) ≤ middle data (`76`) is verified. The result is `false` and the lower searching bound is updated to be the middle subscript plus one, which is `5`. Now, both the lower searching bound and the upper searching bound are `5` as shown in Figure 5.49.

Subscript	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
Value	1	23	43	56	76	84	93	97
						LU		

Figure 5. 49 After three comparisons, the searching scope contains one element only

It is the only data item in the searching scope, and you have completed narrowing the search. You can determine that the only number in the searching scope is either the target number, or the target number is not found in the number list. In our example, the target number is found at subscript `5`.

For this example, you used three comparisons to narrow the searching scope to one element, and a last comparison to determine whether the target number is found. Compared with linear searching, you need eight comparisons and it seems that the benefit is not significant. However, binary searching uses 11 comparisons to narrow a list of around 1000 data items to one item, which is a great gain in performance.

The above design can be implemented using the Java programming language. Assume that the sorted numbers are stored in an array referred to by a variable `numbers` and an attribute `total` specifies the amount of numbers stored in the array. Then, two local variables for storing the lower searching bound and the upper searching bound are declared:

```
int lowerBound = 0;
int upperBound = total - 1;
```

The value of the upper searching bound is the value of the `total` minus one, because the array subscript starts at zero. For example, if the array object contains eight numbers, the value of the attribute total is `8` and the subscripts therefore range from `0` to `7`.

In every step to narrow the searching scope, you need to determine the middle subscript and another local variable is defined:

```
int middle = (lowerBound + upperBound) / 2;
```

If the target number is stored in a variable named `target`, you can update either the lower searching bound or the upper searching bound, based on the comparison result of (`target <= numbers[middle]`). As a result, you have the following `if/else` statement:

```
if (target <= numbers[middle]) {
    upperBound = middle;
}
else {
    lowerBound = middle + 1;
}
```

The above `if/else` statement updates the upper searching bound to be the `middle` subscript if the condition is `true` or updates the lower searching bound to be `middle` plus one if the condition is `false`.

The above two operations are performed repeatedly until there is only one element in the searching scope, which means that the above two operations are performed while the lower searching bound is less than the upper searching bound.

```
while (lowerBound < upperBound) {
    ......
}
```

By combining all of the above pieces of code, you can come up with the following method `contains()` that can be used to determine whether a number can be found in an array that is referred by the variable `numbers`.

```
// Determine whether a number can be found in the array by
// binary searching
public boolean contains(int target) {
    // Declare and initialize the lower searching bound
    int lowerBound = 0;
    // Declare and initialize the upper searching bound
    int upperBound = total - 1;

        // Repeat while the lower searching bound is less than the
        // the upper searching bound
        while (lowerBound < upperBound) {
            // Determine the middle subscript
            int middle = (lowerBound + upperBound) / 2;

            // Update either lower or upper searching bound
            if (target <= numbers[middle]) {
                upperBound = middle;
            }
            else {
                lowerBound = middle + 1;
            }
    }

    // Return the result whether the number in the searching
    // scope is the target number
    return target == numbers[lowerBound];
}
```

You need another method `storeNumber()` as in the `LinearSearcher` to store a number in the array. The complete definition of the `BinarySearcher` class is shown in Figure 5.50.

```
// Definition of the class BinarySearcher
public class BinarySearcher {
    // Attributes
    private int numbers[] = new int[1000];
    private int total;

    // Behaviours

    // Store the number to the array
    public void storeNumber(int number) {
        // Verify whether all array elements are used
        if (total < numbers.length) {
            // If there is still room, store the number
            // and update the amount of number stored
            numbers[total++] = number;
        }
        else {
            // Show message to user
            System.out.println("Too many numbers");
        }
    }

    // Determine whether a number can be found in the array by
    // binary searching
    public boolean contains(int target) {
        // Declare the initialize the lower searching bound
        int lowerBound = 0;
        // Declare and initialize the upper searching bound
        int upperBound = total - 1;

            // Repeat while the lower searching bound is less than the
            // the upper searching bound
        while (lowerBound < upperBound) {
            // Determine the middle subscript
            int middle = (lowerBound + upperBound) / 2;

            // Update either lower or upper searching bound
            if (target <= numbers[middle]) {
                upperBound = middle;
            }
            else {
                lowerBound = middle + 1;
            }
        }

        // Return the result whether the number in the searching
        // scope is the target number
        return target == numbers[lowerBound];
    }
}
```

Figure 5.50 BinarySearcher.java

The last statement

```
return target == numbers[lowerBound];
```

returns a `boolean` value.

A class named `TestBinarySearcher` is prepared to examine the `BinarySearcher` class definition. It is written as shown in Figure 5.51. You should notice that the numbers supplied to the `BinarySearcher` object for searching are in ascending order.

```
// Import statement for JOptionPane
import javax.swing.JOptionPane;

// Definition of class TestBinarySearcher
public class TestBinarySearcher {
    // Main executive method
    public static void main(String args[]) {
        // Create a BinarySearcher object that is referred by variable
        // searcher
        BinarySearcher searcher = new BinarySearcher();

        // Store the numbers in the BinarySearcher object
        // note that the data have to be stored in ascending order
        searcher.storeNumber(1);
        searcher.storeNumber(23);
        searcher.storeNumber(43);
        searcher.storeNumber(56);
        searcher.storeNumber(76);
        searcher.storeNumber(84);
        searcher.storeNumber(93);
        searcher.storeNumber(97);

        // Get a number from user
        String inputNumber = JOptionPane.showInputDialog(
            "Please enter a number");
        int number = Integer.parseInt(inputNumber);

        // Determine whether the input number is in the list of
        // stored numbers, and display message accordingly
        if (searcher.contains(number)) {
            JOptionPane.showMessageDialog(null,
                "The number (" + number + ") is found");
        }
        else {
            JOptionPane.showMessageDialog(null,
                "The number (" + number + ") cannot be found");
        }

        // Terminate the program explicitly
        System.exit(0);
    }
}
```

Figure 5.51 TestBinarySearcher.java

Please use the following activity to further investigate the operations in the `contains()` method of the `BinarySearcher`.

Activity 5.1

In *Unit 4*, you learned the technique to debug software, such as displaying messages in a loop so that you can monitor the changes in the variables in a loop.

Please add statements in the `while` loop of the `contains()` method of the `BinarySearcher` class so that the lower searching bound and upper searching bound are displayed when an iteration starts and ends.

Compile and execute the `TestBinarySearcher` again. Please study the debug messages to confirm that the changes in the bounds are exactly the same as discussed in this section.

Creating and using arrays of objects

We mentioned that array in the Java programming language can be used to manipulate a collection of data of the same type. Up to now, we have discussed arrays with array elements of primitive types. It is possible, however, to have arrays with non-primitive type array elements.

You learned earlier in this unit that there are a few steps in creating and using arrays in the Java programming language. They are:

1 declaring a variable that can refer to an array object

2 creating the array object and assigning its reference to the array variable

3 accessing the elements in the array by specifying a subscript or accessing the entire array object with its reference.

The above steps apply to both the array of primitive types and non-primitive types.

The way to declare a non-primitive array variable is the same. For example:

```
String[] greetings;
```

The difference between an array of primitive type and an array of non-primitive type is the type of the array element is non-primitive. In the above statement, the array element type is `String`, which is non-primitive.

To create an array object of non-primitive type, the statement is similar. For example, the following statement creates an array object with three array elements, each of type `String`.

```
new String[3]
```

As usual, the keyword `new` creates an object and returns the reference of the newly created object. Therefore, it is usual to store the reference of the object just after the object is created, such as:

```
greetings = new String[3];
```

Then, you have the scenario as shown in Figure 5.52.

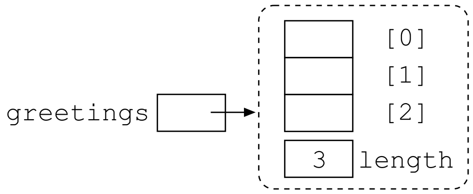


Figure 5.52 An array object with String array element type that is referred to by variable greetings

(Earlier in this unit, the subscripts were listed in descending order from top to bottom to mimic a stack. The order of subscripts in the above diagram is now shown naturally in ascending order from top to bottom.)

You can then access any array element as usual by using a subscript. According to Table 5.1, we said that the initial value for array elements of non-primitive type is `null`. Therefore, the scenario shown in Figure 5.52 is revised to be the exact one as shown in Figure 5.53.

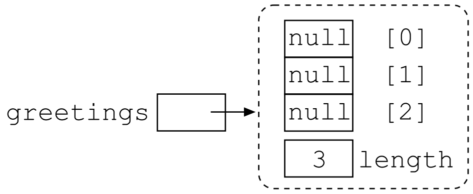


Figure 5.53 Array object of element type of type String with initial value null

You learned that the array element of an array object of type `int` could be considered a usual variable of type `int`. For example, you can have the statement

```
int[] numbers = new int[3];
```

that creates an array object with three array elements and the type of each element is `int`. Therefore, it is possible to use the array elements as if they are usual variables of type `int`, such as:

```
numbers[0] = 123;
```

Therefore, each array element of an array object of type `String` can be considered a variable of type `String`. Recall that you can have a statement

```
String message = "Hello World";
```

that assigns the reference of a String object with the content "Hello World" to the String variable message. Then, you can also have statements like:

```
String[] greetings = new String[3];
greetings[0] = "Hello World";
```

The first array element of the array object that is referred by the variable `greetings` now stores the reference to the `String` object with the content `"Hello World"`.

To make it more explicit, let's consider the following two program segments:

```
int[] numbers = new int[3];
numbers[0] = 0;
numbers[1] = 1;
numbers[2] = 2;
```

and

```
String[] greetings = new String[3];
greetings[0] = "Good morning";
greetings[1] = "Good afternoon";
greetings[2] = "Good evening";
```

The program segments look alike, but their scenarios are significantly different. Their scenarios are shown in Figure 5.54.

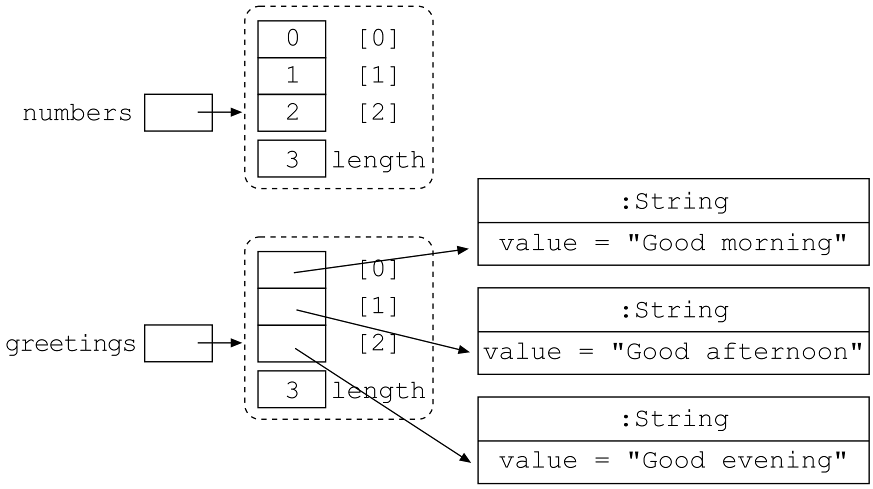


Figure 5.54 Scenarios of array objects of primitive and non-primitive types

You can see that the contents of array element of non-primitive type store the *reference* to an object of the corresponding types.

Please notice that the non-primitive type `String` is a special case in the Java programming language, because a sequence of characters enclosed in a pair of double quotation marks (`"`) is converted into a `String` object at compile time. It is usually necessary to use the keyword `new` to create the objects and assign them to the element arrays. For example, if you need two `TicketCounter` objects to be referred to by two array elements, the program segment can be:

```
TicketCounter[] counters = new TicketCounter[2];
counters[0] = new TicketCounter();
counters[1] = new TicketCounter();
```

The above program segment sets up the scenario that is shown in Figure 5.55.

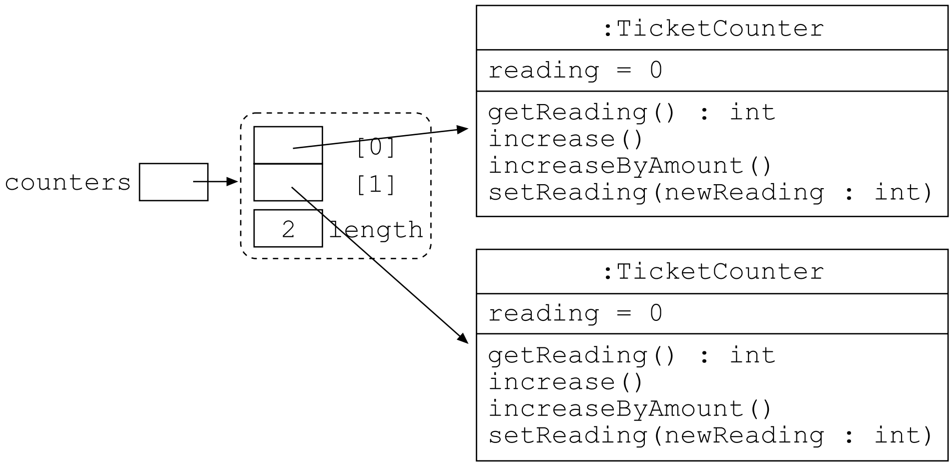


Figure 5.55 Consolidating two TicketCounter objects with an array object

You use dot notation to send a message to the object when requesting it to perform one of its behaviours; that is, to call a method of the object. For example:

```
TicketCounter counter = new TicketCounter();
counter.increase();
```

As mentioned, an array element of non-primitive type can be considered a variable of that non-primitive type. It is possible to assign its contents to a usual non-primitive variable and send a message to it as usual, just like:

```
TicketCounter[] counters = new TicketCounter[2];
counters[0] = new TicketCounter();
counters[1] = new TicketCounter();
.
.
.
TicketCounter alias = counters[0];
alias.increase();
```

A variable `alias` of type `TicketCounter` is declared and is assigned the contents of the first array element of the array object that is referred to by the variable `counters`. The content of the first array element is the reference of that `TicketCounter` object. Therefore, the reference is copied to the variable alias, which means that both `counters[0]` and `alias` are referring to the same `TicketCounter` object. Then, the last statement sends a message `increase` to the referred `TicketCounter` object.

Because both `counters[0]` and `alias` are referring to the same object, it is reasonable to replace the variable `alias` with the expression `counters[0]`; that is:

```
(counters[0]).increase();
```

The pair of square brackets and the dot operator are of the same precedence and both are left-associative. The parentheses in the above statement are hence optional. As a result, the following statement suffices:

```
counters[0].increase();
```

If there is any accessible object attribute, it is possible to access it similarly; for example if the attribute `reading` of the `TicketCounter` is not `private`, such as `public`, it is possible to access it in the following format:

```
counters[0].reading = 10;
```

An array of objects is used in the following class definition of `TestFairCoin` to count the outcomes of flipping a coin. The complete definition of the `TestFairCoin` class is shown in Figure 5.56.

```
// Definition of the class TestFairCoin
public class TestFairCoin {

    // Main executive method
    public static void main(String args[]) {
        // Create an array of type TicketCounter
        TicketCounter[] counters = new TicketCounter[2];

        // Create two TicketCounter objects and assign them to the
        // array elements of the array object
        counters[0] = new TicketCounter();
        counters[1] = new TicketCounter();

        // Flip the coin 10000 times
        for (int i=0; i < 10000; i++) {
            // Determine whether the result is head (0) or
            // tail (1)
            int side = (int) (Math.random() * 2.0);
            // Increase the corresponding counter
            counters[side].increase();
        }

        // Display the results
        System.out.println("Count of Heads = " +
            counters[0].getReading());
        System.out.println("Count of Tails = " +
            counters[1].getReading());
    }
}
```

Figure 5.56 TestFairCoin.java

The definition of the class `TicketCounter` is the same as that in *Unit 3* and is therefore not listed here. Please refer to *Unit 3* for the complete definition.

In the `main()` method of the above `TestFairCoin` class definition, it first creates an array object of type `TicketCounter` with two array elements. Then, it creates two `TicketCounter` objects and assigns their references to the two array elements.

```
// Create an array of type TicketCounter
TicketCounter[] counters = new TicketCounter[2];

// Create two TicketCounter objects and assign them to the
// array elements of the array object
counters[0] = new TicketCounter();
counters[1] = new TicketCounter();
```

Then, there is a `for` loop that repeats 10,000 times. Each iteration simulates flipping a coin by the following statement:

```
int side = (int) (Math.random() * 2.0);
```

The `Math.random()` gives a random number that is greater than or equal to zero and less than one. Therefore,

```
0.0 ≤ Math.random() < 1.0
0.0 ≤ Math.random() * 2.0 < 2.0
```

After casting from `double` to `int`, the result is either `0` or `1`. Afterwards, the value is assigned to the local variable `side`. It is used as the subscript of the array object referred to by the variable `counters` to determine to which `TicketCounter` object the message `increase` is sent in the following statement:

```
counters[side].increase();
```

After 10,000 iterations, the readings of the two `TicketCounter` objects are obtained and displayed on the screen.

```
System.out.println("Count of Heads = " +
    counters[0].getReading());
System.out.println("Count of Tails = " +
    counters[1].getReading());
```

Compile and execute the program. A possible output is:

```
Count of Heads = 4951
Count of Tails = 5049
```

As the returned value of `Math.random()` varies for each method call, you will find that the output message differs every time you execute it.

It is possible to write a class definition that performs similarly without using an array of objects. The above `TestFairCoin` class is written to illustrate a possible use of arrays of non-primitive type.

Please use the following self-test to experience using arrays of non-primitive type.

Self-test 5.5

Write a class named `CountNumbers` that enables the user to enter a sequence of numbers that fall within the range `0` to `9` and displays the occurrences of each number.

Hint: In the `main()` method of the `CountNumbers` class, create an array object of element type `TicketCounter` and assign ten `TicketCounters` to the array elements. Then, use a loop that repeatedly displays a dialog for getting the numbers and calls the `increase()` method of the corresponding `TicketCounter` object that is referred to by an array element. The entry is terminated by a value of `–1`. Afterwards, it displays the number of times each number occurs.

Example program — calculating the total price of items at the cashier's counter

A software system for a supermarket must enable a cashier to calculate the total price of all items that are chosen by a customer. To design the software application that supports such an operation, you first have to consider the operational details to calculate the total price.

The items of the same products are grouped. For each such item group, the name, price and quantity are recorded. The total price of all items is the sum of the subtotals of all item groups.

According to the above description, an Item class is designed to model a single item group. Its design is shown in Figure 5.57.

Item
name : String price : double quantity : int
setName(theName : String) setPrice(thePrice : double) setQuantity(theQuantity : int) findTotal() : double

Figure 5.57 The design of Item class

The definition of the above `Item` class is obvious. Its complete definition in the Java programming language is shown in Figure 5.58.

```
// Definition of class Item
public class Item {
    // Attributes
    private String name;
    private double price;
    private int quantity;

    // Behaviours

    // Set the attribute name
    public void setName(String theName) {
        name = theName;
    }

    // Set the attribute price
    public void setPrice(double thePrice) {
        price = thePrice;
    }

    // Set the attribute quantity
    public void setQuantity(int theQuantity) {
        quantity = theQuantity;
    }

    // Get the subtotal of this item
    public double findTotal() {
        return price * quantity;
    }
}
```

Figure 5.58 Item.java

You need another class `PurchaseOrder` that models a single purchase of a customer. Its design is shown in Figure 5.59.

PurchaseOrder
items : Item[] itemCount : int
addItem(name : String, price double, quantity : int) findTotal() : double

Figure 5.59 The design of PurchaseOrder class

The `PurchaseOrder` has an attribute `items` of type array of `Item` to store all the chosen items. The `addItem()` method of the `PurchaseOrder` class creates a new `Item` object and stores it in the array object referred to by the `items` attribute. The total price of all items to be determined by the `findTotal()` method is the sum of the subtotals of all items.

The definition of the class `PurchaseOrder` is shown in Figure 5.60.

```
// Definition of class PurchaseOrder
public class PurchaseOrder {
    // Attributes
    private Item[] items = new Item[1000];
    private int itemCount = 0;

    // Behaviours

    // To create and add a new item to the array object
    public void addItem(String name, double price, int quantity) {
        // Create a new Item object
        Item newItem = new Item();

        // Set its attributes
        newItem.setName(name);
        newItem.setPrice(price);
        newItem.setQuantity(quantity);

        // Add the Item object to the array object
        items[itemCount++] = newItem;
    }

    // Get the total of all items
    public double findTotal() {
        double sum = 0.0;
        for (int i=0; i < itemCount; i++) {
            sum += items[i].findTotal();
        }
        return sum;
    }
}
```

Figure 5.60 PurchaseOrder.java

To drive the `Item` and `PurchaseOrder` classes, you need a driver program that sets up the execution environment. A `Cashier` class that is designed to set up the environment and calculate the total price of the items is shown in Figure 5.61.

```
// Definition of class Cashier
public class Cashier {

    // Main executive method
    public static void main(String args[]) {
        // Create a new PurchaseOrder object to be referred by
        // the local variable order
        PurchaseOrder order = new PurchaseOrder();

        // Add the items to the purchase order
        order.addItem("Coke", 2.5, 12);
        order.addItem("Milk", 6.0, 1);
        order.addItem("Egg", 0.5, 18);

        // Display the total price
        System.out.println("Total price = " + order.findTotal());
    }
}
```

Figure 5.61 Cashier.java

Compile the above three class definitions and execute the `Cashier` program. You'll get the total price as:

```
Total price = 45.0
```

The `main()` method of the `Cashier` class does not create the `Item` objects. Instead, it calls the `addItem()` method of a `PurchaseOrder` object so that it creates and adds a new `Item` object to its array object.

The scenario is a bit complicated. Therefore, let's examine what is going on during the execution.

When the program starts, the first statement is executed to create a `PurchaseOrder` object and have it referred to by the local variable `order`:

```
PurchaseOrder order = new PurchaseOrder();
```

The situation is shown in Figure 5.62.

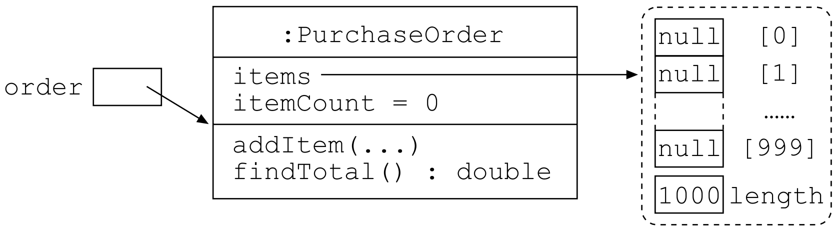


Figure 5.62 The initial scenario when the PurchaseOrder object is created

(The parameter list of the `addItem()` method is not shown exactly, for simplicity.)

The array object of element type `Item` is created and assigned to the attribute `items` of the `PurchaseOrder` object when the `PurchaseOrder` object is created. Initially, the value of the attribute `itemCount` is initialized to be zero, and all array elements of the array object referred to by the attribute `items` are implicitly initialized to be `null`.

Then, the `main()` method executes the following statement:

```
order.addItem("Coke", 2.5, 12);
```

Such a statement sends a message `addItem` with supplementary data to the reference of a `String` object with contents `"Coke"` (of type `String` for parameter `name`), `2.5` (of type `double` for parameter `price`) and `12` (of type `int` for parameter `quantity`) to the `PurchaseOrder` object that is referred to by the variable `order`. The `PurchaseOrder` object performs its `addItem()` behaviour.

```
// To create and add a new item to the array object
public void addItem(String name, double price, int quantity) {
    // Create a new Item object
    Item newItem = new Item();

    // Set its attributes
    newItem.setName(name);
    newItem.setPrice(price);
    newItem.setQuantity(quantity);

    // Add the Item object to the array object
    items[itemCount++] = newItem;
}
```

When the `addItem()` method of the `PurchaseOrder` object is executed, it first creates a new `Item` object,

```
Item newItem = new Item();
```

which is referred to by the local variable `newItem`. It sets up the scenario as shown in Figure 5.63.

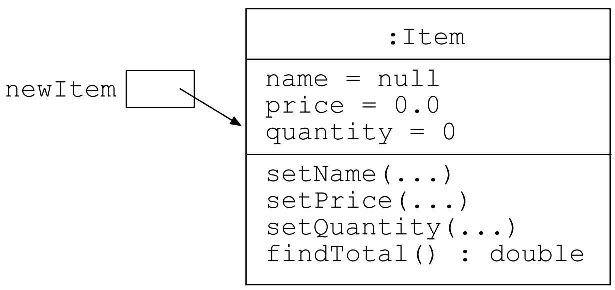


Figure 5.63 An Item object is created and its reference is assigned to variable newItem

Then, the `addItem()` method of the `PurchaseOrder` object sets the attributes of the `Item` object based on the values stored in the parameters, `name`, `price` and `quantity`:

```
newItem.setName(name);
newItem.setPrice(price);
newItem.setQuantity(quantity);
```

The scenario is visualized in Figure 5.64.

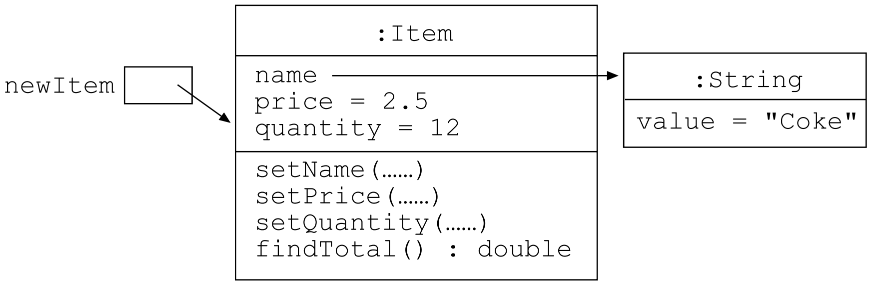


Figure 5.64 The attribute name of the Item object is updated to refer to a String object

Finally, the contents of the local variable `newItem` are stored in the array object referred to by the attribute `items`. The total number of items in the array object, the attribute `itemCount`, is increased by one after executing the following statement:

```
items[itemCount++] = newItem;
```

At this moment, the scenario is visualized in Figure 5.65.

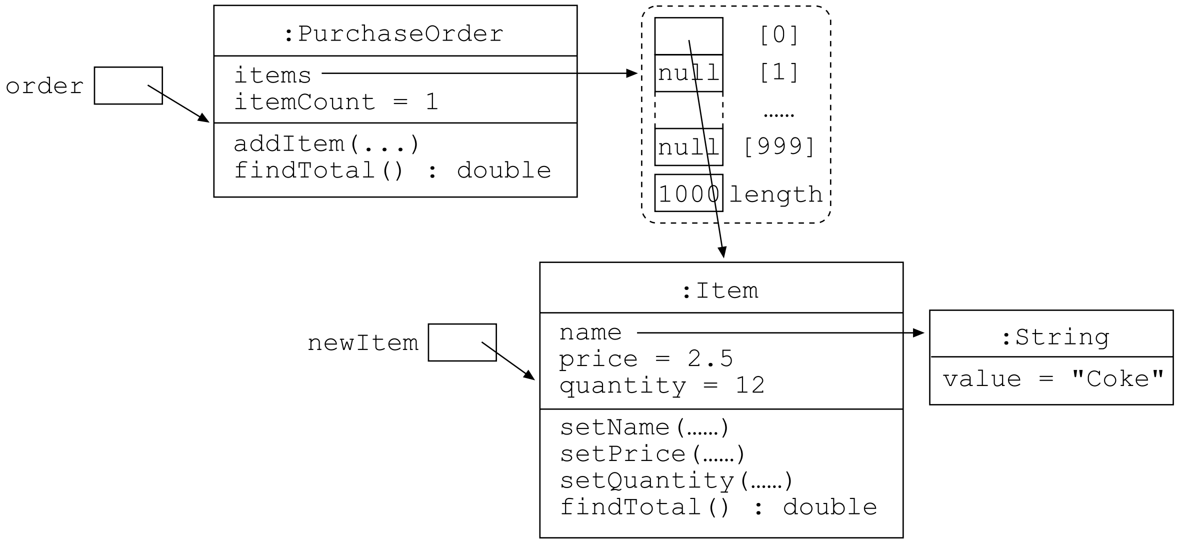


Figure 5.65 Storing the reference of the Item object by the array object of the PurchaseOrder object

The method `addItem()` is terminated and the local variable `newItem` is automatically removed from the JVM memory. When the flow of control is returned to the `main()` method, the local variable `newItem` declared in the `addItem()` method is removed automatically, as shown in Figure 5.66.

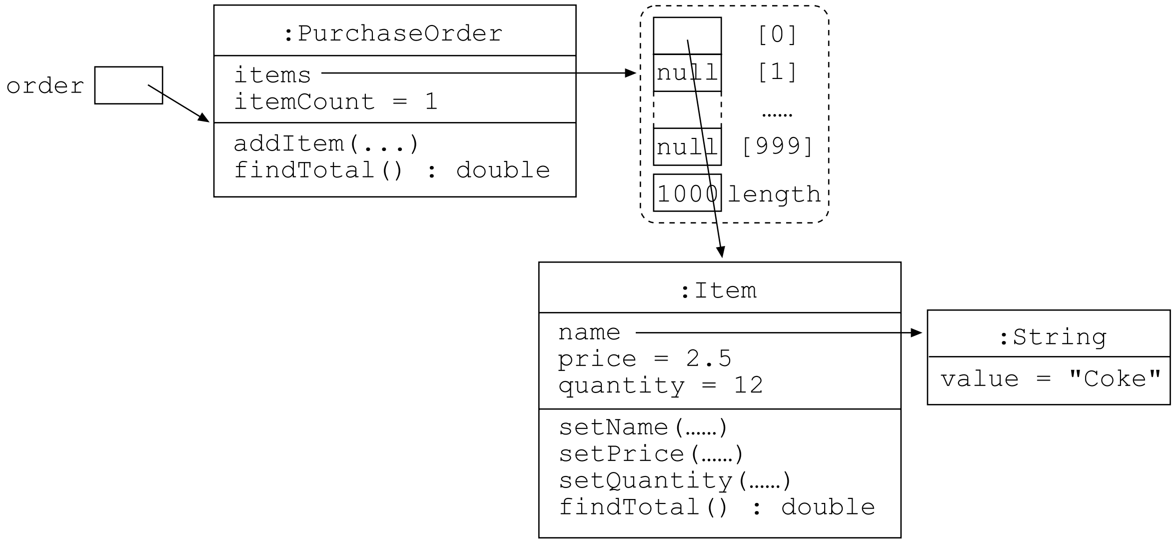


Figure 5.66 The scenario of the objects when the flow of control is returned to the main() method

In the `main()` method of the `Cashier` class, the `addItem()` method of the `PurchaseOrder` object is called twice more:

```
order.addItem("Milk", 6.0, 1);
order.addItem("Egg", 0.5, 18);
```

The operations are similar to the previous method call. After the three method calls to the `addItem()` method, you can visualize the scenario in Figure 5.67.

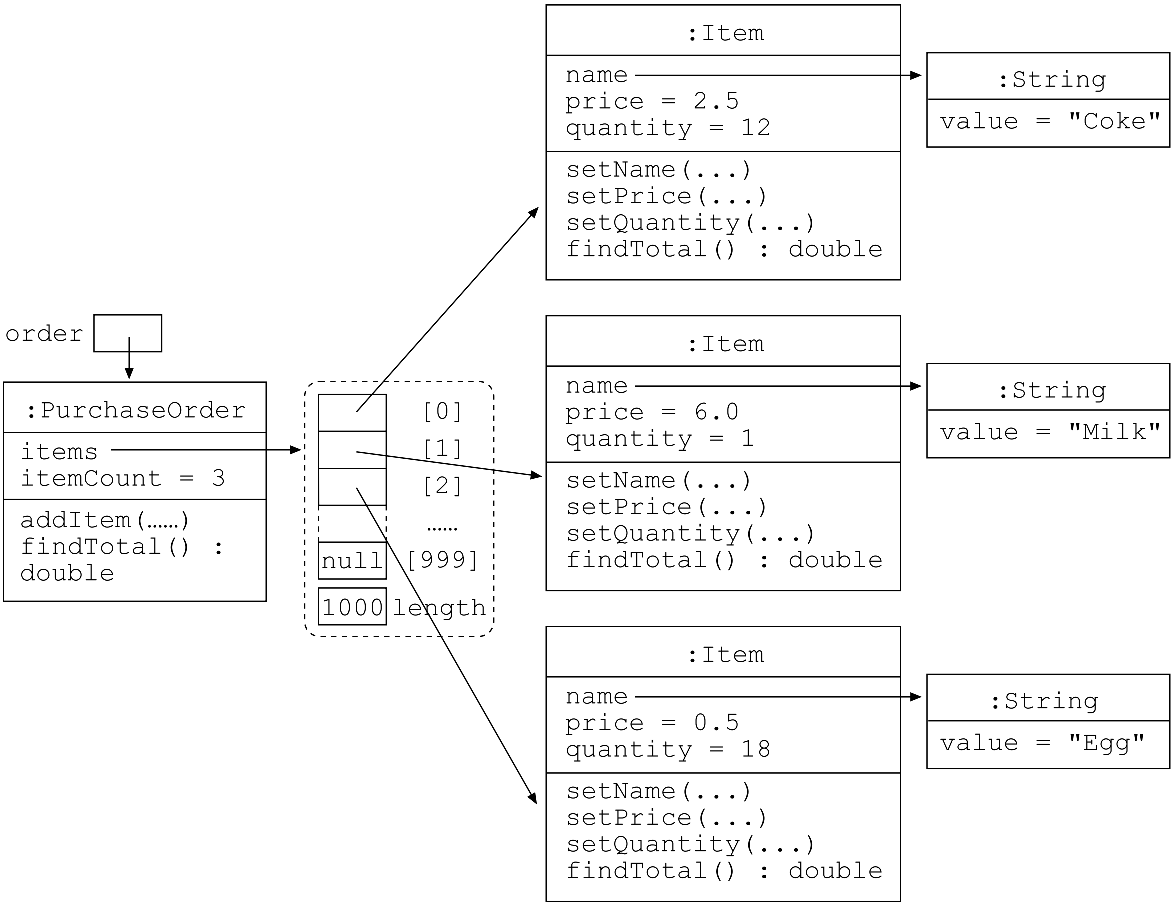


Figure 5.67 The scenario after the addItem() method is executed three times

If you cannot figure out how the above scenario is set up, please review the above description again to make sure that you know what is going on.

The last statement of the `main()` method in `Cashier` class

```
System.out.println("Total price = " + order.findTotal());
```

displays the total price to the screen and the total price is obtained by calling the `findTotal()` method of the `PurchaseOrder` object that is referred to by the local variable `order`. According to the `findTotal()` method of the `PurchaseOrder` class

```
public double findTotal() {
    double sum = 0.0;
    for (int i=0; i < itemCount; i++) {
        sum += items[i].findTotal();
    }
    return sum;
}
```

it subsequently calls the `findTotal()` method of the `Item` objects that are associated by the array object referred to by the attribute `items` in a `for` loop. The method call sequence is shown in Figure 5.68.

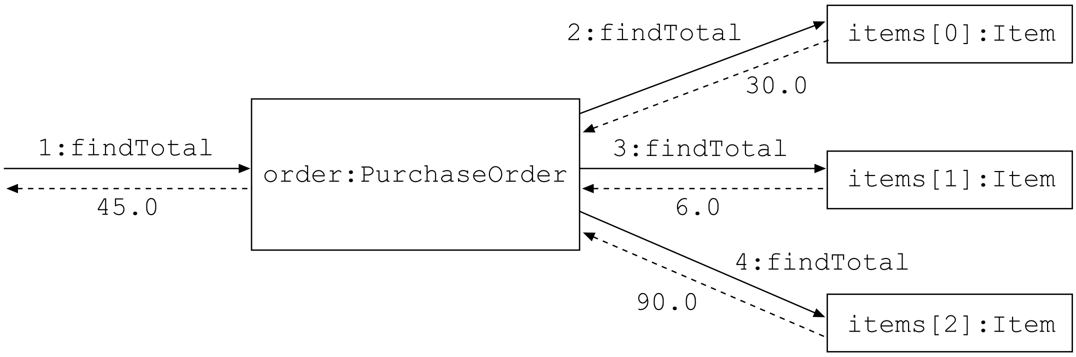


Figure 5.68 The sequence of message flows and return values involved in the findTotal() method

As a result, the total price, `45.0`, is determined by adding all the subtotals returned from the `Item` objects.

Please use the following self-test to examine your understanding of using an array of objects.

Self-test 5.6

Modify the definitions of classes Item and PurchaseOrder to display a purchase receipt, such as:

```
Name     Price    Qty      Sub-total
Coke     2.5      12       30.0
Milk     6.0      1        6.0
Egg      0.5      18       9.0

Total    45.0
```

Hint: Add a `print()` method to the `Item` class that displays the information of a single Item object on the screen. The `PurchaseOrder` class also needs a `print()` method that calls the `print()` method of each Item object that is referred to by the array object and finally displays the total price. In the `main()` method of the `Cashier` class, the `print()` method of the `PurchaseOrder` object is called to display the purchase receipt.

Manipulating arrays of objects

In the discussion of array size in this unit, we mentioned that the way to use an array of a larger size is to create a new array object with a larger size and copy the contents from the existing one to the new one. The new array object is kept and the existing one is neglected — it will be removed automatically from the JVM memory at a later time.

Arrays of objects that are arrays of non-primitive element types are also non-resizable. Therefore, to use an array object of non-primitive types with a larger array size, it is necessary to create a new array object and copy the array elements. You have to bear in mind the array elements of non-primitive type store the references of the objects. As a result, if you use a `for` loop to copy the array element contents, the array elements of both arrays are referring to the same set of objects.

For example, for the following program segment:

```
TicketCounter[] counters = new TicketCounter[2];
counters[0] = new TicketCounter();
counters[1] = new TicketCounter();
```

Two `TicketCounter` objects are created and are referred to by the array object that is referred by variable `counters`. If the following statements are executed afterwards

```
TicketCounter[] temp = new TicketCounter[counter.length];
for (int i=0; i < counters.length; i++) {
    temp[i] = counters[i];
}
```

the array objects referred to by both variables `temp` and `counters` refer to the same two `TicketCounter` objects. When the above program segment is executed, no new `TicketCounter` object is created. Figure 5.69 can help you understand the scenario:

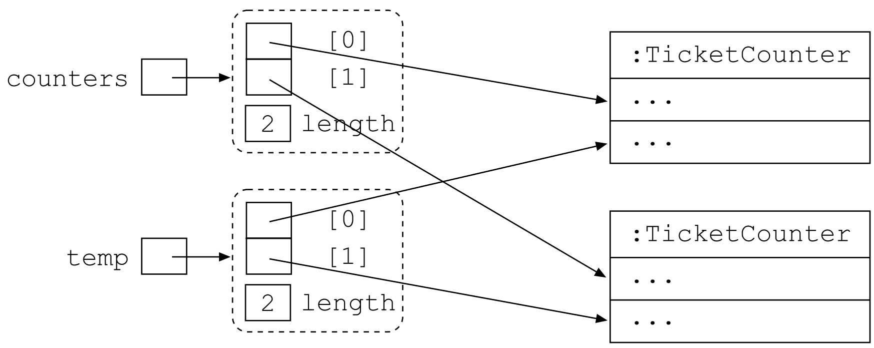


Figure 5.69 Two array objects referring to the same set of TicketCounter object

As a result, the following two statements are equivalent:

```
temp[i].increase();
counters[i].increase();
```

If you want to have another set of `TicketCounter` objects with the same reading, it is necessary to create two new `TicketCounter` objects and set their reading accordingly. That is:

```
for (int i=0; i < counters.length; i++) {
    temp[i] = new TicketCounter();
    temp[i].setReading(counters[i].getReading());
}
```

In the above program segment, two more `TicketCounter` objects are created and the values of their attribute `reading` are set according to the existing `TicketCounter` objects that are associated by the array object referred to by the variable `counters`. Suppose that the readings of the `TicketCounter` objects referred by `counters[0]` and `counters[1]` are `0` and `1` respectively. After the above program segment is executed, there are four `TicketCounter` objects in total. The scenario is shown in Figure 5.70.

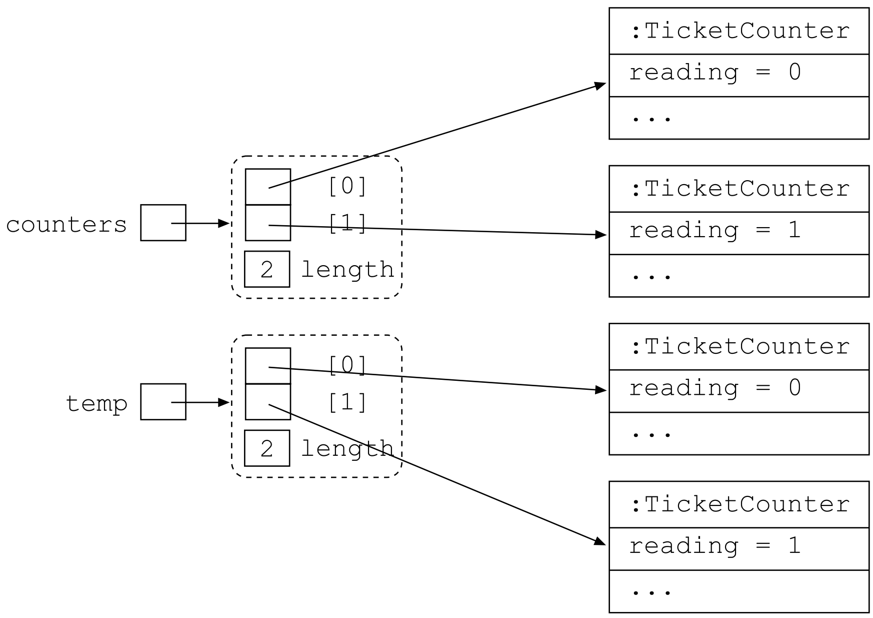


Figure 5.70 Two array objects referring to two sets of different TicketCounter objects

If you need to copy the array element contents of non-primitive type, please consider whether you want to have new objects or use the same set of objects. The choice between the two approaches depends on the requirements of your program. If array size is the only concern, creating new objects is not necessary.

Manipulating arrays with methods

Since the first Java program, we have been presenting an array object but we have not used it yet. Have you noticed that the definition of the `main()` method that we use to set up the execution environment is:

```
public static void main(String args[]) {
    ......
}
```

The type of the parameter, `args`, of the `main()` method is an array of `String`. As we mentioned, the pair of square brackets for declaring an array variable can be placed on each side of the variable name. The `main()` method can be written as:

```
public static void main(String[] args) {
    ......
}
```

They are equivalent and it is up to you to choose your own convention.

Actually, you can execute a program with the following format:

`java` *Program Program-parameter-list*

For example:

```
java TestArgs First Second Third "Fourth Item"
```

When you execute a Java program in the above format, the JVM is started and it reads the class definition (the `.class` file) of `TestArgs` from the drive. The JVM will prepare an array object that refers to `String` objects and each corresponds to a program parameter. The reference of this array object is passed to the `main()` method via the parameter and is named `args` in our example. If there is no program parameter in the command, the size of the array object that is passed to the `main()` method is zero, as shown in Figure 5.71.

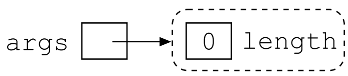


Figure 5.71 The array object referred by the parameter of main() method if no program parameter is given

If the `TestArgs` program is started with the following command,

```
java TestArgs First Second Third "Fourth Item"
```

the program parameters after the class name `TestArgs` are referred by array elements and the array object is passed to the `main()` method as shown in Figure 5.72.

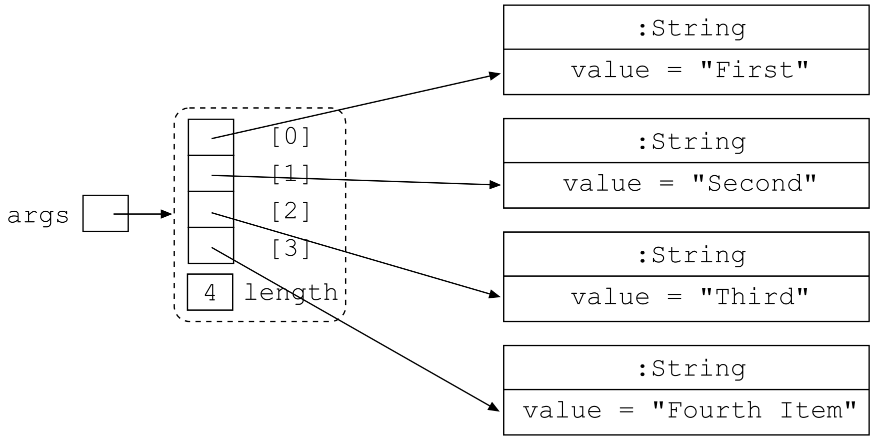


Figure 5.72 The array object referring to the String objects of program parameters

The entries are usually separated by spaces. If an entry contains a space, it must be enclosed in a pair of double quotation marks (`"`).

With such a mechanism, the definition of class `TestArgs` is written to show the program parameters as in Figure 5.73.

```
// Definition of class TestArgs
public class TestArgs {
    public static void main(String args[]) {
        // A loop to display the entries
        for (int i=0; i < args.length; i++) {
            System.out.println(args[i]);
        }
    }
}
```

Figure 5.73 TestArgs.java

If you compile the program and execute it with the command,

```
java TestArgs First Second Third "Fourth Item"
```

you will get the following output:

```
First
Second
Third
Fourth Item
```

You can make use of this mechanism to set up an environment for subsequent program execution. For example, you can create a new `TestBinarySearcher2` that is based on the `TestBinarySearcher` class definition, and the numbers in the array to be searched are obtained from the program parameters. The definition of the `TestBinarySearcher2` is shown in Figure 5.74.

```
// Import statement for JOptionPane
import javax.swing.JOptionPane;

// Definition of class TestBinarySearcher2
public class TestBinarySearcher2 {
    // Main executive method
    public static void main(String args[]) {
        // Create a BinarySearcher object and is referred by variable
        // searcher
        BinarySearcher searcher = new BinarySearcher();

        // Store the numbers in the BinarySearcher object
        for (int i=0; i < args.length; i++) {
            int number = Integer.parseInt(args[i]);
            searcher.storeNumber(number);
        }

        // Get a number from user
        String inputNumber = JOptionPane.showInputDialog(
            "Please enter a number");
        int number = Integer.parseInt(inputNumber);

        // Determine whether the input number is in the list of
        // stored numbers, and display message accordingly
        if (searcher.contains(number)) {
            JOptionPane.showMessageDialog(null,
                "The number (" + number + ") is found");
        }
        else {
            JOptionPane.showMessageDialog(null,
                "The number (" + number + ") cannot be found");
        }

        // Terminate the program explicitly
        System.exit(0);
    }
}
```

Figure 5.74 TestBinarySearcher2.java

The `for` loop in the `main()` method derives the numbers from the program parameters and requests the `BinarySearcher` object to store them. Therefore, to search a number by binary searching in a number list of `1`, `23`, `43`, `56`, `76`, `84`, `93` and `97`, as defined in the `TestBinarySearcher` class, execute the TestBinarySearcher2 with parameters as follows:

```
java TestBinarySearcher2 1 23 43 56 76 84 93 97
```

It is more convenient to use the `TestBinarySeracher2` rather than the `TestBinarySearcher` because it is not necessary to recompile the program for different number lists but just to change the program parameters.

Instead of requesting the `BinarySearcher` object to store the numbers one at a time, it is possible to pass the reference of an array object of element type `int` to it using a single method call. A modified version of `BinarySearcher`, called `BinarySearcher2`, is defined with one more method, `setNumbers()`, that can accept a reference of an array object with array element type of `int`. The definition of the method is:

```
public void setNumbers(int[] theNumbers) {
    numbers = theNumbers;
    total = theNumbers.length;
}
```

The type of the parameter, `theNumber`, is array of `int`. When such a method is called, the reference of an array object of element type `int` is expected. The original array object that attribute `numbers` was referring to is lost. The complete definition of the class `BinarySearcher2` is shown in Figure 5.75.

```
// Definition of the class BinarySearcher2
public class BinarySearcher2 {
    // Attributes
    private int numbers[] = new int[1000];
    private int total;

    // Behaviours

    // Update the attribute numbers to the reference of another
    // array of int via parameter. It is assumed that all elements
    // of the passed array object are all involved in searching
    public void setNumbers(int[] theNumbers) {
        numbers = theNumbers;
        total = theNumbers.length;
    }

    // Store the number to the array
    public void storeNumber(int number) {
        if (total < numbers.length) {
            numbers[total++] = number;
        }
        else {
            System.out.println("Too many numbers");
        }
    }

    // Determine whether a number can be found in the array by
    // binary searching
    public boolean contains(int number) {
        // Declare the initialise the lower searching bound
        int lowerBound = 0;
        // Declare and initialise the upper searching bound
        int upperBound = total - 1;

        // Repeat while the lower searching bound is less than the
        // the upper searching bound
        while (lowerBound < upperBound) {
            // Determine the middle subscript
            int middle = (lowerBound + upperBound) / 2;

            // Update either lower or upper searching bound
            if (number <= numbers[middle]) {
                upperBound = middle;
            }
            else {
                lowerBound = middle + 1;
            }
        }

        // Return the result whether the number in the searching
        // scope is the target number
        return number == numbers[lowerBound];
    }
}
```

Figure 5.75 BinarySearcher2.java

To use a `BinarySearcher2` object, either the `setNumbers()` method or the `storeNumber()` method should be used to supply the numbers to be searched. Furthermore, once the `setNumbers()` method is called to supply the `BinarySearcher2` object with the reference of an array object, the `BinarySearcher2` object cannot store any number by a subsequent method that calls to its `storeNumber()` method.

For example, the following program segment creates a `BinarySearcher2` object and an array object with eight numbers. Then, the method `setNumbers()` of `BinarySearcher2` object is used to update the array object that the `BinarySearcher2` is using.

```
public static void main(String args[]) {
    BinarySearcher2 searcher = new BinarySearcher2();
    int[] numberList = new int[8];
    numberList[0] = 1;
    numberList[1] = 23;
    numberList[2] = 43;
    numberList[3] = 56;
    numberList[4] = 76;
    numberList[5] = 84;
    numberList[6] = 93;
    numberList[7] = 97;

    searcher.setNumbers(numberList);

    searcher.contains(84);
                   }
```

When the first statement is executed, a `BinarySearcher2` object is created and is referred to by local variable `searcher` as shown in Figure 5.76.

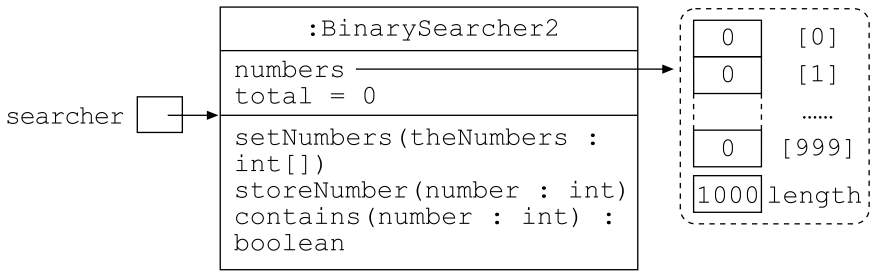


Figure 5.76 The initial scenario when the BinarySearcher2 object is created

Then, the following program segment prepares an array object and sets the array elements accordingly:

```
int[] numberList = new int[8];
numberList[0] = 1;
numberList[1] = 23;
numberList[2] = 43;
numberList[3] = 56;
numberList[4] = 76;
numberList[5] = 84;
numberList[6] = 93;
numberList[7] = 97;
```

It sets up the scenario in the computer, as shown in Figure 5.77.

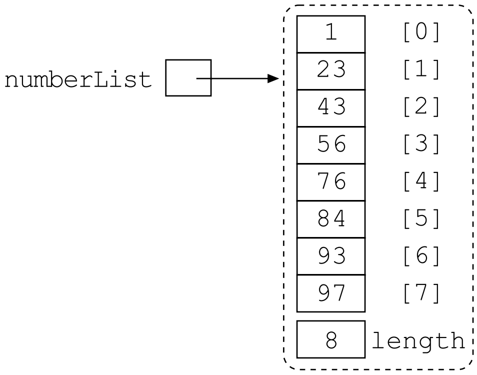


Figure 5.77 The array object created and initialized by executing the sequence of statements

The last statement,

```
searcher.setNumbers(numberList);
```

passes the value of the local variable `numberList`, which is the reference of the array object with eight numbers, to the `BinarySearcher2` object that is referred to by the local variable `searcher`. The scenario is visualized in Figure 5.78.

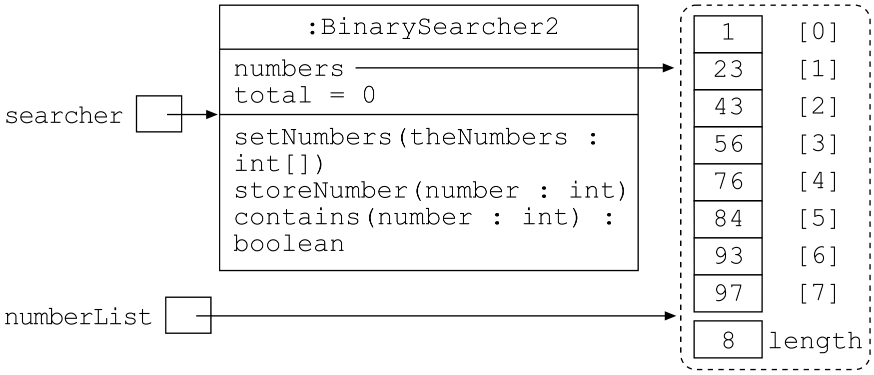


Figure 5.78 The BinarySearcher2 object is updated to refer to the array object referred by the variable numberList in main() method

The array object that is referred by the attribute `numbers` of the `BinarySearcher2` object is ready. You can now use the `contains()` method to determine whether a number is included in the numbers, such as:

```
searcher.contains(84);
```

Another important observation from the above diagram is that the local variable `numberList` in the method `main()` and the attribute `numbers` of the `BinarySearcher2` object are both referring to the same array object. Therefore, if a method of the `BinarySearcher2` object modifies any array element value that is referred to by the attribute `numbers`, the `main()` method can get the updated array element value with the local variable `numberList`.

If the array elements are predefined at compile time, instead of creating the array object and setting its array elements one by one, you can use a shorthand way to do this. For example, the following program segment

```
int[] numberList = new int[8];
numberList[0] = 1;
numberList[1] = 23;
numberList[2] = 43;
numberList[3] = 56;
numberList[4] = 76;
numberList[5] = 84;
numberList[6] = 93;
numberList[7] = 97;
```

is equivalent to the following statement:

```
int[] numberList = {1, 23, 43, 56, 76, 84, 93, 97};
```

The pair of curly brackets (`{` and `}`) encloses the array element values of the array object to be created. It is known as the array initializer. You can use a similar format to create an array object of other primitive types. For example,

```
boolean[] isEven = {true, false, true, false, true, false};
```

or

```
double[] studentMarks = {50.0, 60.0, 70.0};
```

You can therefore further simplify the previous main() method:

```
public static void main(String args[]) {
    BinarySearcher2 searcher = new BinarySearcher2();
    int[] numberList = {1, 23, 43, 56, 76, 84, 93, 97};

    searcher.setNumbers(numberList);

    searcher.contains(84);
}
```

Array initializer can be used for non-primitive arrays as well. For example, if it is necessary to create an array object that refers to two `TicketCounter` objects, instead of the following program segment,

```
TicketCounter[] counters = new TicketCounter[2];
counters[0] = new TicketCounter();
counters[1] = new TicketCounter();
```

the following statement is an equivalent shorthand:

```
TicketCounter[] counters = {
    new TicketCounter(),
    new TicketCounter()
};
```

The advantage of using array initializer is that it is not necessary to count the number of array elements. The compiler automatically creates an array object of suitable array type, and the necessary array size is determined by the number of items enclosed in the pair of curly brackets.

We mentioned that the Java compiler could automatically create `String` objects. As a result, it is possible to use array initializer for an array of `String` without using the keyword `new`. That is, the following statement is valid in the Java programming language:

```
String[] messages = {
    "Good morning",
    "Good afternoon",
    "Good evening"
};
```

The above statement declares a variable `message` of type array of `String`, and an array object with three elements of type `String` that are referring to the `String` objects with the contents `"Good morning"`, `"Good afternoon"` and `"Good evening"` respectively. It is equivalent to the following statements:

```
String[] messages = new String[3];
messages[0] = "Good morning";
messages[1] = "Good afternoon";
messages[2] = "Good evening";
```

Self-test 5.7

Write definitions for `TestLinearSearcher2` and `LinearSearcher2` classes based on the definitions of `TestLinearSearcher` and `LinearSearcher` classes, so that the `main()` method of `TestLinearSearcher2` creates an array of `int` with array element values according to the program parameters. Add a new method `setNumbers()` to the `LinearSearcher2` class so that the `main()` method of `TestLinearSearcher2` can pass the reference of the array object to the `LinearSearcher2` object. Then, a dialog is used to get a number from the user and the `LinearSearcher2` object searches the number in the array.

Summary

Arrays are indispensable tools in the Java programming language. They are mainly used for consolidating a collection of data or objects of the same type. There are basically three steps in using arrays in the Java programming language:

1. Declare a variable of type array. The element type can be primitive or non-primitive. The format is:

   type`[]` *variable-name*;

   or

   type variable-name`[]`;

2. Create an array object by specifying the type and the number of array elements required. The format is:

   `new` *type*`[`*array-size*`]`

   The reference of the newly created array is usually assigned to the variable of the corresponding array type. The first step and the second step can be done in a single statement as:

   `type` *variable-name*`[] = new` *type*`[`*array-size*`];`

3. Access an array element by specifying the subscript, or manipulate the entire array object with its reference.

The valid subscript of an array of size `n` ranges from `0` to `n – 1`. Accessing an array element with an invalid subscript that is out of the valid range generates a runtime error, and the program execution is abnormally terminated.

An array variable can be declared without specifying the array size, and it can refer to an array object of any size. To determine the size — the number of array elements — of an array object, it is possible to use its `length` attribute. Often a for loop with the attribute length is used to access each array element.

The array size is determined when the array object is created and cannot be changed. Therefore, if an array of a larger size is required, it is necessary to create another array object with a larger size and copy the array element contents from the existing one to the new one.

The array elements of an array object of primitive type store the actual primitive values; the array elements of an array object of non-primitive type store the *references* of the objects. If the array elements are predefined at compile time, you can use an array initializer as a shorthand form to create the array object.

While manipulating the array elements, you can make use of the `++` operator and `--` operator. The operators can be placed before or after a variable. If they are placed before a variable, they are the pre-increment operator and pre-decrement operator, where the value of the variable are increased or decreased by one, and the resultant value is used. If they are placed after a variable, they are the post-increment operator and post-decrement operator. Then the value of the variable is used and it is increased or decreased afterwards. The use of these two operators in the above-mentioned ways is tricky and complicates the statements. Therefore, you should know how they work so that you can read the class definitions written by other programmers, but you should use these two operators in simple expressions or statements only.

With the knowledge you learned of using arrays, we discussed an important operation that is common in many programming tasks — searching. The two common searching methods are linear searching and binary searching. Linear searching is easier to implement, as it accesses each array element sequentially. It is not an effective way to search for a number in a huge amount of data. In contrast, binary searching is much faster, especially if the amount of data to be searched is huge. The shortcoming of binary searching is that the data in the array must be sorted.

From now on, whenever your application has to handle a collection of data of the same type, no matter whether they are of primitive type or non-primitive type, you should consider using an array as a means to manipulate them.

Suggested answers to self-test questions

Self-test 5.1

1 The following is the class definition of the `IntegerStack` class that can store at most ten elements. (The modifications made to the class `IntegerStack3` are highlighted.)

```
// Definition of the class IntegerStack (Self-test 5.1)
public class IntegerStack {
    // Attributes
    // The storage for the numbers in the stack using an array with
    // 10 elements
    private int[] storage = new int[10];
    // The attribute for the subscript of the element that can store
    // the newly added number
    private int top = 0;

    // Behaviours

    // The behaviour to push a new number
    public void push(int number) {
        // Verify whether all array elements have been used
        if (top < 10) {
            // Store the number and increase the subscript for
            // storing the next number afterwards
            storage[top++] = number;
        }
        else {
            // Show message to prompt the user
            System.out.println("The stack is full");
        }
    }

    // The behaviour to pop the last number
    public int pop() {
        // A local variable result is declared to store the value
        // to be returned
        int result = -1;

        // Verify whether the stack is empty
        if (top > 0) {
            // Decrease the subscript for the last number and
            // Store the last number to local variable result afterwards
            result = storage[--top];
        }
        else {
            // Show message to prompt the user
            System.out.println("The stack is empty");
        }
        return result;
    }
}
```

2 **IntegerQueue.java**

```
// Definition of the class IntegerQueue
public class IntegerQueue {
    // Attributes
    // The storage for the numbers in the queue using an array with
    // 6 elements
    private int[] storage = new int[6];
    // The attribute indicates the subscript of array element to be
    // returned
    private int top = 0;
    // The attribute indicates the subscript of array element to
    // store new number
    private int total;

    // Behaviours

    // The behaviour to add a new number to the queue
    public void enqueue(int number) {
        // Verify whether all array elements have been used
        if (total < 6) {
            // Store the number and increase the subscript for
            // storing the next number afterwards
            storage[total++] = number;
        }
        else {
            // Show message to prompt the user
            System.out.println("The queue is full");
        }
    }

    // The behaviour to remove the first available number
    public int dequeue() {
        // A local variable result is declared to store the value
        // to be returned
        int result = -1;

        // Verify whether the stack is empty
        if (top < total) {
            // Store the first available number to local variable result
            result = storage[top++];
        }
        else {
            // Show message to prompt the user
            System.out.println("The queue is empty");
        }
        return result;
    }
}
```

Self-test 5.2

The definition of the `IntegerQueue` is similar to the definition for Self-test 5.1, except the `enqueue()` method. Therefore, only the `enqueue()` method is shown here. The complete definition of the `IntegerQueue` for this self-test can be found in the CD-ROM.

```
// The behaviour to add a new number to the queue
public void enqueue(int number) {
    // Verify whether all array elements have been used
    if (total < storage.length) {
        // Store the number and increase the subscript for
        // storing the next number afterwards
        storage[total++] = number;
    }
    else {
        // Show message to prompt the user
        System.out.println("The stack is full");
    }
}
```

Self-test 5.3

The dramatic increase of execution time is due to the fact that when a number is pushed to the array object of the `IntegerStack5` object and the all array elements are used, it is necessary to create a new array object, and array elements need replicating. Then, the original array object is neglected and the JVM needs to remove it from the memory. All these operations contribute to the increased execution times.

To reduce the execution time, when all array elements of an array object are used, it is possible to create an array object of a much larger size than the original one. For example, the statement that creates a new larger array object can be modified to be

```
int[] newArray = new int[storage.length + 100];
```

or even

```
int[] newArray = new int[storage.length + 1000];
```

Then, the frequency of creating new array objects and copying array elements can be greatly reduced, and hence the execution times are reduced.

If a software application works properly but the comment is made that its performance is unacceptable, software developers usually investigate the underlying performance bottleneck. Often, it is possible to modify some settings or implementations to improve the execution performance. Such a process is known as *tuning*.

Self-test 5.4

OddEvenCounter.java

```
// Definition of class OddEvenCounter
public class OddEvenCounter {
    // Attributes
    private int numbers[] = new int[1000];
    private int total;

    // Behaviours

    // Store the number to the array
    public void storeNumber(int number) {
        // Verify whether all array elements are used
        if (total < numbers.length) {
            // If there is still room, store the number
            // and update the amount of number stored
            numbers[total++] = number;
        }
        else {
            // Show message to user
            System.out.println("Too many numbers");
        }
    }

    // Count the odd numbers in the array
    public int countOdd() {
        // Declare a local variable for counting odd number
        int count = 0;
        // Verify each array element
        for (int i = 0; i < total; i++) {
            if (numbers[i] % 2 != 0) {
                count++;
            }
        }
        // Return the sum
        return count;
    }

    // Count the even numbers in the array
    public int countEven() {
        // Declare a local variable for counting even number
        int count = 0;
        // Verify each array element
        for (int i = 0; i < total; i++) {
            if (numbers[i] % 2 == 0) {
                count++;
            }
        }
        // Return the sum
        return count;
    }
}
```

TestOddEvenCounter.java

```
// The class definition of TestOddEvenCounter for setting up the
// environment and test the OddEvenCounter object
public class TestOddEvenCounter {

    // The main executive method
    public static void main(String args[]) {
        // Create an OddEvenCounter object and is referred by
        // local variable counter
        OddEvenCounter counter = new OddEvenCounter();

        // Sending messages with numbers to the OddEvenCounter
        // object to be stored in the array
        counter.storeNumber(1);
        counter.storeNumber(4);
        counter.storeNumber(7);
        counter.storeNumber(10);
        counter.storeNumber(12);
        counter.storeNumber(16);
        counter.storeNumber(19);

        // Display the result to the screen
        System.out.println("Number of odd numbers = " +
            counter.countOdd());
        System.out.println("Number of even numbers = " +
            counter.countEven());
    }
}
```

Feedback on activities

Activity 5.1

To display the values of upper searching bound and lower searching bound when an iteration starts and ends, the `contains()` method is modified to be:

```
// Determine whether a number can be found in the array by
// binary searching
public boolean contains(int target) {
    // Declare the initialise the lower searching bound
    int lowerBound = 0;
    // Declare and initialise the upper searching bound
    int upperBound = total - 1;

    // Repeat while the lower searching bound is less than the
    // the upper searching bound
    while (lowerBound < upperBound) {
        System.out.println("Loop-start: " +
            lowerBound + "-" + upperBound);
        // Determine the middle subscript
        int middle = (lowerBound + upperBound) / 2;

        // Update either lower or upper searching bound
        if (target <= numbers[middle]) {
            upperBound = middle;
        }
        else {
            lowerBound = middle + 1;
        }
        System.out.println("Loop-end: " +
            lowerBound + "-" + upperBound);
    }

    // Return the result whether the number in the searching
    // scope is the target number
    return target == numbers[lowerBound];
}
```

Compile it and execute the `TestBinarySearcher` program. The following outputs are shown:

```
Loop-start: 0-7
Loop-end: 4-7
Loop-start: 4-7
Loop-end: 4-5
Loop-start: 4-5
Loop-end: 5-5
```

Self-test 5.5

CountNumbers.java

```
// Import statement for JOptionPane
import javax.swing.JOptionPane;

// Definition of class CountNumbers
public class CountNumbers {

    // Main executive method
    public static void main(String args[]) {
        // Create an array of 10 TicketCounter objects
        TicketCounter[] counters = new TicketCounter[10];
        for (int i=0; i < counters.length; i++) {
            counters[i] = new TicketCounter();
        }

        // Declare a local variable for storing the value
        // obtained from user
        int number;

        // A do/while loop repeatedly prompts the user for numbers
        // and call the increase() method of corresponding
        // TicketCounter.
        do {
            // Get a number from the user
            String inputNumber = JOptionPane.showInputDialog(
                "Input a number [-1 to quit]");
            number = Integer.parseInt(inputNumber);
            // If the number is not -1, call the increase() method of
            // the corresponding TicketCounter
            if (number != -1) {
                counters[number].increase();
            }
        } while (number != -1);

        // Prepare a String as output message by getting the readings
        // from the TicketCounter objects
        String output = "";
        for (int i=0; i < counters.length; i++) {
            output += i + " = " + counters[i].getReading() + "\n";
        }

        // Show the output message
        JOptionPane.showMessageDialog(null, output);

        // Terminate the program explicitly as dialog is used
        System.exit(0);
    }
}
```

Self-test 5.6

Item.java

```
// Definition of class Item
public class Item {
    // Attributes
    private String name;
    private double price;
    private int quantity;

    // Behaviours

    // Set the attribute name
    public void setName(String theName) {
        name = theName;
    }

    // Set the attribute price
    public void setPrice(double thePrice) {
        price = thePrice;
    }

    // Set the attribute quantity
    public void setQuantity(int theQuantity) {
        quantity = theQuantity;
    }

    // Find the subtotal of this item
    public double findTotal() {
        return price * quantity;
    }

    // Display this item on the screen
    public void display() {
        System.out.println(name + "\t" + price + "\t" +
            quantity + "\t" + findTotal());
    }
}
```

PurchaseOrder.java

```
// Definition of class PurchaseOrder
public class PurchaseOrder {
    // Attributes
    private Item[] items = new Item[1000];
    private int itemCount = 0;

    // Behaviours

    // To create and add a new item to the array object
    public void addItem(String name, double price, int quantity) {
        // Create a new Item object
        Item newItem = new Item();

        // Set its attributes
        newItem.setName(name);
        newItem.setPrice(price);
        newItem.setQuantity(quantity);

        // Add the Item object to the array object
        items[itemCount++] = newItem;
    }

    // Find the total of all items
    public double findTotal() {
        double sum = 0.0;
        for (int i=0; i < itemCount; i++) {
            sum += items[i].findTotal();
        }
        return sum;
    }

    // Display a purchase receipt
    public void display() {
        System.out.println("Name\tPrice\tQty\tSub-total");
        for (int i=0; i < itemCount; i++) {
            items[i].display();
        }
        System.out.println("\nTotal\t" + findTotal());
    }
}
```

Cashier.java

```
// Definition of class Cashier
public class Cashier {

    // Main executive method
    public static void main(String args[]) {
        // Create a new PurchaseOrder object to be referred by
        // the local variable order
        PurchaseOrder order = new PurchaseOrder();

        // Add the items to the purchase order
        order.addItem("Coke", 2.5, 12);
        order.addItem("Milk", 6.0, 1);
        order.addItem("Egg", 0.5, 18);

        // Display the purchase receipt
        order.display();
    }
}
```

Self-test 5.7

LinearSearcher2.java

```
// Definition of class LinearSearcher2
public class LinearSearcher2 {
    // Attributes
    private int numbers[] = new int[1000];
    private int total;

    // Behaviours

    // Update the attribute numbers to the reference of another
    // array of int via parameter. It is assumed that all elements
    // of the passed array object are all involved in searching
    public void setNumbers(int[] theNumbers) {
        numbers = theNumbers;
        total = theNumbers.length;
    }

    // Store the number to the array
    public void storeNumber(int number) {
        if (total < numbers.length) {
            numbers[total++] = number;
        }
        else {
            System.out.println("Too many numbers");
        }
    }

    // Determine whether the array contains the number
    public boolean contains(int target) {
        boolean found = false;
        for (int i=0; i < numbers.length; i++) {
            if (numbers[i] == target) {
                found = true;
            }
        }
        return found;
    }

    // Count the occurrences of the desired number in the array
    public int count(int target) {
        int total = 0;
        for (int i=0; i < numbers.length; i++) {
            if (numbers[i] == target) {
                total++;
            }
        }
        return total;
    }
}
```