MT201

Unit 6

Advanced control structures and arrays

Course team

Developer:	Herbert Shiu, Consultant
Designer:	Dr Rex G Sharman, OUHK
Coordinator:	Kelvin Lee, OUHK
Member:	Dr Vanessa Ng, OUHK

External Course Assessor

Professor Jimmy Lee, Chinese University of Hong Kong

Production

ETPU Publishing Team

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

Contents

Introduction

You started your programming in Java in *Unit 3*, where you learned how to use class definitions to define objects to be created at runtime, according to the sets of attributes and behaviours they possess. At runtime, each object models a real-world object. In *Unit 4*, you started defining object behaviours with branching and looping so that the objects can exhibit a kind of 'intelligence' and can mimic more realistic behaviours of the real-world objects. In *Unit 5*, you learned the use of arrays to consolidate collections of data of the same type so that you could manipulate them as a whole.

With the above knowledge, you can now write programs that perform some useful operations such as determining the sum, average, maximum and minimum in a list of numbers. Furthermore, you can apply a searching technique to find a data item in a collection of data of the same type.

This unit discusses some problems that you are either not able — or that are at least non-trivial — to resolve with the programming techniques you have learned so far. For example, it is tedious to write `if/else` statements that concern two conditions at the same time. Another issue is that you know how to use arrays to store a list of numbers, but then how can you store a table of numbers?

You will learn some advanced uses of looping and branching structures and arrays in this unit. By studying the applications of the new features in the examples, you will appreciate the beauty and necessity of those advanced uses.

In *Unit 5*, you learned that you could access array elements sequentially and determine the maximum and minimum values among a list of numbers. If you can repeatedly find the minimum value in a list of numbers and remove it from the list, you can imagine that the sequence of obtained numbers will be in ascending order. This is another common programming application — sorting. *Unit 6* discusses a sorting method called insertion sort.

Toward the end of *Unit 6*, we discuss another interesting and powerful programming technique — recursion. Such a programming technique enables programmers to simplify the way they handle and resolve a problem. You will find recursion interesting and challenging.

Objectives

At the end of *Unit 6*, you should be able to:

1. *Apply* complex logical expressions.
2. *Use* nested branching statements.
3. *Describe* the effect of missing `break` in `switch/case` statements.
4. *Describe* the effect of using `break` and `continue` in loops.
5. *Use* nested looping statements.
6. *Apply* sorting algorithms.
7. *Apply* multidimensional arrays.
8. *Describe* recursions.

Complex logical expressions

You learned from previous units that the statements in a program are executed in sequence, and you can change the order of execution by branching and looping statements. The conditions for the branching and looping statements you came across were all simple; for example, a condition that specifies the value of variable `a` is less than `0` (`a < 0`). What if you want to specify a condition in which age is less than 18 (`age < 18`) and gender male (`gender == 'M'`)? You will often find that you have to specify such types of condition.

A complex logical expression is made up of two or more simple logical expressions combined, using various logical operators. Each simple logical expression can be evaluated to give either `true` or `false`, and the overall result of the complex logical expression is determined according to the simple logical expression results.

In the following sections, we discuss how to construct complex logical expressions and how to determine their results.

The And operator (&&)

Let's consider the following condition presented in mathematical notation:

0 < x < 100

The meaning is to verify whether the content of the variable `x` is greater than zero and less than 100. If you want to represent the above logical expression in the Java programming language, you might wonder whether it is possible to write an `if` statement in the following format:

```
if (0 < x < 100) {
    ......
}
```

Is the expression above, in parentheses, valid in the Java programming language? To answer this question, let's investigate the expression in detail. There are two logical operators, both < operators. As the < operator is left associative, the expression is interpreted as:

```
(0 < x) < 100
```

The result of the left < operator is either `true` or `false`, which means that the expression can be either:

```
(true) < 100
```

or

```
(false) < 100
```

We mentioned in *Unit 3* that `boolean` values can be used only with the `==` operator or `!=` operator. Therefore, the above two expressions are invalid, and hence the original logical expression (`0 < x < 100`) is invalid.

Because (`0 < x < 100`) means “`x > 0` and `x < 100`”, we can use the && operator in the Java programming language to implement the conjunction “and” in the statement. As a result, two possible ways to present the expression are:

```
(x > 0) && (x < 100)
```

and

```
(0 < x) && (x < 100)
```

The && operator represents the `AND` operator in the Java programming language. The result is determined by the following truth Table 6.1.

Table 6.1 Truth table of the && operator

Conditions		Result
a	*b*	*a* && *b*
`true`	`true`	`true`
`true`	`false`	`false`
`false`	`true`	`false`
`false`	`false`	`false`

In Table 6.1, conditions *a* and *b* denote two conditions with values of either `true` or `false`. Each row in the truth table represents a possible combination of values of conditions *a* and *b*. The rightmost column shows the overall result of conditions *a* && *b*. With such a truth table, you can obtain the overall result of the condition *a* && *b* for different values of conditions *a* and *b*.

For the above-mentioned expression `(x > 0) && (x < 100)`, you can substitute conditions `a` and `b` with sub-conditions `(x > 0)` and `(x < 100)` respectively. With different values of variable `x`, the results of the conditions `(x > 0)` and `(x < 100)` can be different. It is impossible (impractical) to evaluate the overall results for all possible values of variable `x`. However, you can choose a few representative values to verify the conditions. For example, the above expression `(x > 0) && (x < 100)` involves two numeric values, `0` and `100`. Figure 6.1 shows a number line on which any point represents a particular possible value (not necessarily an integral value) of variable `x` that can help you to determine some suitable testing values.

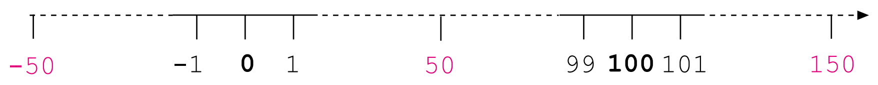


Figure 6.1 A number line with the two values 0 and 100 indicated

The number line clearly shows that all possible values of variable `x` are partitioned into three regions: `x < 0`, `0 < x < 100`, `x > 100`. Therefore, you can arbitrarily select a value from each region to be a testing value, such as, `-50`, `50` and `150`. With the two numeric values in the expression, `0` and `100`, you could test the expression with five testing values: `-50`, `0`, `50`, `100` and `150`.

By making use of Table 6.1, you can come up with Table 6.2 with different assumed values of variable `x`.

Table 6.2 The truth table for condition `(x > 0) && (x < 100)` for various values of x

Value of `x`	Result of `x > 0`	Result of `x < 100`	Result of `(x > 0) && (x < 100)`
`-50`	`(-50>0)` `false`	`(-50<100)` `true`	`false`
`0`	`(0>0)` `false`	`(0<100)` `true`	`false`
`50`	`(50>0)` `true`	`(50<100)` `true`	`true`
`100`	`(100>0)` `true`	`(100<100)` `false`	`false`
`150`	`(150>0)` `true`	`(150<100)` `false`	`false`

This truth table helps you investigate the relationship between the results of sub-conditions and the overall result. Furthermore, it helps you verify whether a condition is properly constructed. For example, if the above condition is modified to be `(x < 0) && (x >100)`, a truth table with different values of variable `x` can be obtained, as in Table 6.3.

Table 6.3 The truth table for condition `(x < 0) && (x > 100)` for various values of x

Value of `x`	Result of `x < 0`	Result of `x > 100`	Result of `(x < 0) && (x > 100)`
`-50`	`(-50<0)` `true`	`(-50>100)` `false`	`false`
`0`	`(0<0)` `false`	`(0>100)` `false`	`false`
`50`	`(50<0)` `false`	`(50>100)` `false`	`false`
`100`	`(100<0)` `false`	`(100>100)` `false`	`false`
`150`	`(150<0)` `false`	`(150>100)` `true`	`false`

From Table 6.3, the overall result of `(x < 0) && (x > 100)` is always `false` for all testing values. It indicates that there might be a problem with the condition, as it is obvious that there is no value less than zero, which is a negative value, that can be greater than 100. Therefore, if the condition `(x < 0) && (x > 100)` is used as the condition in a `while` loop or a `for` loop, the loop will terminate at once and the loop body will not be executed at all. However, if a condition that is always `true` is used as the condition for a loop construct (you'll see an example soon), the loop will repeat forever and it is, for all practical purposes, an infinite loop.

Sometimes, a truth table may hint that a `boolean` expression can be simplified. For example, with respect to a `boolean` expression `(x < 0) && (x < 100)`, a truth table can be obtained as shown in Table 6.4.

Table 6.4 The truth table for condition `(x < 0) && (x < 100)` for various values of x

Value of `x`	Result of `x < 0`	Result of `x < 100`	Result of `(x < 0) && (x < 100)`
`-50`	`(-50<0)   true`	`(-50<100)   true`	`true`
`0`	`(0<0)   false`	`(0<100)   true`	`false`
`50`	`(50<0)   false`	`(50<100)   true`	`false`
`100`	`(100<0)   false`	`(100<100)   false`	`false`
`150`	`(150<0)   false`	`(150<100)   false`	`false`

The results of the entire `boolean` expression are the same as the sub-expression, `(x < 0)`. Then, you should ask yourself whether the sub-expression `(x < 100)` is redundant — the entire `boolean` expression can be simplified to be `(x < 0)`. In this example, it is possible to simplify the complex `boolean` expression to be just `(x < 0)`.

The Or operator (||)

To verify whether the value of a variable `x` is less than `0` or is greater than `100`, there are two expressions:

```
x < 0
x > 100
```

In English, the condition is "`x < 0`" or "`x > 100`". In the Java programming language, it is written as:

```
(x < 0) || (x > 100)
```

The `||` operator represents the logical OR operator. The `boolean` expressions on both sides of the `||` operator can be `true` or `false`, and the result of the `||` operator is obtained based on Table 6.5.

Table 6.5 Truth table of the || operator

Conditions		Result
a	*b*	*a* `||` *b*
`true`	`true`	`true`
`true`	`false`	`true`
`false`	`true`	`true`
`false`	`false`	`false`

By making use of the truth table shown in Table 6.5, you can evaluate the overall result of the `boolean` expression `(x < 0) || (x > 100)` with suitably selected testing values as shown in Table 6.6.

Table 6.6 The truth table for condition `(x < 0) || (x > 100)` for various values of x

Value of `x`	Result of `x < 0`	Result of `x > 100`	Result of `(x < 0)||(x > 100)`
`-50`	`(-50<0)   true`	`(-50>100)  false`	`true`
`0`	`(0<0)   false`	`(0>100)   false`	`false`
`50`	`(50<0)   false`	`(50>100)   false`	`false`
`100`	`(100<0)  false`	`(100>100)  false`	`false`
`150`	`(150<0)  false`	`(150>100)   true`	`true`

You can see that the overall result of the condition is `true` if the value of `x` is less than `0` or greater than `100`.

Two relational operators, >= and <=, can be written as two sub-conditions with an `||` operator. For example:

Expression	Equivalent to
`x >= 0`	`(x > 0) || (x == 0)`
`x <= 100`	`(x < 100) || (x == 100)`

The expressions in the left column are simpler and more readable than those in the right column.

Let's consider another condition `(x > 0) || (x < 100)`. You can similarly obtain the truth table for it as shown in Table 6.7.

Table 6.7 The truth table for condition `(x > 0) || (x < 100)` for various values of x

Value of `x`	Result of `x > 0`	Result of `x < 100`	Result of `(x > 0) || (x < 100)`
`-50`	`(-50>0)   false`	`(-50<100)   true`	`true`
`0`	`(0>0)   false`	`(0<100)   true`	`true`
`50`	`(50>0)   true`	`(50<100)   true`	`true`
`100`	`(100>0)   true`	`(100<100)   false`	`true`
`150`	`(150>0)   true`	`(150<100)   false`	`true`

From Table 6.7, the result of the entire `boolean` expression is always `true`. If such a `boolean` expression were used as the condition for an *if* statement or a loop, the `if` part is always executed or the loop is an infinite loop respectively.

The relational operators take precedence over the && operator and the `||` operator. Therefore, the parentheses that enclose the sub-conditions are usually unnecessary. For example, the condition

```
((0 < x) && (x < 100))
```

can be simplified as:

```
(0 < x && x < 100)
```

If the condition is used in an `if/else` construct, `while` loop and `do/while` loop, you should notice that the syntax mandates the condition must be enclosed in parentheses, such as:

```
if (0 < x && x < 100) {
 ......
}
```

It is not necessary to enclose the sub-expressions in parentheses, but it is always a good programming practice to enclose them because it can make the expression clearer, such as:

```
if ((0 < x) && (x < 100)) {
 ......
}
```

The exclusive Or operator (^)

When a car approaches a crossroads with a traffic signal, the driver should decelerate to prepare to stop the car if either the red or the yellow traffic light is on. However, if both the red and the yellow lights of the traffic signal are on, it is not necessary to decelerate the car because such a combination of red and yellow lights indicates the green traffic light is going to be switched on very soon and the driver can safely drive through the crossroad (in Hong Kong). (If both the red light and the yellow light are off, it implies that the green light is on and the driver can keep on driving through the crossroads.)

To represent such a situation or condition in a program, you can implement a kind of program construct that looks like the following:

```
if (either the red light or yellow light is on but not both) {

  // decelerate the car for stopping
}
```

The above English-like program segment is quite difficult and tedious to represent using the && operator and `||` operator. Such a condition is known as 'exclusive or' in which the overall result is `true` if either one of the sub-conditions is `true` but not both. It is commonly denoted as the XOR operator in logic. The Java programming language provides a ^ (head) operator for such a logic operator. The truth table is shown in Table 6.8.

Table 6.8 The truth table for the ^ operator

Conditions		Result
a	*b*	*a* ^ *b*
`true`	`true`	`false`
`true`	`false`	`true`
`false`	`true`	`true`
`false`	`false`	`false`

The only difference between the usual 'or' operator and the 'exclusive or' operator is that when the expressions on both sides of the ^ operator are `true`, the overall result is `false`. Therefore, the Java programming language can implement the above English-like program segment through the following program segment

```
TrafficLight light = ...;
......
if (light.redIsOn() ^ light.yellowIsOn()) {
  // Decelerate the car for stopping
}
```

where `redIsOn()` and `yellowIsOn()` are methods of the `TrafficLight` object that returns `true` if the corresponding light is on and `false` if otherwise.

The Not operator (!)

It is possible to negate the `boolean` expression result by using the ! (exclamation mark) operator. Table 6.9 is its truth table.

Table 6.9 Truth table for the ! operator

boolean expression `x`	Result of `!x`
`true`	`false`
`false`	`true`

The type of expression that follows the `!` operator must be `boolean,` and the `!` operator negates the `boolean` result that follows it. For example, you can have an expression:

```
! (x < 0)
```

The meaning is 'the content of variable `x` is not less than zero'. (The parentheses are required. We discuss why that is so in the next section.) Please study the following table to see how it works with some assumed values of x:

Value of `x`	Result of `x < 0`	Result of `! (x < 0)`
`-5`	`true`	`false`
`0`	`false`	`true`
`5`	`false`	`true`

If you compare the above table with the one for the expression `(x >= 0)`

Value of `x`	Result of `x >= 0`
`-5`	`false`
`0`	`true`
`5`	`true`

the expression `!(x < 0)` is equivalent to `x >= 0`. For the time being, you might think this `!` is not very useful and it just makes the expression more complicated. However, you will find that this operator is sometimes required.

Self-test 6.1

Please provide the logical expressions for the following scenarios:

1 For variables `a`, `b` and `c` of type `int`, returns `true` if `c` is between the value of `a` and `b.` It returns `false` otherwise, provided that the value of `a` is less than or equal to `b`.

2 For variables `a` and `b` of type `int`, returns `true` if the sign of the product of `a` and `b` is non-negative. It returns `false` otherwise.

3 For variables `a`, `b`, `c` and `d` of type `int`, returns `true` if the values of all variables are equal. It returns `false` otherwise.

Short-circuit evaluation of logical expressions

The evaluations of the && and `||` operators in the Java programming language are considered to be short-circuit. Their characteristics are the same — the sub-conditions on both sides of the && and `||` operators are evaluated from left to right. If the result of the sub-condition on its left-hand side (the first operand) determines the result of the entire condition, the sub-condition on its right-hand side (the second operand) is not evaluated.

In some instances, the results do not change no matter whether the sub-conditions other than the first one are evaluated or not. For example:

```
if (i > 0 && j > 0) {
    ......
}
```

If the first sub-condition `i > 0` is `false`, you can observe from the truth table below that the overall result must be `false` no matter what the result of the second sub-condition `j > 0` is.

Conditions		Result
a	*b*	*a* && *b*
`true`	`true`	`true`
`true`	`false`	`false`
`false`	don’t care	`false`

The entry "don't care" in the table means we do not care about the value, which can be either `true` or `false`. Therefore, if the first sub-condition is `false`, the overall result of the above condition must be `false.` It is not necessary to evaluate the second sub-condition. However, if the first condition is evaluated to be `true`, the overall result of the entire condition depends on the second sub-condition, and it is therefore necessary to evaluate the second sub-condition.

The `||` operator features the same short-circuit evaluation. Let’s consider an `if` statement that assigns a value of `0.5` to the variable `discount` according to the value of the variable `age`.

```
double discount = 0.0;
if (age < 18 || age > 65) {
    discount = 0.5;
}
```

There are two sub-conditions in the above complex `boolean` expression. If the value of the variable `age` equals `12`, the result of the first sub-condition is `true`. Do you think it is necessary to evaluate the second sub-condition `age > 65`?

From the truth table of the `||` operator

Conditions		Result
a	*b*	*a* `||` *b*
`true`	don't care	`true`
`false`	`true`	`true`
`false`	`false`	`false`

You can see that if the first operand (the condition *a*) of the `||` operator is evaluated to be `true`, the overall result must be `true`. Then, it is not necessary to evaluate the second operand (condition *b*) of the operator, which means that the second sub-condition `age > 65` need not be evaluated if the value of the variable `age` is `12`.

The && operator and `||` operator enable us to construct even more complex logical expressions. For example, the following `if` statement can be used to assign the value `30` to the variable `day` if the value of the variable `month` is `4`, `6`, `9` or `11`:

```
if (month==4 || month==6 || month==9 || month==11) {
    day = 30;
}
```

The `||` operator is left associative, and the above condition is interpreted as:

```
(((month==4 || month==6) || month==9) || month==11)
```

If the value of the variable `month` is `4`, the above condition is evaluated to be:

```
(((true || month==6) || month==9) || month==11)
```

Because of the short-circuit evaluation nature of the `||` operator, the complex boolean expression (`true || month==6`) is `true` without evaluating the condition `month==6` that is the second operand of the `||` operator. Then, the condition becomes:

```
((true || month==9) || month==11)
```

Similarly, it is not necessary to evaluate the condition `month == 9`, because of the short-circuit evaluation nature. As a consequence, as soon as a sub-condition is evaluated to be `true`, the subsequent sub-conditions can be neglected, and the overall result is determined to be `true`.

You can use the && operator to construct complex boolean expressions that are composed of more than two sub-conditions. For example:

```
if (a > 0 && b > 0 && c > 0 && d > 0) {
    ......
}
```

Like the `||` operator, the && operator is left associative and the condition in the above statement is interpreted as:

```
((((a > 0 && b > 0) && c > 0) && d > 0)
```

With similar arguments for the `||` operator, as soon as a sub-condition is evaluated to be `false`, all subsequent sub-conditions are neglected and the overall result is determined to be `false`.

Most programmers use the && operator and `||` operator for constructing complex `boolean` expressions that involve more than one `boolean` sub-expression for logical AND and logical OR operations respectively without being concerned whether it is short-circuit or not. The && and `||` operators implicitly eliminate unnecessary `boolean` sub-expression evaluations so the execution performance can be improved.

However, whenever you want to construct a complex `boolean` expression with short-circuit operators, you should remind yourself whether the sub-conditions, other than the first one, have to be evaluated in all cases. If all sub-conditions have to be evaluated, especially those with side effects, you should not use the short-circuit operators. We will discuss another way to specify such complex `boolean` expressions.

The following is a practical use of a short-circuit operator. Do you remember the bank transfer operation we discussed in *Unit 2*? It involves two class definitions, `Bank` and `BankAccount`. Each `BankAccount` object has two behaviours, `withdrawal` and `deposit`. If feedback (the return value after the method execution) is required where the `boolean` values `true` and `false` denote successful and failed executions respectively, the class designs are modified as shown in Figure 6.2.

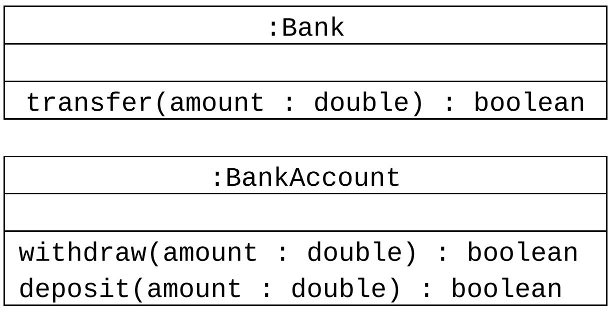

:Bank
transfer(amount : double) : boolean
:BankAccount
withdraw(amount : double) : boolean
deposit(amount : double) : boolean

Figure 6.2 The designs of classes `Bank` and `BankAccount`

The `transfer()` method can be modified to be:

```
public boolean transfer( BankAccount source,
                         BankAccount destination,
                         double amount) {
    // withdraw money from source account and
    // deposit money to destination account, then
    // the overall result is determined by the
    // and operator (double ampersand operator).
    return
        source.withdraw(amount) &&
        destination.deposit(amount);
}
```

The two method calls, `source.withdraw(amount)` and `destination.deposit(amount)`, return either `true` or `false`. If both method calls return `true`, it means that the money has been successfully withdrawn from the source account and deposited in the destination account. If one of the method calls returns `false`, the operation should be considered to be `false`, which is the reason why the && operator is chosen.

What would you think if the result of the first sub-operation `source.withdraw(amount)` is `false`? Is it necessary to, or even try to, deposit money to the destination account? The first sub-operation `false` means that it is not possible to withdraw money from the source account, most likely because of insufficient funds (the balance) in the source account. Then, it is not necessary to execute the second method call `destination.deposit(amount)` to deposit the money in the destination account.

Due to the short-circuit evaluation feature of the && operator, the above definition of the method `transfer()` is equivalent to the following one:

```
public boolean transfer( BankAccount source,
                         BankAccount destination,
                         double amount) {
    // withdraw money from source account
    boolean result;
    if (result = source.withdraw(amount)) {
        // deposit money to destination account
        result = destination.deposit(amount);
    }
    // Return the overall result
    return result;
}
```

If the execution of the `withdraw()` method of the `source` account object is `true`, it *is* necessary to call the `deposit()` method of the destination `BankAccount` object, and the overall result is determined by the `deposit()` method call. If the execution of the `withdraw()` method of the source account object is already `false`, it is the overall result and it *is not* necessary to call the `deposit()` method of the `destination` account object.

Comparing the two definitions of the method `transfer()`, we see that the former is simpler and implicitly reduces unnecessary condition evaluations, whereas the latter one is more explicit. Furthermore, you should notice that it is dangerous to define the behaviour of a method that depends on short-circuit evaluation, because it taxes inexperienced programmers and the program logic cannot be simply imported to other programming languages. Not all programming languages support short-circuit evaluations.

If a condition contains either all && operators or all `||` operators, the parentheses are usually unnecessary, and the sub-conditions are evaluated from the leftmost one to the right. However, you have to be careful if a complex condition contains both && operator(s) and `||` operator(s), such as:

```
if (a > 0 || b > 0 && c > 0) {
    ......
}
```

The && operator takes precedence over the `||` operator. Therefore, the above complex condition is interpreted to be

```
(a > 0 || (b > 0 && c > 0))
```

instead of

```
((a > 0 || b > 0) && c > 0)
```

If your real intention in the above in the values of either variable `a` or `b` is greater than zero, and the value of variable `c` must be greater than zero, the `if` statement should be modified to be:

```
if ((a > 0 || b > 0) && c > 0) {
    ......
}
```

You are hence highly recommended to use parentheses in complex conditions that contain both && operator(s) and `||` operator(s) to avoid ambiguities and make it easier for readers to understand the conditions *and* make it easier for programmers to maintain the program.

There are equivalent but non-short-circuit operators for logical AND and OR operators. They are the & (single ampersand) operator and `|` (single pipe) operator. For example, the following two program segments,

```
if (a > 0 && b > 0) {
    ......
}
```

and

```
if (a > 0 & b > 0) {
    ......
}
```

are equivalent, but the latter is not a short-circuit evaluation, which means that no matter what the result of the first operand of the & operator is, the second operand must be evaluated, and the overall result is determined according to the truth table shown in Table 6.10.

Table 6.10 The truth table of the & operator

Conditions		Result
a	*b*	*a* `&` *b*
`true`	`true`	`true`
`true`	`false`	`false`
`false`	`true`	`false`
`false`	`false`	`false`

The truth table of the & operator is exactly the same as for the && operator — both denote the logical AND operation.

Similarly, the non-short circuit equivalent operator for the `||` operator is the `|` operator — both denote the logical OR operator. The difference between them is that even if the result of the first operand of the `|` operator is evaluated to be `true`, the second operand must be evaluated, and the overall result is determined according to the truth table shown in Table 6.11. Its truth table is exactly the same as that for the `||` operator.

Table 6.11 The truth table of the | operator

Conditions		Result
a	*b*	*a* `|` *b*
`true`	`true`	`true`
`true`	`false`	`true`
`false`	`true`	`true`
`false`	`false`	`false`

In conclusion, short-circuit operators suffice for most uses, as they can minimize most unnecessary condition checking. However, non-short-circuit operators should be used if the two operands of the operator have to be evaluated, especially the `boolean` sub-expressions with side effects. The sub-expressions, both of which are method calls that return a `boolean` return value, used in the `if` statement of the `transfer()` method in the bank account transfer scenario are examples of `boolean` sub-expressions with side effects. Another example of `boolean` expressions with side effect is the use of pre/post increment/decrement operators (`++` and `--`), such as in the following conditions:

```
(++i > 0 && ++j > 0)
```

The interpretation of the above condition is that the value of the variable `i` is increased by one, and the value after increment is compared with value `0`. If the value after increment is greater than `0`, the value of the variable `j` is increased by one and then compared with value `0`. If the value of the variable `i` after increment is less than or equal to `0`, the `second operand` of the && operator, `++j > 0`, is not executed and the value of the variable `j` will not be increased by one.

A condition that is composed of short-circuit && and/or `||` operators with `boolean` sub-expressions that have side effects is difficult to understand and maintain, which is one of the reasons why it is recommended you use the `++` and `--` operators in simple expressions only.

The following table summarizes precedence and associativities of the logical and relational operators you have learned so far.

Table 6.12 Associativities and precedences of boolean and relational operators

Precedence	**Associativity**
`>= > < <=`	Left
`== !=`	Left
`&`	Left
`^`	Left
`|`	Left
`&&`	Left
`||`	Left

Self-test 6.2

Determine the values of the variables `a`, `b` and `c` after the following program segments are executed individually:

1
```
int a = 0;
int b = 0;
int c = 0;
if (a++ >= 0 || b++ >= 0) {
    c++;
}
```

2
```
int a = 0;
int b = 0;
int c = 0;
if (a++ >= 0 && b++ >= 0) {
    c++;
}
```

Bitwise operators

The `&`, `|` and `^` operators are not only `boolean` operators that involve two `boolean` values, they are also bitwise operators to be used with two integral values. As an integral value is represented as a sequence of binary digits (`0` and `1`)in the binary number system in a computer, a bitwise operator applies the logical `AND`, `OR` or `XOR` operations bit by bit with the two integral numbers where binary digits `1` and `0` are considered to be `boolean` values `true` and `false` respectively.

For example, the following eight-digit binary numbers represent integral values `73` and `98`.

73_{10}	=	0	1	0	0	1	0	0	1

98_{10}	=	0	1	1	0	0	0	1	0

The result of the expression `73 & 98` is interpreted to be

73_{10}	=	0	1	0	0	1	0	0	1	
98_{10}	=	0	1	1	0	0	0	1	0	
$73_{10}\&98_{10}$	=	0	1	0	0	0	0	0	0	= 64_{10}

where `1` and `0` denote `true` and `false` respectively.

Similarly, the expressions `73 | 98` and `73 ^ 98` are evaluated to be:

73_{10}	=	0	1	0	0	1	0	0	1	
98_{10}	=	0	1	1	0	0	0	1	0	
$73_{10}\|98_{10}$	=	0	1	1	0	1	0	1	1	= 107_{10}

73_{10}	=	0	1	0	0	1	0	0	1	
98_{10}	=	0	1	1	0	0	0	1	0	
73_{10}^98_{10}	=	0	0	1	0	1	0	1	1	= 43_{10}

Also, there is a bitwise negation or complement operator, the `~` operator. It is similar to the `!` operator for a `boolean` value that gives `true` for `false` and `false` for `true` respectively. With respect to bitwise operations, the `~` operator gives `1` for `0` and `0` for `1`. For example, the expression `~73` is evaluated to be:

73_{10}	=	0	1	0	0	1	0	0	1
~73_{10}	=	1	0	1	1	0	1	1	0

With bitwise operators, it is possible to manipulate a particular bit of an integral value and each bit of the integral value can be used as storage of a `boolean` value, either `true` or `false`. For example, a student can enroll in courses among a list of available courses, and a bit (of integral value) can be used to indicate whether the enrolment list contains a particular course, such as:

TM222	T225	T223	T222	T202	S271	MT269	MT268	MT260	MT258	MT210	MT201	MST207	MST204	MT261	MT246	M245	M221	M205
0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0

There are 19 courses, and the least significant 19 bits of a value of type `int` corresponds to a particular course. If the value of a bit is `1`, the corresponding course is chosen in the enrolment. Otherwise, the corresponding course is not chosen in the enrolment.

By making use of such an implementation, a class `Enrolment` is written as shown in Figure 6.3. Note that the values for the courses are expressed as hexadecimal values.

```
// Definition of class Enrolment
public class Enrolment {
   // Constant variables for different courses
   public static final int COURSE_M205   = 0x00000001;
   public static final int COURSE_M221   = 0x00000002;
   public static final int COURSE_M245   = 0x00000004;
   public static final int COURSE_M246   = 0x00000008;
   public static final int COURSE_M261   = 0x00000010;
   public static final int COURSE_MST204 = 0x00000020;
   public static final int COURSE_MST207 = 0x00000040;
   public static final int COURSE_MT201  = 0x00000080;
   public static final int COURSE_MT210  = 0x00000100;
   public static final int COURSE_MT258  = 0x00000200;
   public static final int COURSE_MT260  = 0x00000400;
   public static final int COURSE_MT268  = 0x00000800;
   public static final int COURSE_MT269  = 0x00001000;
   public static final int COURSE_S271   = 0x00002000;
   public static final int COURSE_T202   = 0x00004000;
   public static final int COURSE_T222   = 0x00008000;
   public static final int COURSE_T223   = 0x00010000;
   public static final int COURSE_T225   = 0x00020000;
   public static final int COURSE_TM222  = 0x00040000;

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

    // Immutable array object maintaining course names
    public static final String[] courseNames = {
        "Fundamentals of Computing",
        "Mathematical Methods",
        "Probability and Statistics",
        "Elements of Statistics",
        "Mathematics for Computing",
        "Mathematical Models and Methods",
        "Mathematical Methods, Models and Modelling",
            "Computing Fundamentals with Java",
            "Computing Fundamentals",
            "Programming and Database",
            "Computer Architecture and Operating Systems",
            "Commercial Information Processing",
            "Commercial Information Systems and Programming",
            "Discovering Physics",
            "Analogue and Digital Electronics",
            "Electronics Principles and Digital Design",
            "Microprocessor-based Computers",
            "Analogue Circuits",
            "The Digital Computer"
        };

        // The variable representing the courses a student is taking,
        // and the enrolment list initially contains no course.
        private int coursesTaking = 0;

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

        // Remove a course from the Enrolment
        public void removeCourse(int course) {
            coursesTaking &= ~course;
        }

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

        // Determine whether the course is chosen
        public boolean contains(int course) {
            return (coursesTaking & course) != 0;
        }

        // Show the chosen courses in this Enrolment
        public void showCourses() {
            int thisCourse = 1;
            int courseCount = 0;
            for (int i = 0; i < courseNames.length; i++) {
                if (contains(thisCourse)) {
                    courseCount++;
                    System.out.println(courseCount + ": " +
                        courseNames[i] + " [" +
                        courseCodes[i] + "]");
                }
                thisCourse *= 2;
            }
        }
    }
```

Figure 6.3 Enrolment.java

The definition of the class `Enrolment` defines a list of public immutable class variables, such as

```
public static final int COURSE_MT201 = 0x00000080;
```

and they are initialized with a value of `int` that is represented with a hexadecimal number that is prefixed with `0x`. Each hexadecimal value represents a 32-bit binary number in which a single bit is `1`.

The class definition further defines two immutable array objects that maintain `String` objects for course codes and course names.

Each `Enrolment` object has an object attribute `coursesTaking` of type `int`. The least significant 19 bits of the object attribute are used to represent the 19 courses. The attribute is initialized to be `0`, which implies no course is chosen initially.

The `addCourse()` method is called with a course code to add the corresponding course (denoted by the supplied course code) to the `Enrolment` object. In the method, the statement

```
coursesTaking |= course;
```

involves the `|=` operator, which look likes the `+=` operator. It is equivalent to the following statement:

```
coursesTaking = coursesTaking | course;
```

For example, if the values of the object attribute `coursesTaking` and the parameter `course` are

	TM222	T225	T223	T222	T202	S271	MT269	MT268	MT260	MT258	MT210	MT201	MST207	MST204	MT261	MT246	M245	M221	M205
`coursesTaking`	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0

and

	TM222	T225	T223	T222	T202	S271	MT269	MT268	MT260	MT258	MT210	MT201	MST207	MST204	MT261	MT246	M245	M221	M205
`course`	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

respectively, the value of the parameter `course` can be represented by the immutable variable `COURSE_MT201` (`0x00000080`). The result of the expression

```
coursesTaking | course
```

is given by:

	TM222	T225	T223	T222	T202	S271	MT269	MT268	MT260	MT258	MT210	MT201	MST207	MST204	MT261	MT246	M245	M221	M205
`coursesTaking`	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0
`course`	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
Result of |	0	0	0	0	0	0	1	0	1	0	0	1	0	0	0	0	0	0	0

Comparing the result and the original value of `coursesTaking`, the bit that corresponds to the course MT201 is set to be `1` by the bitwise `|` operator. An observation is that no matter whether the bit for the course MT201 is either `0` or `1`, the bit in the result must be `1`. Afterwards, the result of the above expression is assigned to the object attribute `coursesTaking`.

To remove a course from the `Enrolment` object, the statement in the method `removeCourse()` is:

```
coursesTaking &= ~course;
```

An equivalent statement of the above is:

```
coursesTaking = coursesTaking & ~course;
```

The `~` operator takes precedence over the bitwise `&` operator. Therefore, each bit in the value of parameter `course` is negated or complemented, so that `1` becomes `0` and `0` becomes `1`. Afterwards, the complemented value is involved in a bitwise `AND` operation with the current value of the object attribute `coursesTaking` and the result is updated to the object attribute.

For example, the values of the `coursesTaking` and `course` are

	TM222	T225	T223	T222	T202	S271	MT269	MT268	MT260	MT258	MT210	MT201	MST207	MST204	MT261	MT246	M245	M221	M205
`coursesTaking`	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0

and

	TM222	T225	T223	T222	T202	S271	MT269	MT268	MT260	MT258	MT210	MT201	MST207	MST204	MT261	MT246	M245	M221	M205
`course`	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0

respectively. According to the statement, the value of the parameter `course` is bitwise negated as visualized below:

	TM222	T225	T223	T222	T202	S271	MT269	MT268	MT260	MT258	MT210	MT201	MST207	MST204	MT261	MT246	M245	M221	M205
`course`	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
`~course`	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1

(There are 32 bits for a value of type `int` and the bits other than the least significant 19 bits are not shown above, as they are unused.)

Then, the complemented value is involved in a bitwise `AND` operation with the current value of the object attribute `coursesTaking`:

	TM222	T225	T223	T222	T202	S271	MT269	MT268	MT260	MT258	MT210	MT201	MST207	MST204	MT261	MT246	M245	M221	M205
`coursesTaking`	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0
`~course`	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1
Result of `&`	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

Comparing the result with the original value of the object attribute `coursesTaking`, the bit for the course MT260 is set to be `0`.

The method `toggleCourse()` can be used to remove a course if the course is chosen or to choose a course if the course is currently selected. At first glance, it is necessary to use an `if/else` statement to set or unset a bit for the corresponding course. However, with the use of a bitwise `^` operator, the method body is as simple as the statement:

```
coursesTaking ^= course;
```

Similarly, the statement with an `^=` operator is equivalent to the following one:

```
coursesTaking = coursesTaking ^ course;
```

For example, the values of the object attribute `coursesTaking` and parameter `course` are

	TM222	T225	T223	T222	T202	S271	MT269	MT268	MT260	MT258	MT210	MT201	MST207	MST204	MT261	MT246	M245	M221	M205
`coursesTaking`	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0

and

	TM222	T225	T223	T222	T202	S271	MT269	MT268	MT260	MT258	MT210	MT201	MST207	MST204	MT261	MT246	M245	M221	M205
`course`	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

The result of the expression

```
coursesTaking ^ course
```

is obtained as:

	TM222	T225	T223	T222	T202	S271	MT269	MT268	MT260	MT258	MT210	MT201	MST207	MST204	MT261	MT246	M245	M221	M205
`coursesTaking`	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0
`course`	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
Result of `^`	0	0	0	0	0	0	1	0	1	0	0	1	0	0	0	0	0	0	0

The bit for the course MT201 is set in the result according to the bitwise `XOR` operation, and the result is used to update the object attribute `coursesTaking`. If such a value of object attribute `coursesTaking` is involved in a bitwise `XOR` operation with the same value as parameter `course`, the result is obtained as:

	TM222	T225	T223	T222	T202	S271	MT269	MT268	MT260	MT258	MT210	MT201	MST207	MST204	MT261	MT246	M245	M221	M205
`coursesTaking`	0	0	0	0	0	0	1	0	1	0	0	1	0	0	0	0	0	0	0
`course`	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
Result of ^	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0

The bit for the course MT201 is now set to be `0`. Therefore, the bitwise `^` operator can be used to toggle a bit in the integer value instead of using an `if/else` statement such as:

```
public void toggleCourse(int course) {
  if (contains(course)) {
    removeCourse(course);
  } else {
    addCourse(course);
  }
}
```

The `contains()` method can be used to determine whether the supplied course is chosen by the `Enrolment` object. The method contains a single `return` statement that returns the result of the following expression,

```
   (coursesTaking & course) != 0
```

The bitwise & operator performs a bitwise `AND` operation bit by bit for the values of object attribute `coursesTaking` and parameter `course`. According to truth table of the `AND` operation, a bitwise `AND` operation gives a value of `1` only if the bits with the same position of the two integer values are both `1`. As the parameter `course` determines the course to be tested (where the corresponding bit of the course is `1`), if the same bit of the object attribute `coursesTaking` is `1`, the result is `1`, or `0` otherwise. If the result of the bitwise `AND` operation is non-zero, that is, there is at least one bit in the result that is non-zero, it indicates the bit for the corresponding course specified by the object attribute `coursesTaking` is `1`.

For example, the current value of the object attribute `coursesTaking` is:

	TM222	T225	T223	T222	T202	S271	MT269	MT268	MT260	MT258	MT210	MT201	MST207	MST204	MT261	MT246	M245	M221	M205
`coursesTaking`	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0

To verify whether the `Enrolment` contains the course MT201, the method `contains()` is called with a value `Enrolment.COURSE_ MT201` via parameter `course.` The result of the expression

```
   coursesTaking & course
```

is given by:

	TM222	T225	T223	T222	T202	S271	MT269	MT268	MT260	MT258	MT210	MT201	MST207	MST204	MT261	MT246	M245	M221	M205
`coursesTaking`	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0
`course`	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
Result of `&`	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

The result of the bitwise `&` operation is non-zero, and the expression

```
(coursesTaking & course) != 0
```

is evaluated to be `true` and is returned as the `return` value of the method call. However, if the same `Enrolment` object is checked to see if it contains the course MT210, the `contains()` method is called with the value `Enrolment.COURSE_MT210` (`0x00000100`) and the expression of

```
coursesTaking & course
```

is evaluated to be:

	TM222	T225	T223	T222	T202	S271	MT269	MT268	MT260	MT258	MT210	MT201	MST207	MST204	MT261	MT246	M245	M221	M205
`coursesTaking`	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0
`course`	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
Result of `&`	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

All bits of the resultant value are zero and the result of the expression

```
(coursesTaking & course) != 0
```

is therefore `false` and is returned as the method call `return` value.

Finally, the `Enrolment` class defines the `showCourses()` method for listing the courses chosen on the screen. In the method, a `for` loop is used to verify whether courses are chosen one by one.

```
int thisCourse = 1;
int courseCount = 0;
for (int i = 0; i < courseNames.length; i++) {
    if (contains(thisCourse)) {
      courseCount++;
      System.out.println(courseCount + ": " +
            courseNames[i] + " [" +
            courseCodes[i] + "]");
    }
    thisCourse *= 2;
}
```

The local variable `thisCourse` stores the current course code to be verified. For the first iteration of the `for` loop, its value is `1`, that is:

	TM222	T225	T223	T222	T202	S271	MT269	MT268	MT260	MT258	MT210	MT201	MST207	MST204	MT261	MT246	M245	M221	M205
thisCourses	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Then, the value of the local variable `thisCourse` is supplied to the `contains()` method to verify whether the course M205 is chosen. If the return value of the `contains()` method is `true`, the `if` part of the `if` statement is executed showing the course on the screen.

Afterwards, the statement

```
thisCourse *= 2;
```

is executed. The statement is equivalent to the following:

```
thisCourse = thisCourse * 2;
```

The value of the variable `thisCourse` becomes `2` and the corresponding binary number is 10_2, that is:

	TM222	T225	T223	T222	T202	S271	MT269	MT268	MT260	MT258	MT210	MT201	MST207	MST204	MT261	MT246	M245	M221	M205
thisCourses	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

Every time the value of the variable `thisCourse` doubles, the current bit is set to `0` and the next bit is set to `1`. With such a behaviour, the variable `thisCourse` takes values of all possible course codes to be verified. The chosen courses, which are determined by calling the `contains()` method with the value of `thisCourse`, are shown on the screen.

The methods defined in the `Enrolment` class are short but might be strange to you at first glance. To help you understand the class definition further, a class `TestEnrolment` is written in Figure 6.4 for testing the `Enrolment` class.

```
// Definition of class TestEnrolment
public class TestEnrolment {

    // Main executive method
    public static void main(String args[]) {
        // Create the Enrolment object
        System.out.println("Create an Enrolment object");
        Enrolment enrolment = new Enrolment();

        // Add the courses
        System.out.println("Add the courses");
        enrolment.addCourse(Enrolment.COURSE_MT201);
        enrolment.addCourse(Enrolment.COURSE_MT260);
        enrolment.addCourse(Enrolment.COURSE_MT268);
        enrolment.addCourse(Enrolment.COURSE_MT269);
        // Show the chosen courses
        System.out.println("Show the existing Enrolment object");
        enrolment.showCourses();

        // Modify the Enrolment object
        System.out.println("Modify the Enrolment object");
        enrolment.removeCourse(Enrolment.COURSE_MT268);
        enrolment.toggleCourse(Enrolment.COURSE_MT210);
        enrolment.toggleCourse(Enrolment.COURSE_MT269);

        // Show the chosen course again
        System.out.println(
            "Show the Enrolment object after modification");
        enrolment.showCourses();

        // Determine whether the course MT201 is chosen
        if (enrolment.contains(Enrolment.COURSE_MT201)) {
            System.out.println(
                "The course MT201 has been chosen.");
        }
        else {
            System.out.println(
                "The course MT201 has not been chosen.");
        }
    }
}
```

Figure 6.4 TestEnrolment.java

Compile the classes and execute the `TestEnrolment` program. The following output is shown on the screen:

```
Create an Enrolment object
Add the courses
Show the existing Enrolment object
1: Computing Fundamentals with Java [MT201]
2: Computer Architecture and Operating Systems [MT260]
3: Commercial Information Processing [MT268]
4: Commercial Information Systems and Programming [MT269]
Modify the Enrolment object
Show the Enrolment object after modification
1: Computing Fundamentals with Java [MT201]
2: Computing Fundamentals [MT210]
3: Computer Architecture and Operating Systems [MT260]
The course MT201 has been chosen.
```

Please use the following self-test to verify your understanding of bitwise operators.

Self-test 6.3

1 With the `Enrolment` class definition discussed above, is it possible to add, remove, and toggle more than one course by a single method call? For example:

```
Enrolment enrolment = new Enrolment();
enrolment.addCourse(Enrolment.COURSE_MT201 |
                    Enrolment.COURSE_MT210 |
                    Enrolment.COURSE_MT268);
```

2 Is it possible to verify whether an `Enrolment` object contains more than one course at a time? For example,

```
Enrolment enrolment = new Enrolment();
......
enrolment.contains(Enrolment.COURSE_MT201 |
                   Enrolment.COURSE_MT210 |
                   Enrolment.COURSE_MT268);
```

What is the interpretation of the method call to the `contains()` method above? It does not work as expected. Suggest a way to modify the `contains()` method if the method does not perform as expected.

3 Modify the definition of the `Enrolment` class so that it uses a 19-element array object with element type `boolean` to maintain the courses chosen. Compare the two implementations of the `Enrolment` class.

Remember, complex logical expressions can be quite tricky, especially those with the results depending on short-circuit evaluations. If used incorrectly, they can be a source of frustration even for experienced programmers. Therefore, think things through very carefully before you start using such expressions in your programs; make sure you really know what you want to do before you start. Put another way, make sure you build in the correct ‘logic’ before you build your complex logical expressions — make sure *your* logic is logical.

Advanced branching statements

Branching statements are crucial in programming, as they represent and implement the logic behind a behaviour. For example, the following is a common `if/else` statement that determines a value to be assigned to a variable

```
if (isMember) {
    discount = 0.8;
}
else {
    discount = 0.9;
}
```

where the variable `isMember` is of `boolean` type with a value of either `true` or `false`. If `isMember` is `true`, a value of `0.8` is assigned to the variable `discount`. Otherwise, the value `0.9` is assigned to the variable `discount`. In the Java programming language, there is a ternary conditional operator (`?:`) that can be treated as a shorthand form of the above `if/else` statement. The format of the operator is:

condition ? value-for-true : value-for-false

It is the only ternary operator (an operator that involves three expressions) in the Java programming language. The *condition* is evaluated first. If the condition result is evaluated to be `true`, the overall result of the above expression is determined to be the *value-for-true*. Otherwise, the *value-for-false* is the overall expression result. With such an operator, the above `if/else` statement can be simplified as:

```
discount = isMember ? 0.8 : 0.9;
```

The operator is very simple and very handy. If the condition, the *value-for-true* or the *value-for-false* involves complex expressions, it is preferable to use parentheses to avoid ambiguity, such as:

```
price = (age < 12) ? 2.5 : 5.0;
```

You can use this ternary conditional operator to simplify branching statements that are dedicated to determining a value for a variable.

Besides the above operator, you learned two branching statement structures in *Unit 4* —`if/else` and `switch/case` statements — which are supported by the Java programming language. Furthermore, in the previous section, you learned how to construct complex conditions. Combining these two techniques, you can implement even more complex algorithms. For example, if an object possesses attributes, `boyCount`, `girlCount`, `manCount` and `ladyCount`, which are the counts for boys, girls, men and women, you can write the following `increase()` method to increase the corresponding attribute based on the parameters:

```
    public void increase(int age, char gender) {
        if (age < 18 && gender == 'M') {
                boyCount++;
            }
            if (age < 18 && gender == 'F') {
                girlCount++;
            }
            if (age >= 18 && gender == 'M') {
                manCount++;
            }
            if (age >= 18 && gender == 'F') {
                ladyCount++;
            }
        }
```

There are four combinations and hence four complex conditions in the definition of the `increase()` method. The four complex conditions are evaluated in sequence. If the first complex condition is evaluated to be `true`, you can be sure that all subsequent complex conditions must be `false`. However, according to the above method definition, the remaining three conditions will also be evaluated, which involves unnecessary condition evaluations and hence worse performance. Furthermore, the four complex conditions written in this way are error-prone, because you have to make sure that the four complex conditions correspond correctly to the four different categories.

In the following sections, we discuss some advanced uses of the branching statements so that complex conditions can be improved and simplified. First of all, please use the following reading to review the general use of `if/else` statements. Some advanced uses of `if/else` statements such as nested `if/else` statements are also mentioned. Afterwards, some concepts introduced in the reading are clarified and elaborated on.

Reading

King, section 4.4 ‘If statements with else clauses’, pp. 140–47

Nested `if` / `else` statements

The usual format of an `if/else` statement is:

```
if (condition) {
    // if part statements
    ......
}
else {
    // else part statements
    ......
}
```

The statements in the `if` part and `else` part that you came across earlier were just a sequence of statements. As mentioned in the reading you have just completed, the statements in the two parts of the `if/else` statement can also contain other `if/else` statements. For example:

```
if (condition1) {
    // outer if part statements
    if (condition2) {
        // inner if part statements for
        // condition1==true && condition2==true
        ......
    }
    else {
        // inner else part statements for
        // condition1==true && condition2==false
        ......
    }
}
else {
    // outer else part statements
    if (condition3) {
        // inner if part statements for
        // condition1==false && condition3==true
        ......
    }
    else {
        // inner else part statements for
        // condition1==false && condition3==false
        ......
    }
}
```

For the above program structure, an `if/else` statement enclosed in another `if/else` statement is known as a nested `if/else` statement. The above nested `if/else` statement involves three conditions. First of all, *condition1* is evaluated. If the result is `true`, the `if` part of the outer `if/else` statement is executed. Then, *condition2* is evaluated to determine whether the `if` part or the `else` part of the inner `if/else` statement is executed.

If the outer `if/else` statement is evaluated to be `false`, however, the `if` part of the outer `if/else` statement is skipped and the `else` part is executed instead. Then, the result of *condition3* is evaluated to determine whether the `if` part or the `else` part of the inner `if/else` statement, which is enclosed in the outer `else` part statement, is executed.

The description of the nested `if/else` statement can be visualized through Figure 6.5.

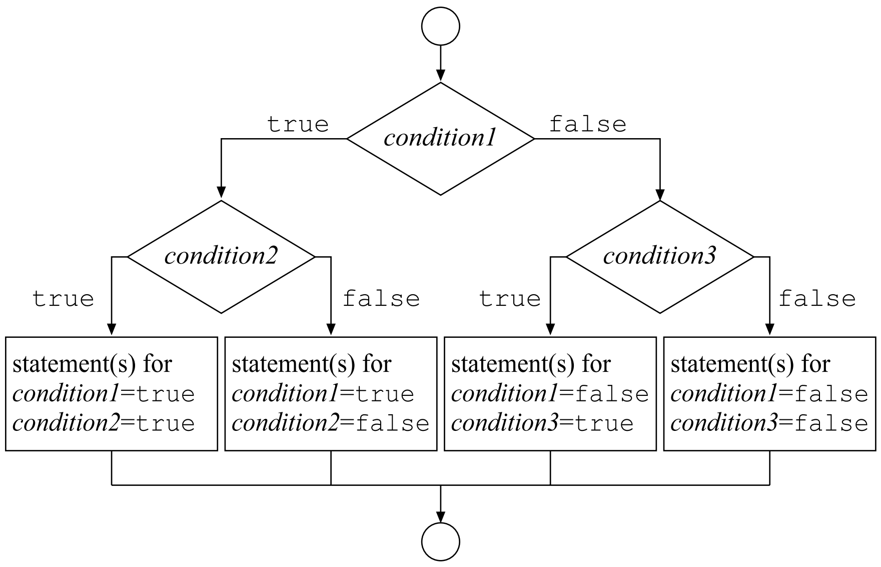

true
condition1
false
condition2
true
false
condition3
true
false
statement(s) for *condition1*=true *condition2*=true
statement(s) for *condition1*=true *condition2*=false
statement(s) for *condition1*=false *condition3*=true
statement(s) for *condition1*=false *condition3*=false

Figure 6.5 A flowchart illustrating a nested `if/else` construct

For example, the definition of the `increase()` method presented in the previous section can be enhanced to be:

```
public void increase(int age, char gender) {
    if (age < 18) {
        // either boy or girl
        if (gender == 'M') {
            // statement for boy
            boyCount++;
        }
        else {
            // statement for girl
            girlCount++;
        }
    }
    else {
        // either man or woman
        if (gender == 'M') {
            // statement for man
            manCount++;
        }
        else {
            // statement for woman
            womanCount++;
        }
    }
}
```

You can see that even though the structure seems to be more complex, it is easier for readers to understand the logic behind the operations. Furthermore, there are only three *simple* conditions in the method and no redundant condition. As a result, the performance is better than the previous version. If the age for classifying children and adults is changed to `21`, only one simple condition needs modification. Compared with the previous version, all four conditions have to be modified, and it can be error-prone if the software developer forgets to update any one of the four.

The above structure is only one version of nested `if/else` statements. The `if` part and `else` part of an `if/else` statement can contain any number of inner `if/else` statements. Furthermore, the `else` part of an `if/else` statement is optional. You can write your own nested `if/else` statements according to the required logic.

We mentioned that if there is only one statement in the `if` part or `else` part, the pair of curly braces is optional. Therefore, most curly braces in the `increase()` method are optional, and the method can be simplified to be:

```
public void increase(int age, char gender) {
    if (age < 18)
        // either boy or girl
        if (gender == 'M')
            // statement for boy
            boyCount++;
        else
            // statement for girl
            girlCount++;
    else
        // either man or woman
        if (gender == 'M')
            // statement for man
            manCount++;
        else
            // statement for woman
            womanCount++;
}
```

Although the method definition is much shorter and the indentation is easier for readers to understand the nested `if/else` statement, it is more difficult to maintain and debug, especially when the statements are not properly indented. For this reason, it is highly recommended you use the pair of curly braces even if there is only one statement in the block.

A problem of determining the grade according to the mark was also discussed in the reading. The ranges for the grades are listed in the following table.

Table 6.13 The grades with their corresponding mark ranges

Mark	Grade
90 to 100	A
80 to 89	B
70 to 79	C
60 to 69	D
0 to 59	F

Several program segments are presented in the reading to display the grade on the screen. In this section, we discuss the program segments that determine the grade, from the simplest one to those that are more advanced or compact.

The simplest program segment for determining the grade is:

```
String grade = "F";
if (mark >= 60) {
    grade = "D";
}
if (mark >= 70) {
    grade = "C";
}
if (mark >= 80) {
    grade = "B";
}
if (mark >= 90) {
    grade = "A";
}
```

If you review the above program segment, you will find that it involves redundant condition checking, as all four `if` statements are independent, but there are actually relationships among the conditions. For example, if the value of the variable `mark` is `75`, the results of the first two conditions are `true` and the `if` parts of these `if` statements are executed (the variable `grade` has been updated twice). The third condition is evaluated to be `false` and it is actually not necessary to evaluate the fourth condition `mark >= 90`. However, according to the above program segment, the fourth condition must be evaluated even though the result must be `false`.

As a result, we can convert the above `if` statements into a nested `if/else` statement so that the redundant conditions will not be verified. For example:

```
String grade = "";
if (mark >= 90) {
    grade = "A";
}
else {
    if (mark >= 80) {
        grade = "B";
    }
    else {
        if (mark >= 70) {
            grade = "C";
        }
        else {
            if (mark >= 60) {
                grade = "D";
            }
            else {
                grade = "F";
            }
            }
        }
    }
```

There are also four conditions in the above program segment. It is called a cascaded `if/else` statement in the reading. Now, let’s trace the execution sequence if the value of the variable `mark` is `75`. The first condition `mark >= 90` is evaluated and the result is `false`. Therefore, the `if` part of the first level `if/else` statement is skipped.

```
    String grade = "";
    if (mark >= 90) {
        grade = "A";
    }
    else {
        if (mark >= 80) {
            grade = "B";
        }
        else {
            if (mark >= 70) {
                grade = "C";
            }
            else {
                if (mark >= 60) {
                    grade = "D";
                }
                else {
                    grade = "F";
                }
            }
        }
    }
```

(In the above program segment, the shaded statements are skipped.)

The flow of control then proceeds to evaluate the second condition `mark >= 80;` the result is also `false`. The `if` part of the second level `if/else` statement is skipped.

```
    String grade = "";
    if (mark >= 90) {
        grade = "A";
    }
    else {
        if (mark >= 80) {
            grade = "B";
        }
        else {
            if (mark >= 70) {
                grade = "C";
            }
            else {
                if (mark >= 60) {
                    grade = "D";
                }
                else {
                    grade = "F";
                }
            }
        }
    }
```

Afterwards, the third condition is evaluated and the result is true. Therefore, the `if` part of the third level `if/else` is executed and the `else` part is skipped.

```
    String grade = "";
    if (mark >= 90) {
        grade = "A";
    }
    else {
        if (mark >= 80) {
            grade = "B";
        }
        else {
            if (mark >= 70) {
                grade = "C";
            }
            else {
                if (mark >= 60) {
                    grade = "D";
                }
                else {
                    grade = "F";
                }
            }
        }
    }
```

Then, the statement

```
   grade = "C";
```

is executed, which assigns the reference to a `String` object with content `"C"` to the variable `grade`. The subsequent conditions in the `else` part are skipped. You can see that as soon as a condition is verified to be `true`, the subsequent conditions will not be verified.

The above nested `if/else` statement is better than the one presented in *Unit 4* at the expense that it becomes a multiple level nested `if/else` statement that seems to be more complicated. We said that if there is only a single statement in either the `if` part or `else` part, the pair of curly braces is optional. In the above program segment, there is just an `if/else` statement in the `else` part of each level. Hence, we can remove the pairs of curly braces and rearrange the statements, so that it becomes

```
    String grade = "";
    if (mark >= 90) {
        grade = "A";
    }
    else
```

```
        if (mark >= 80) {
            grade = "B";
        }
        else
            if (mark >= 70) {
                grade = "C";
            }
            else
                if (mark >= 60) {
                    grade = "D";
                }
                else {
                    grade = "F";
                }
```

or

```
    String grade = "";
    if (mark >= 90) {
        grade = "A";
    }
    else if (mark >= 80) {
        grade = "B";
    }
    else if (mark >= 70) {
        grade = "C";
    }
    else if (mark >= 60) {
        grade = "D";
    }
    else {
        grade = "F";
    }
```

The former format reveals clearly that the program segment is a nested `if/else` statement, whereas the latter format prevents your program from indentation creep as the indentation increases for each new condition.

As all `if` and `else` parts of the above nested `if/else` statements contain one statement, all pairs of curly braces are therefore optional. The above program segment is therefore possible to be modified to be:

```
    String grade = "";
    if (mark >= 90)
        grade = "A";
    else if (mark >= 80)
        grade = "B";
    else if (mark >= 70)
        grade = "C";
    else if (mark >= 60)
        grade = "D";
    else
        grade = "F";
```

Although the last program segment is simpler, it is not recommended because it is more difficult to maintain and can be more error-prone.

Self-test 6.4

The state of water is determined according to its temperature. If the temperature is less than 0°C, the state of water is ice. If the temperature is higher than 100°C, it becomes steam.

Write a method `findState()` that accepts a temperature via a parameter of type `int` that returns `String` with content of `"ice"`, `"water"` or `"steam"` based on the value of the parameter.

Dangling-else problem

Suppose that a game accepts only ladies, and children who are under 18. The following program segment is written to determine whether a case is accepted or not:

```
boolean isAccepted = false
if (age >= 18)
    if (gender == 'F')
        isAccepted = true;
else
    isAccepted = true;
```

However, it is found that the value of the variable `isAccepted` is always `false` for a person under 18, which is not the intention of the program segment. What is the problem with the above program segment?

It is a typical dangling-else problem. The core problem of the above problem segment is with which `if` statement the `else` part associates. There are two possibilities:

```
boolean isAccepted = false
if (age >= 18) {
    if (gender == 'F')
        isAccepted = true;
}
else
    isAccepted = true;
```

and

```
boolean isAccepted = false
if (age >= 18) {
    if (gender == 'F')
        isAccepted = true;
    else
        isAccepted = true;
}
```

Please notice that the `else` part always associates with the nearest `if` statement, which means that the ambiguous program segment is interpreted as the latter version. As a result, if the first condition `age >= 18` is evaluated to be `false`, the nested `if/else` statement is skipped. When the readers read the original program segment, they are misled by the indentation.

An `if/else` statement without curly braces can give rise to the dangling else problem, which is the reason it is a good programming practice to use curly braces with branching and loop statements even if they contain only a single statement.

The use of `break` in `switch` / `case` structures

In *Unit 4*, we discussed the `switch/case` structure and its usage. The usual format of `switch/case` structure is:

```
switch (expression) {
case value1:
    statements-for-value1;
    break;
case value2:
    statements-for-value2;
    break;
......
......
default:
    statements-for-default;
    break;
}
```

Please use the following reading to review the use of `switch/case` structures and the role of the `break` statement.

Reading

King, section 8.2, 'The switch statement', pp. 306–14

For each case in the `switch/case` structure, the `break` statement is the last statement so that after all the statements that are dedicated for a case are executed, the `switch/case` structure terminates and the flow of control skips to the end of the structure. If the statements for a particular case are not terminated by a `break` statement, after all the statements in that case are executed, the flow of control will proceed to the next case so that the statements in the next case are executed as well. The statements for the subsequent cases are executed until a `break` statement is executed or end of the `switch/case` structure is reached.

With the aforementioned nature of the `break` statement in a `switch/case` structure, we discussed a situation in *Unit 4* that is used for determining the number of days in a month.

```
// Determine the number of days based on the given
month
switch (month) {
case 2:
   days = 28;
   break;
case 4:
case 6:
case 9:
case 11:
   days = 30;
   break;
default:
   days = 31;
   break;
}
```

For simplicity, the problem of the number of days in February in a leap year is not handled in the discussion.

The above `switch/case` structure has five cases and one `default` case. Values `4`, `6` and `9` have no statement; just the case for value `11` has. If the value of the variable `month` is `4`, `6` or `9`, the execution starts immediately. But there is no statement and no `break` statement, so the execution keeps on until the statements for the case for value `11`. The above `switch/case` structure is equivalent to the following nested `if/else` statement:

```
if (month == 2) {
    days = 28;
}
else if (month == 4 || month == 6 || month == 9 ||
month == 11) {
    days = 30;
}
else {
    days = 31;
}
```

To place or not to place the `break` statement into the cases in a `switch/case` structure gives you greater flexibility. For example, the following `switch/case` statement can determine the subjects a student takes according to his or her academic form.

```
boolean takeComputer = false;
boolean takePhysics = false;
boolean takeChemistry = false;
boolean takeBiology = false;
boolean takeChinese = false;
boolean takeEnglish = false;
boolean takeMaths = false;
boolean takeMusic = false;

switch (form) {
case 5: case 4:
    takeComputer = true;
case 3:
    takePhysics = true;
    takeChemistry = true;
    takeBiology = true;
default:
    takeChinese = true;
    takeEnglish = true;
    takeMaths = true;
    takeMusic = true;
    break;
}
```

In the above program segment, eight `boolean` variables are declared and initialized as `false`. Then, the `switch/case` structure is used to update those variables according to the value of the variable `form`. If the value of `form` is either `4` or `5`, all eight variables are updated to be `true`. Seven variables will be updated to be `true` if the value of `form` is `3`. Otherwise, such as for variable `form` with a value of `1` or `2`, only the last four variables will be updated to be `true`.

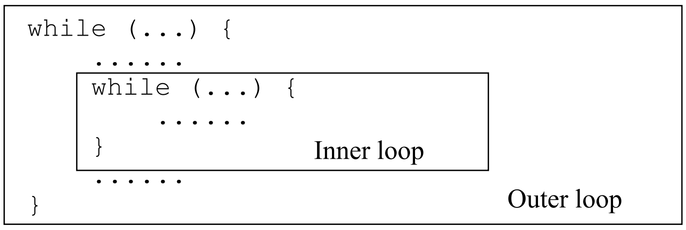
Of course, it is possible to write an equivalent program segment that uses `if/else` statements. However, the above `switch/case` statement is relatively more elegant and easier to read.

Self-test 6.5

Use a `switch/case` structure to update the variable `grade` according to the value of the variable `mark`. [Hint: what is the result of the expression `(mark/10)` for various values of variable `mark`?]

Advanced looping statements

Any number of statements can be placed in the loop body of a loop. These statements not only can include branching statements but looping statements as well, such as:



```
while (...) {
    ......
    while (...) {
        ......
    }
    ......
}
```

The above looping structure is known as a nested loop of two levels. The loop that encloses another loop is commonly called the outer loop; the loop being enclosed is called the inner loop. The outer loop and the inner loop are not necessarily of the same loop structures as in the above example. They can be any of the three looping statements that the Java programming language provides, such as:

```
while (...) {
    ......
    for (...; ...; ...) {
        ......
    }
    ......
}
```

The above is a two-level nested loop with a `while` outer loop and a `for` inner loop. It is strongly recommended you indent the program statements properly so that the readers can easily distinguish the inner loop in the outer loop body.

For example, you can use the following simple `for` loop to print a line of four stars (`*`).

```
for (int j=0; j < 4; j++) {
    System.out.print("*");
}
System.out.println();
```

The `for` loop in the above program segment displays a `*` (star) character on the screen four times, and hence a line of four stars is shown. The last statement displays nothing on the screen, but it ensures that the output to be displayed next starts at the leftmost position of the next line. If you execute a program with the above program segment, you will get the following display on the screen:

```
****
```

You can repeat the above program segment four times by placing the above program segment in the loop body of another loop, such as:

```
        for (int i=0; i < 4; i++) {
            for (int j=0; j < 4; j++) {
                System.out.print("*");
            }
            System.out.println();
        }
```

For every iteration of the outer loop, the inner loop is started, iterated and terminated. Therefore, for *each* iteration of the outer loop, the inner loop that displays a line of four stars is *completely* executed, and the outer loop iterates four times. As a result, if you execute the above program segment, four lines of four stars are shown on the screen as follows:

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

Of course, there are simpler ways to display the same output on the screen, and the above program is used for demonstrating the use of nested looping structures. We discuss some practical examples in this section where nested looping statements are a necessity.

The number of levels of a nested loop structures is unrestricted, but a nested looping structure with three levels usually suffices for most situations — and three levels is complicated enough. For example:

```
do {
    ......
    while (...) {
        ......
        for (...; ...; ...) {
            ......
        }
        ......
    ......
} while (...);
```

We mentioned that in a nested looping structure, the inner loop starts, iterates and terminates for each iteration of the outer loop. As a result, the number of times the innermost loop body is executed is usually determined by the product of the numbers of iterations of all levels of the nested looping structure. For example, the following has a similar nested `for` loop as in the previous one for printing stars.

```
for (int i=0; i < 4; i++) {
    for (int j=0; j < 4; j++) {
        ......
    }
}
```

A complete execution of the inner loop iterates its loop body four times. Since the outer loop repeats the complete execution of the inner loop four times, the total number of times the inner loop body is executed is 4 times 4, which equals 16.

If a nested loop is used just to repeat a loop body more, it is not very useful. For example, the previous nested `for` loop structure can be rewritten as the following simple `for` loop:

```
for (int i=0; i < 16; i++) {
    ......
}
```

By investigating the nested `for` loop, you can figure out its superficial features. First of all, in a nested `for` loop, two control variables (variable `i` and `j` in our example) are declared and are changed during the iteration of the inner loop and the outer loop. The outer `for` loop declares the control variable `i`, which is therefore accessible in the entire outer `for` loop body. The inner `for` loop, however, declares another control variable `j` so that it is accessible in the inner `for` loop. As the inner `for` loop is part of the loop body of the outer `for` loop, the inner `for` loop body can access the control variable `i` as well — as visualized in Figure 6.6.

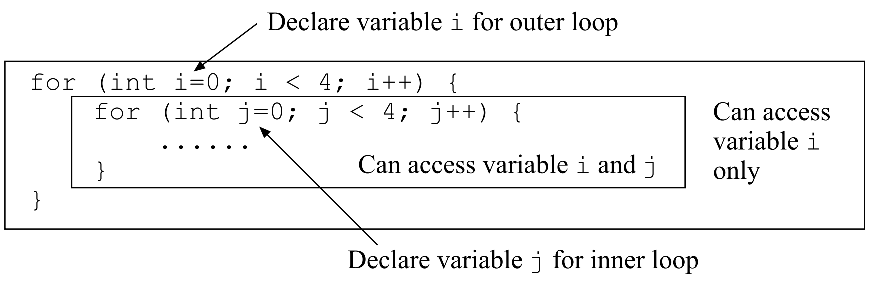

Declare variable i for outer loop
for (int i=0; i < 4; i++) {
for (int j=0; j < 4; j++) {
......
}
}
Can access variable i and j
Can access variable i only
Declare variable j for inner loop

Figure 6.6 Accessing the control variables in a nested looping structure

You should notice that if the variable `j` is defined inside the outer loop and before the inner `for` loop, or even before the outer `for` loop, the variable `j` can be accessed within the outer loop. However, such an approach is not preferable, especially if the value of the variable `j` is changed in both the outer loop and inner loop, because it complicates the program logic and the program is more difficult to read and maintain.

For the first iteration of the outer `for` loop, the value of the control variable `i` is kept as `0`. Then, the inner loop starts and the control variable `j` starts with `0` and increases to values `1`, `2` and `3` in the subsequent iterations. In the inner loop body, the combinations of values of variable `i` and `j` are therefore:

i	j
0	0
0	1
0	2
0	3

After the entire inner `for` loop is completed for the first time, the first iteration of the outer loop terminates. The value of the control variable `i` is increased to `1` and the second iteration of the outer `for` loop starts.

Then, the inner loop starts all over again. Therefore, the variable `j` is changed in each iteration, and it takes the values `0` to `3`. As a result, the combinations of the values for variables `i` and `j` are:

i	j
1	0
1	1
1	2
1	3

Similarly, the combinations of the values for the control variables in the third iteration of the outer `for` loop are:

i	j
2	0
2	1
2	2
2	3

For the fourth and the last iteration of the outer `for` loop, the combinations are:

i	j
3	0
3	1
3	2
3	3

To verify the changes in the variables while executing the nested loop, a class `ObjectWithNestedLoop` is written as shown in Figure 6.7.

```
// Definition of class ObjectWithNestedLoop
public class ObjectWithNestedLoop {

    // Show the control variables while executing a nested loop
    public void showVariablesInNestedLoop(int outerLimit,
                                          int innerLimit) {
        // Declare a variable to count the total number of
        // executions of the innermost loop body
        int count = 0;

        // The outer loop that iterates for outerLimit times
        for (int i=0; i < outerLimit; i++) {
            // The inner loop that iterates for innerLimit times
            for (int j=0; j < innerLimit; j++) {
                count++;
                // Display the values of variable i and j
                System.out.println(count + " : i=" + i + " & j=" + j);
            }
        }
        System.out.println(
            "Total number of iterations = " + count);
    }
}
```

Figure 6.7 ObjectWithNestedLoop.java

To test the `ObjectWithNestedLoop` class, a driver program, `TestObjectWithNestedLoop,` is written in Figure 6.8.

```
// Definition of class TestObjectWithNestedLoop
public class TestObjectWithNestedLoop {

    // Main executive method
    public static void main(String args[]) {
        // Verify whether sufficient program parameters are given
        if (args.length < 2) {
            // If insufficient program parameters are given, show
            // usage message
            System.out.println(
                "Usage: java TestObjectWithNestedLoop " +
                "<outer limit> <inner limit>");
        }
        else {
            // Create an ObjectWithNestedLoop object
            ObjectWithNestedLoop objectWithNL =
                new ObjectWithNestedLoop();
            // Obtain the outer and inner limits
            int outerLimit = Integer.parseInt(args[0]);
            int innerLimit = Integer.parseInt(args[1]);
            // Show the values of the control variables in a
            // nested for loop
            objectWithNL.showVariablesInNestedLoop(
                outerLimit, innerLimit);
        }
    }
}
```

Figure 6.8 TestObjectWithNestedLoop.java

Compile the classes and execute the `TestObjectWithNestedLoop` program with two program parameters for the outer limit and inner limit respectively, such as:

```
java TestObjectWithNestedLoop 4 3
```

The following output is shown on the screen:

```
1 : i=0 & j=0
2 : i=0 & j=1
3 : i=0 & j=2
4 : i=1 & j=0
```

```
5 : i=1 & j=1
6 : i=1 & j=2
7 : i=2 & j=0
8 : i=2 & j=1
9 : i=2 & j=2
10 : i=3 & j=0
11 : i=3 & j=1
12 : i=3 & j=2
Total number of iterations = 12
```

You can use the above variation patterns in the control variables to do something useful. For example, you can use the following simple (inner) `for` loop to show the multiples of the value of the variable `i`.

```
            for (int j=1; j <= 9; j++) {
                System.out.print(i * j + "\t");
            }
            System.out.println();
```

In the above `for` loop, the value of the control variable `j` starts with `1` and its value increases by `1` after an iteration. When its value reaches `10`, the condition `j <= 9` becomes `false` and the `for` loop is terminated. In its loop body, the `print()` method shows the value, `i` times `j`, and a `'\t'` (tab) character that will skip some space to the next tab stop of the current line on the screen. As the method used is `print()` instead of `println()`, the next output on the screen is displayed on the same line. After the loop terminates, the method `println()` without argument is executed to ensure the next output to be shown on the screen will start at the leftmost position of the next line.

If the value of the variable `i` is `1`, the output to be shown on the screen is:

```
1      2      3      4      5      6      7      8      9
```

By placing the above program segment in another (outer) `for` loop with control variable `i` so that its value increases from `1` to `9`, such as

```
        for (int i=1; i <= 9; i++) {
            for (int j=1; j <= 9; j++) {
                System.out.print(i * j + "\t");
            }
            System.out.println();
        }
```

each iteration of the outer `for` loop displays a line of the multiples of the value of the control variable `i`. Therefore, the outer `for` loop iterates nine times, which displays nine lines for multiples of values from `1` to `9`.

The above program segment is implemented in the method `show()` of the `MultiplicationTable` class as shown in Figure 6.9.

```
// Definition of class MultiplicationTable
public class MultiplicationTable {

    // Show a nine by nine multiple table
    public void show() {
        // Outer loop that displays lines of multiples
        for (int i=1; i <= 9; i++) {
            // Inner loop that display a line of multiples
            // for value of variable i
            for (int j=1; j <= 9; j++) {
                System.out.print(i * j + "\t");
            }
            // Ensure next output starts at the next line
            System.out.println();
        }
    }
}
```

Figure 6.9 MultiplicationTable.java

A simple driver program `TestMultiplicationTable` is written for creating a `MultiplicationTable` object called `show()` method to show the multiplication table. The class `TestMultiplicationTable` is written in Figure 6.10.

```
// Definition of class TestMultiplicationTable
public class TestMultiplicationTable {

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

        // Show the multiplication table
        table.show();
    }
}
```

Figure 6.10 TestMultiplicationTable.java

When you compile the classes and execute the `TestMultiplicationTable` program, you will get the following output:

```
1    2    3    4    5    6    7    8    9
2    4    6    8    10   12   14   16   18
3    6    9    12   15   18   21   24   27
4    8    12   16   20   24   28   32   36
5    10   15   20   25   30   35   40   45
6    12   18   24   30   36   42   48   54
7    14   21   28   35   42   49   56   63
8    16   24   32   40   48   56   64   72
9    18   27   36   45   54   63   72   81
```

The changes in the control variables facilitate the construction of the above multiplication table. A simple `for` loop can also give the same output, but the program would be a little bit more complicated. A question in Self-test 6.6 asks you to write a program to show a multiplication table with a single `for` loop. Now you can compare the two implementation methods.

The result of 3 times 5 equals that of 5 times 3, and the result 15 is shown twice in the multiplication table. If it is necessary to show non-duplicated products in the tables, which means only the lower triangular part of the multiplication table is to be shown, you can use an `if` statement in the inner loop body to control whether a number is displayed or not, such as:

```
for (int i=1; i <= 9; i++) {
    for (int j=1; j <= 9; j++) {
        if (j <= i) {
            System.out.print(i * j + "\t");
        }
    }
    System.out.println();
}
```

The control variables `i` and `j` can be considered the row number and column number of a value shown in the multiplication table, and both of their values range from `1` to `9`. If a condition `j <= i` is added to the inner loop body, the `if` part is executed if the column count is less than or equal to the row count.

Therefore, for the first row, the value of the control variable `i` is `1` and only one number is displayed when the value of control variable `j` is `1`. For other values of variable `j`, the condition of the `if` statement is `false` and the `if` part of the `if` statement is not executed. When the control variable `i` becomes `5`, for example, the numbers are displayed for control variable `j` with values `1` to `5`. With such a pattern, you can display the lower triangular part of the table as follows:

```
1
2    4
3    6    9
4    8    12   16
5    10   15   20   25
6    12   18   24   30   36
7    14   21   28   35   42   49
8    16   24   32   40   48   56   64
9    18   27   36   45   54   63   72   81
```

If you analyse the above program segment, the `if` statement effectively controls whether the statement in the `if` part is to be executed or not. Once the value of the control variable `i` is greater than that of control variable `j`, the `if` part of the `if` statement is not executed. Therefore, the iterations when the value of control variable `i` are greater than that of control variable `j` are not necessary. By this observation, it is possible to further enhance the program segment by controlling the number of iterations of the inner loop rather than preventing the statement from executing by using an `if` statement. A way to do this is to place the condition of the `if` statement in the `for` loop declaration, such as:

```
for (int i=1; i <= 9; i++) {
    for (int j=1; j <= i; j++) {
        System.out.print(i * j + "\t");
    }
    System.out.println();
}
```

The above program segment is implemented in the definition of class `MultiplicationTable2` in Figure 6.11.

```
// Definition of class MultiplicationTable2
public class MultiplicationTable2 {

    // Show a nine by nine multiple table
    public void show() {
        // Outer loop that displays lines of multiples
        for (int i=1; i <= 9; i++) {
            // Inner loop that display a line of multiples
            // for value of variable i
            for (int j=1; j <= i; j++) {
                System.out.print(i * j + "\t");
            }
            // Ensure next output starts on the next line
            System.out.println();
        }
    }
}
```

Figure 6.11 MultiplicationTable2.java

A driver program, the class `TestMultiplicationTable2`, is written for testing the `MultiplicationTable2` class. It is not shown here since it is similar to the class `TestMultiplicationTable` except the object to be created is a `MultiplicationTable2` object. You can find the driver program in the CD-ROM and the course website.

Because an inner loop can access the control variable(s) of the outer loop(s), the condition of the inner loop can involve the control variable(s) of the outer loop(s), and you can therefore design different conditions to obtain different combinations of values of the control variables.

Please use the following self-test to test your understanding and your creativity in using nested loops.

Self-test 6.6

1 Write program segments that use nested `for` loops to construct the following pattern on the screen:

a
```
*
**
***
****
*****
```

b
```
*****
****
***
**
*
```

c
```
*****
 ****
  ***
   **
    *
```

d
```
    *
   ***
  *****
 *******
*********
```

2 Modify the `show()` method of the `MultiplicationTable` class (in Figure 6.9) so that a simple `for` loop is used in the method to show the same output on the screen. (Hint: There are `81` numbers shown on the screen. If a variable `i` takes values from `0` to `80`, what are the results `i / 9` and `i % 9`?)

Up to this point, we have discussed only the nested `for` loop. It is possible to use a `while` loop or a `do/while` loop as either the outer or inner loop, and the scenarios are pretty much the same. However, you should notice that the variables involved in a `while` loop and `do/while` loop are defined before the start of the loop. Therefore, a variable involved in the condition of the inner loop must be defined in or before the outer loop. As a result, the outer loop can access the value of the variable used in the inner loop. Similar to the nested `for` loop, it is recommended you not change the variable that is dedicated to the inner loop in the outer loop. Otherwise, the program is difficult to understand and maintain.

Therefore, we are not going to discuss the combinations. We'll simply leave it up to you to try them.

In *Unit 5*, we discussed a searching method called linear search. The core principle of the linear search is that each array element of the array object is verified one by one by using a looping structure. After the loop is terminated, you can determine whether a data item is found among all data associated with the array object.

However, if you just want to know whether a data item is found, instead of counting the occurrences, you can terminate the loop as soon as the data are found and it is not necessary to continue the loop. Hence, the performance of the searching is improved.

You can alter the sequence of execution in a loop by using `break` and `continue` statements. We discuss their uses in the following subsections.

The use of `break` in loops

You have used `break` statements as the last statement in a sequence of statements for a case in `switch/case` structures. When a `break` statement is executed, the entire `switch/case` structure is terminated. `break` statements can also be used in looping structures such as `for`, `while` and `do/while` looping structures. The function of a `break` statement in a looping structure is similar to its effect in a `switch/case` structure — executing a `break` statement in a looping structure will terminate the looping structure at once.

The following reading tells what you need to know about the use of `break` in loops.

Reading

King, section 4.8, 'Exiting from a loop: the `break` statement', pp. 163–69

From the reading, you should have learned that whatever the looping structure is, executing a `break` statement within a loop will quit the loop immediately. Therefore, the use of `break` statements enables you to write a loop more flexibly, as it is another way to skip subsequent statements in a loop and terminate the loop unconditionally.

In addition to the `break` statement, the Java programming language provides the `continue` statement that also enables you to alter the execution sequence in a loop body. It is discussed next.

The use of `continue` in loops

While executing a loop body of any looping structure, the statements in the loop body are executed one by one unless there is a branching statement or a looping structure. Besides the `break` statement, the Java programming language provides another statement, `continue`, that can be used to alter the execution sequence of the statements in a loop body.

Please read the following about the use of `continue`.

Reading

King, section 8.4, 'The `continue` statement', pp. 316–17

From the reading, you should have learned that you can place a `continue` statement in a loop body and it is usually the last statement of an `if` part or `else` part of an `if/else` statement in the loop body, such as:

Table 6.14 The usual positions of using the `continue` statement

`do/while` loop	`for` loop	`while` loop
`do {`	`for (..;..;..) {`	`while (..) {`
`  ..`	`  ..`	`  ..`
`  if (`*condition*`)`	`  if (`*condition*`)`	`  if (`*condition*`)`
`  {`	`  {`	`  {`
`    ..`	`    ..`	`    ..`
`    continue;`	**`    continue;`**	**`    continue;`**
`  }`	`  }`	`  }`
`  else {`	`  else {`	`  else {`
`    ..`	`    ..`	`    ..`
`    continue;`	**`    continue;`**	**`    continue;`**
`  }`	`  }`	`  }`
`  ..`	`  ..`	`  ..`
`} while (..);`	`}`	`}`

In Table 6.14, the statements indicate the usual positions of placing a `continue` statement. Whether the `continue` statement is executed depends on the result of the `if/else` statement condition. Whenever the `continue` statement is executed, the behaviour of the program is:

1 The flow of control is transferred to the point just before the end of the loop body, and it skips practically all remaining statements in the loop body.

2 The loop structure behaves as usual. For `do/while` and `while` loops, the condition of the loop is evaluated to determine whether the loop terminates. For the `for` loop, the update part of the `for` loop is executed and the loop condition is evaluated afterwards to determine whether the loop continues or not.

Both `break` and `continue` can alter the flow of control in a looping structure. Table 6.15 summarizes the differences between the two.

Table 6.15 Comparisons of `break` and `continue` statements

	`break`	`continue`
Uses	Can be used in all looping structures and `switch/case` statements.	Can be used in looping structures only.
Effects	Quits the loop or `switch/case` statement immediately and unconditionally.	Skips the remaining statements in the loop body, the update part is executed for the `for` loop, and the condition of the loop is checked immediately to determine whether the loop continues or not.

Proper uses of `break` and `continue`

The `break` and `continue` statements enable you to write the program more flexibly, as they provide other paths of execution. However, they usually make the loop structures more difficult to read and maintain. For most scenarios, there are usually equivalent implementations with neither `break` nor `continue` statements, which are much more readable.

For example, in the program segment presented in the last reading for the `continue` statement (it has been copied from your textbook to save you referring to the text),

```
while (true) {
  SimpleIO.prompt("Enter a Social Security number: ");
  ssn = SimpleIO.readLine();
  if (ssn.length() < 11) {
    System.out.println("Error: Number is too short");
    continue;
  }
  if (ssn.length() > 11) {
    System.out.println("Error: Number is too long");
    continue;
  }
  if (ssn.charAt(3) != '-' ||
    ssn.charAt(6) != '-') {
    System.out.println(
      "Error: Number must have the form ddd-dd-dddd");
    continue;
  }
  break;    // Input passed all the tests, so exit the loop
}
```
(King 2000, 317)

the program segment uses both `break` and `continue` statements, and it is quite difficult to figure out the logic behind the loop body. There is a better implementation without using `continue` statements:

```
while (true) {
  SimpleIO.prompt("Enter a Social Security number: ");
  ssn = SimpleIO.readLine();
  if (ssn.length() < 11) {
    System.out.println("Error: Number is too short");
  }
  else if (ssn.length() > 11) {
    System.out.println("Error: Number is too long");
  }
  else if (ssn.charAt(3) != '-' ||
    ssn.charAt(6) != '-') {
    System.out.println(
      "Error: Number must have the form ddd-dd-dddd");
  }
  else {
    break;    // Input passed all the tests, so exit the loop
  }
}
```

A nested `if/else` statement replaces the sequence of three individual `if` statements. Such a nested `if/else` statement clearly shows that the `break` statement is only executed if the results of all three conditions are `true`, which implies the input social security number passes all tests.

Another implementation variation that uses neither the `break` nor `continue` statement is as follows:

```
do {
  SimpleIO.prompt("Enter a Social Security number: ");
  ssn = SimpleIO.readLine();
  if (ssn.length() < 11) {
    System.out.println("Error: Number is too short");
  }
  else if (ssn.length() > 11) {
    System.out.println("Error: Number is too long");
  }
  else if (ssn.charAt(3) != '-' ||
    ssn.charAt(6) != '-') {
    System.out.println(
      "Error: Number must have the form ddd-dd-dddd");
  }
} while (ssn.length() != 11 ||
         ssn.charAt(3) != '-' ||
         ssn.charAt(6) != '-');
```

The infinite `while` loop is replaced by a `do/while` loop, and the condition of the loop specifies the conditions of an invalid social security number. It is a better implementation than the previous one, as it uses neither `break` nor `continue` statements. However, there is a pitfall in that the condition of the `do/while` loop duplicates that of the `if/else` statement. Such an problem gets more significant if the number of `if/else` statement conditions increases.

The problem can be handled by introducing a temporary `boolean` variable for storing the validity of the social security number so that it can be used in the `do/while` loop condition. Such a program segment is as follows:

```
do {
  // The input SSN is assumed to be invalid
  boolean isValidSSN = false;
  SimpleIO.prompt("Enter a Social Security number: ");
  ssn = SimpleIO.readLine();
  if (ssn.length() < 11) {
    System.out.println("Error: Number is too short");
  }
  else if (ssn.length() > 11) {
    System.out.println("Error: Number is too long");
  }
  else if (ssn.charAt(3) != '-' ||
    ssn.charAt(6) != '-') {
    System.out.println(
      "Error: Number must have the form ddd-dd-dddd");
  }
  else {
    // The input SSN passes all tests and it is
therefore valid
    isValidSSN = true;
  }
} while (! isValidSSN);
```

Compared with the original program segment presented in your last reading (and replicated earlier), although a local variable `isValidSSN` is introduced, the above version is easier to understand is therefore recommended.

Let’s review part of another sample program segment presented in your second-to-last reading that uses a `break` statement. Again, it is presented here to save you referring to King.

```
SimpleIO.prompt("Enter a Social Security number: ");
String ssn = SimpleIO.readLine().trim();
// If the input isn't 11 characters long, or lacks dashes
// in the proper places, prompt the user to re-enter
// the SSN; repeat until the input is valid
while (true) {
  if (ssn.length() != 11) {
    System.out.prinltn("Error: Number must have 11 " +
                       "characters");
  } else if (ssn.charAt(3) != '-' ||
             ssn.charAt(6) != '-') {
    System.out.println(
      "Error: Number must have the form ddd-dd-dddd");
  } else
    break;
  SimpleIO.prompt("\nPlease re-enter number: ");
  ssn = SimpleIO.readLine().trim();
}
```
(King 2000, 166–67)

The `break` statement terminates the infinite loop if the input social security number passes all tests immediately and the statements for re-entering the number are skipped. It is an example of how the use of a `break` statement simplifies the loop body, but its underlying logic is not clear. A longer but clearer implementation that uses no `break` statement is as follows:

```
SimpleIO.prompt("Enter a Social Security number: ");
String ssn = SimpleIO.readLine().trim();
// If the input isn't 11 characters long, or lacks dashes
// in the proper places, prompt the user to re-enter
// the SSN; repeat until the input is valid
do {
  // the input SSN is assumed to be invalid
  boolean isValidSSN = false;

  // Check the input SSN rule by rule
  if (ssn.length() != 11) {
    System.out.prinltn("Error: Number must have 11 " +
                       "characters");
  } else if (ssn.charAt(3) != '-' ||
             ssn.charAt(6) != '-') {
    System.out.println(
      "Error: Number must have the form ddd-dd-dddd");
  } else {
    // if the input SSN passes all tests, it is valid
    isValidSSN = true;
  }

  // If the input SSN is invalid, prompt the user to re-enter
  if (!isValidSSN) {
    SimpleIO.prompt("\nPlease re-enter number: ");
    ssn = SimpleIO.readLine().trim();
  }
} while (! isValidSSN);
// the loop continues while the input SSN is not valid
```

Compared with the program segment presented in the reading, the above program segment is longer, and a local variable `isValidSSN` is required to store the validity of the input SSN. However, it is more readable because to the following:

1 The loop in the original program segment is an infinite loop, and it is necessary to investigate whether there is a `break` statement in the loop body and under what conditions the `break` statement is to be executed. The loop in the modified version is clear — the loop repeats while the input social security number is not valid.

2 The condition of each level of the nested `if/else` statement corresponds to a particular condition that invalidates an input social security number. If the last `else` part of the nested `if/else` statement is executed, that means the input social security number passes all tests and is considered to be valid.

3 The statements to prompt and enable the user to re-enter the number are executed only if the input social security number is not valid.

The modified versions of the sample programs discussed in the last two readings illustrate that there are usually equivalent and more readable versions of a `for` loop body with either `break` or `continue` statements. Unless you determine that the use of the `break` or `continue` statement can make a program segment simpler and easier to understand, it is recommended you not use them.

Self-test 6.7

Study the following program segment that is designed to count the number of odd numbers and even numbers in a range from 1 to 100. Two statements are missing in the program segment that are either `break` or `continue` statements. Please determine and discuss whether `break` statements or `continue` statements could be used so that the program segment can give correct results.

```
int oddCount = 0;
int evenCount = 0;

for (int i=1; i <= 100; i++) {
    switch (i % 2) {
    case 0:
        evenCount++;
        ___________;
    case 1:
        oddCount++;
        ___________;
    }
}
```

Sorting

In *Unit 5*, you learned how to use an array object to store a collection of data of the same type; for example, you can have an array object that stores a collection of numbers of type `int` as the array elements. We also discussed how to use a loop, usually a `for` loop, to access the array elements one by one so that you can determine whether a number can be found in that collection of numbers. Similarly, you can use an array object with array elements of non-primitive types to consolidate a collection of objects of the same type, and you can then use a similar approach to determine whether an object can be found in a collection of objects of the type. Such an operation is known as *searching*.

You learned that there are two common approaches to search for a data item (or object) within a collection of data consolidated by an array object. The difference between them is the pattern of accessing the array elements to determine the target data. If you access the array elements one by one, this is *linear* searching. If the data are arranged in a particular order, such as in ascending order of numeric value, you can use *binary* searching to locate the data. The binary searching approach is much faster because it keeps on dividing the searching scope in half until it contains a single array element.

The key criterion for using binary searching is that the numbers must be in a particular order, but the data typically collected in the real world are usually in arbitrary order. Therefore, you need a process to arrange the data according to a particular order, such as in ascending order of numeric values. Such an operation is known as *sorting*. For example, if you have an array object with element type of `int` as follows,

12	32	4	87	22

it is expected after sorting, the contents of the array elements have become:

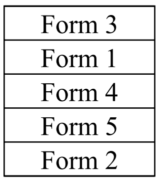
4	12	22	32	87

Computer scientists researched the issue of sorting and have derived many approaches to sorting. This section covers one of the common sorting methods — insertion sort.

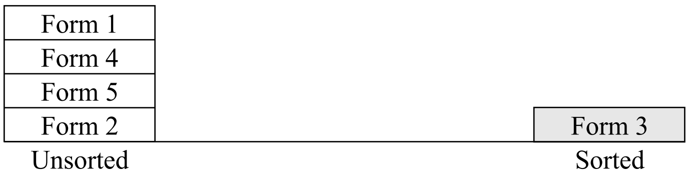
Insertion sort

Before we discuss what insertion sort is, think for yourself that if you took out your result sheets for Form 1 to Form 5 and found they were not in order, what would you do to sort the result sheets so that Form 1 was on the top of the pile and the Form 5 sheet was at the bottom?

For example, assume the pile of result sheets is as follows:

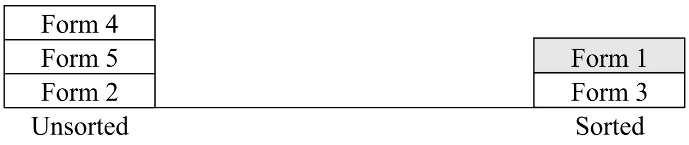


One possible way to sort the result sheets is to take the first result sheet and place it on the desk; that is:

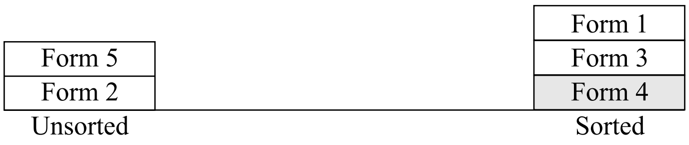


You immediately have two piles of result sheets — one sorted and one unsorted.

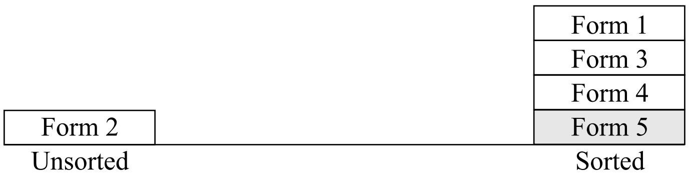
You take the next answer sheet from the top of the unsorted pile and place it in the sorted pile, making sure the sorted pile is kept sorted, which is:



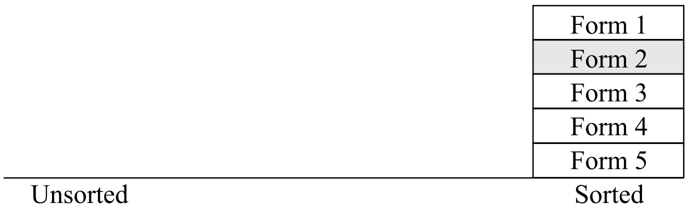
For the next step, the result sheet for Form 4 is taken from the unsorted pile and is inserted into the sorted pile properly so that the pile is kept sorted. The piles become:



The next result sheet to be inserted into the sorted pile is the one for Form 5. It is taken from the unsorted pile and is inserted at the bottom of the sorted pile.



Finally, the result sheet for Form 2 is inserted between the Form 1 and Form 3 sheets in the sorted pile, which becomes:



There are no more unsorted result sheets; the result sheets are considered sorted. You probably use the above steps to sort your documents in your office every day, but you might not be aware of it.

The underlying principles of the insertion sort are exactly what you have done with the result sheets. With respect to writing a program to implement insertion sort, the data are stored in an array object so that it is possible to access each array element easily. Furthermore, it is preferable not to have two array objects to store sorted data and unsorted data because it needs two array objects of the same size and one of them is abandoned afterwards. It takes double the amount of memory space. Instead, the array object that stores the data is separated into two partitions — for sorted data and unsorted data.

Example — sorting a numeric array

With the idea of the insertion sort method in mind, we can now implement the idea to sort a few numbers maintained by an array object referred by an array variable `numbers`, such as the one shown in Figure 6.12.

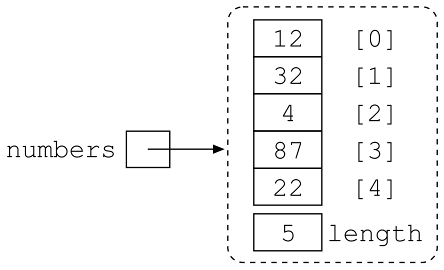


Figure 6.12 An array object maintaining 5 numbers to be sorted

The core idea of the insertion sort method is that the items to be sorted are categorized into sorted and unsorted. With respect to an array object, consider it as partitioned into two regions — the sorted and unsorted — and the sorted region initially contains the first element of the array object. That is:

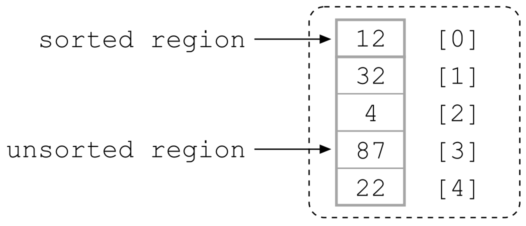


Figure 6.13 The initial sorted and unsorted regions of the array object

(In Figure 6.13 and the subsequent figures in this section, the attribute `length` of the array object is omitted, as it is of out of the scope of the current discussion.)

For implementation purposes, the sorted and unsorted regions are partitioned with a variable, say `sorted`, that stores the subscript of the last element of the sorted region, so that the unsorted region starts with the next element, with subscript `sorted+1`.

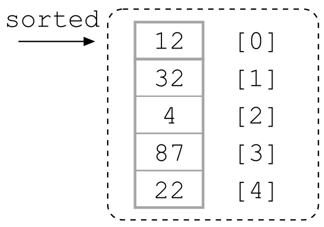


Figure 6.14 The initial sorted and unsorted regions of the array object

The sorted region starts with array subscript `0.` Its end is specified by the value of the variable `sorted` and its initial value is `0`, which means that the sorted region contains one number only. It is shown in the figures by an arrow (under the word `sorted`) pointing to the element with the current value of the variable `sorted`. The subscript range of the unsorted region is from `sorted + 1` (that is, `1` initially) up to the last subscript of the array object (that is, `numbers.length – 1`, which is `4`).

When the last number is moved from the unsorted region to the sorted region, the number of elements in the sorted region is `numbers.length - 1` and there is only one number left in the unsorted region. Then, the subscript of the last number in the sorted region is `numbers.length - 2`. As the variable `sorted` takes values from `0` to `numbers.length – 2`, the outer loop can be implemented by the following `for` loop:

```
for (int sorted = 0; sorted < numbers.length – 1; sorted++) {
  // The process to move a number from unsorted region to
  // the sorted region
}
```

In each step of moving a number from the unsorted region to the sorted region, the first number of the unsorted region, `numbers[sorted+1]`, is copied to a temporary storage, say a variable `storage` by

```
storage = numbers[sorted+1];
```

where `storage` has been declared a variable with the same type as the base type of `numbers` (see Figure 6.15). The first element of the unsorted region is considered to be empty, although a value to be replaced is actually still there.

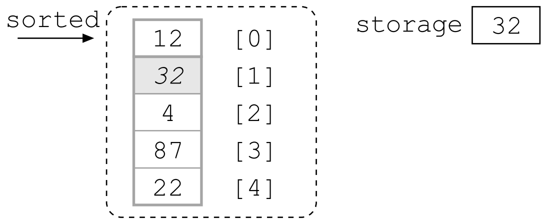


Figure 6.15 The array object after moving 32 to a temporary variable `storage`

The number `32` stored in the array element, `numbers[1]`, in Figure 6.15 is set in italic to highlight that the value in the cell is going to be replaced.

It is now necessary to rearrange the sorted region (and the first element of the unsorted region) so that the variable `storage` can be put in a suitable location and the numbers in the sorted region are kept in order. It is simple and obvious that the number `32` should be stored in `numbers[1]`, which is the original array element that maintained the number. Therefore the variable `storage` is assigned to `numbers[1]`.

Now it's the turn of `numbers[2]` and it is stored in `storage` (see Figure 6.16).

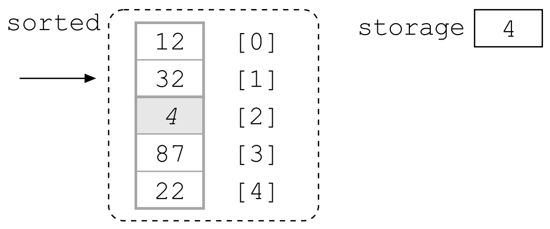


Figure 6.16 The array object after moving 4 to a temporary variable `storage`

After making room for it, the array becomes:

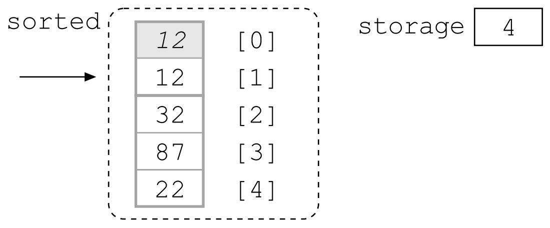


Figure 6.17 The array object after making room for the variable `storage`

It is necessary to rearrange the sorted region so that all numbers that are larger than that of variable `storage` are moved to their next elements. In other words, all numbers that are greater than variable `storage` are shifted to the next position in the sorted region. For example, the number `32` is moved from `numbers[1]`to `numbers[2]` and then the number `12` is moved from `numbers[0]`to `numbers[1]`. Consequently, an empty location is obtained between all numbers that are less than or equal to the variable `storage` and those that are greater than the variable `storage`. Figure 6.17 is a special case in which no elements are less than `storage`. The detailed steps of the move are explained later. Now the value `4` can be copied to `numbers[0]`.

After determining the number `87` should remain in `numbers[3]`(similar to the process of the number `32`), we are in the last round of the sorting process. The value of `numbers[4]` is copied to `storage` and the array can now be visualized in Figure 6.18.

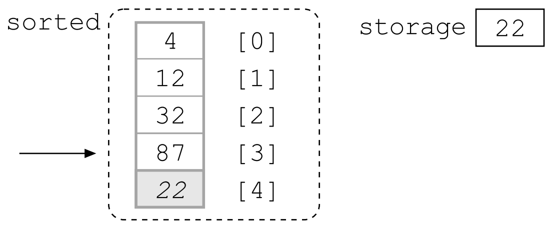


Figure 6.18 The last round of the sorting process

It is expected that the numbers in the sorted region will be rearranged as in Figure 6.19 after the numbers `32` and `87` have been moved down one position.

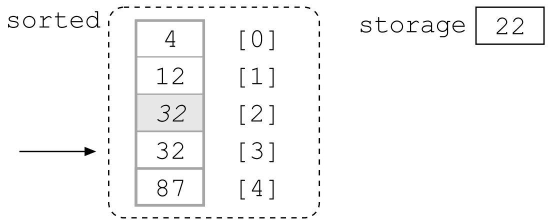


Figure 6.19 After the numbers 32 and 87 have been moved to suitable positions

Now we come back to explain how we can rearrange the sorted numbers to free a location for the number to be inserted. A systematic approach to rearrange the numbers is to compare each number in the sorted region with the variable `storage`. Then, you would need another variable (say, `forCompare`) to store the subscript of the current element for comparison (see Figure 6.20). The first number to be compared can be the last number in the sorted region, and its initial value is therefore equal to that of `sorted`. You may start the comparison from the first number and, with other suitable changes, the final result is the same.

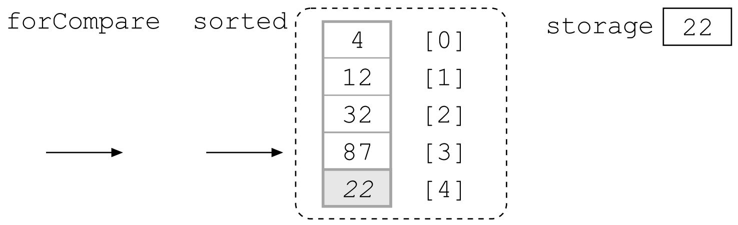


Figure 6.20 Comparing each number in sorted region with `storage`

The value of the array element with subscript specified by the variable `forCompare` — that is, `numbers[forCompare]` — is compared with the variable `storage`. If the value of `numbers[forCompare]` is greater than that of variable `storage` — that is, the condition `numbers[forCompare] > storage` is evaluated to be `true` — the value of `numbers[forCompare]` is copied to the next array element. The corresponding statement is:

```
numbers[forCompare+1] = numbers[forCompare];
```

Then, the scenario becomes as shown in Figure 6.21.

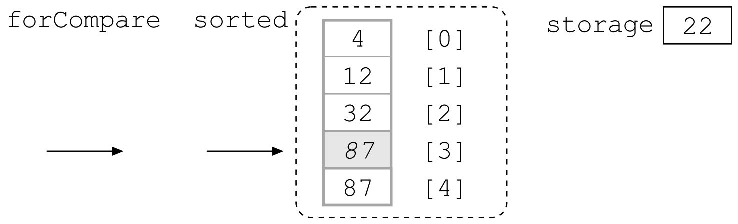


Figure 6.21 Array after the number 87 has been moved

Then, the next largest number in the sorted region is handled similarly. Therefore, the value of the variable `forCompare` is decreased by one, such as by the statement

```
forCompare--;
```

and the scenario becomes that in Figure 6.22.

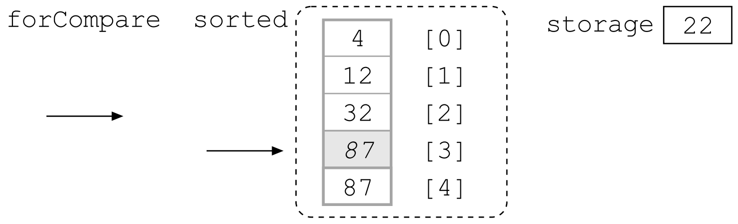


Figure 6.22 Comparing the next number in the sorted region with `storage`

The condition, `numbers[forCompare] > storage` (which is `32 > 22`), is evaluated to be `true`. The number `32` is moved and the sorted region becomes as shown in Figure 6.23.

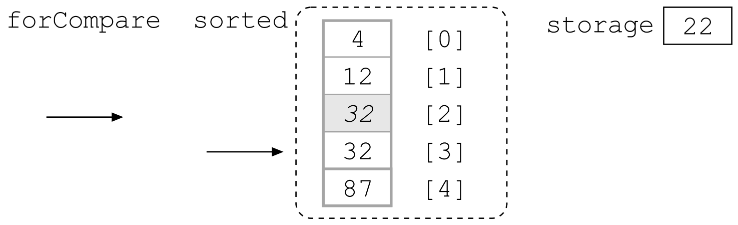


Figure 6.23 Array after the number 32 has been moved

It is necessary to continue to test the next numbers in the sorted region. To compare the next number, it is necessary to update the variable `forCompare` first by executing the following statement again

```
forCompare--;
```

and the scenario becomes as shown in Figure 6.24.

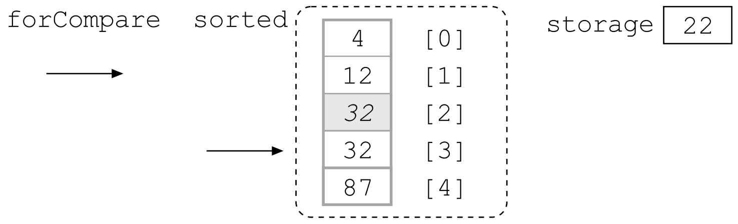


Figure 6.24 Array after the variable forCompare is decreased by one

The condition, `numbers[forCompare] > storage` (which is `12 > 22`), is evaluated to be `false`. The number `12` is not moved, and it is not necessary to verify the other numbers in the sorted region because they must be less than that of variable `storage` since they are sorted.

After the numbers in the sorted region are rearranged, the array element for storing the value stored by the variable `storage` is `numbers[forCompare+1]`. Then, the scenario finally becomes as shown in Figure 6.25.

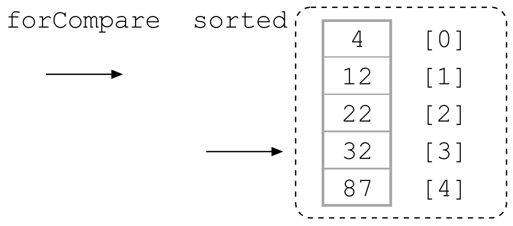


Figure 6.25 Array after the number stored in the variable storage has been assigned to the array element numbers[forCompare+1]

Based on the above discussion and the mentioned statements, we can merge the statements to derive the program segment for insertion sort.

If the number to be compared, `numbers[forCompare]`, is greater than the value of the variable `storage`, it is necessary to move it to the next element in the sorted region. The corresponding statement is:

```
if (numbers[forCompare] > storage) {
    // Copy the number to the next element
    numbers[forCompare+1] = numbers[forCompare];
}
```

Afterwards, the number next to the largest number is compared with the variable `storage`. Therefore, the value of the variable `forCompare` decreases until its value reaches zero. As the variable `forCompare` takes values from the value of `sorted` to `0`, it can be implemented with the following `for` loop:

```
for (int forCompare = sorted; forCompare >= 0; forCompare--) {
  // Process a number in the sorted region
}
```

Combined with the previous `if` statement, the complete program segment for copying the number that is greater than the value of variable `storage` is:

```
for (int forCompare = sorted; forCompare >= 0; forCompare--) {
  if (numbers[forCompare] > storage) {
    // Copy the number to the next element
    numbers[forCompare+1] = numbers[forCompare];
  }
}
```

As the numbers in the sorted region are tested starting from the largest to the smallest, it is not necessary to continue the testing as soon as the number to be compared, `numbers[forCompare]`, is found to be less than or equal to that of variable `storage`. Therefore, the condition `numbers[forCompare]>storage` should be added to the condition of the loop so that unnecessary iterations are avoided, and the `for` loop becomes:

```
for (int forCompare = sorted;
     forCompare >= 0 && numbers[forCompare] > storage;
     forCompare--) {
    // Copy the number to the next element
    numbers[forCompare+1] = numbers[forCompare];
}
```

The above `for` loop terminates under one of the following two scenarios:

1 The condition `forCompare>=0` becomes `false`; that is, the value of the variable `forCompare` is less than 0. The implication is that all numbers in the sorted region are all larger than the variable `storage` and the numbers are shifted to the next elements.

2 The condition `numbers[forCompare]>storage` becomes `false`; that is, the current element for comparison is less than or equal to the variable `storage`.

For both cases, the empty element for storing the value stored by the variable `storage` is the next one that is specified by the variable `forCompare`; that is, `numbers[forCompare+1]`. Therefore, the number stored in the variable `storage` is copied to the empty element,

```
numbers[forCompare+1] = storage;
```

Combining the above `for` loop and the above statement, we have:

```
for (int forCompare = sorted;
     forCompare >= 0 && numbers[forCompare] > storage;
     forCompare--) {
    // Copy the number to the next element
    numbers[forCompare+1] = numbers[forCompare];
}
numbers[forCompare+1] = storage;
```

However, the above program segment can cause compile-time errors because the variable `forCompare` is declared in the `for` loop, and it can only be accessed anywhere in the `for` loop only. To resolve the problem, the variable `forCompare` is declared before the `for` loop. That is:

```
int forCompare;
for (forCompare = sorted;
     forCompare >= 0 && numbers[forCompare] > storage;
     forCompare--) {
    // Copy the number to the next element
    numbers[forCompare+1] = numbers[forCompare];
}
numbers[forCompare+1] = storage;
```

Even though the above program segment is workable, its structure may not seem to be as straightforward as the looping structures we have seen so far. By rearranging the statements in the above looping structure and replacing the `for` loop with a `while` loop, the following program segment is obtained:

```
int forCompare = sorted;
while (forCompare >= 0 && numbers[forCompare] > storage) {
    // Copy the number to the next element
    numbers[forCompare+1] = numbers[forCompare];
    // Test next number
    forCompare--;
}
numbers[forCompare+1] = storage;
```

The above program segment is the process to move a number from the unsorted region to the sorted region. Therefore, combining the previous `for` loop for iterating different values of variable `sorted` and the above program segment, the following program segment is obtained:

```
for (int sorted=0; sorted < numbers.length-1; sorted++) {
    int forCompare = sorted;
    while (forCompare >= 0 && numbers[forCompare] > storage) {
        // Copy the number to the next element
        numbers[forCompare+1] = numbers[forCompare];
        // Test next number
        forCompare--;
    }
    numbers[forCompare+1] = storage;
}
```

Based on the above discussion, the following definition of class `InsertionSorter` is shown in Figure 6.26.

```
// Definition of class InsertionSorter
public class InsertionSorter {

   // Sort an array of numbers of type int
   public static void sort(int[] numbers) {
      // For each value of the last subscript of the sorted region
      for (int sorted = 0; sorted < numbers.length - 1; sorted++) {
         // Store first number from the unsorted region
         int storage = numbers[sorted + 1];
         // The first number to be compared is the last number in
         // the sorted region
         int forCompare = sorted;
         // While there are numbers to be compared and the number
         // to be compared is greater than that of storage
         while (forCompare >= 0 && numbers[forCompare] > storage) {
            // Copy the compared number to the next one
            numbers[forCompare + 1] = numbers[forCompare];
            // Decrease forCompare to test next number
            forCompare--;
            }
         // The next element of the last compared number is the
         // position to store the number maintained by storage
         numbers[forCompare + 1] = storage;
      }
   }
}
```

Figure 6.26 InsertionSorter.java

The `sort()` method accepts a reference to an array object with element type `int`. To test the `InsertionSorter` class, the driver class `TestInsertionSorter` is shown in Figure 6.27.

```
// Definition of class TestInsertionSorter
public class TestInsertionSorter {

    // Main executive method
    public static void main(String args[]) {
        // Check if no number specified, show usage message
        if (args.length == 0) {
            System.out.println(
                "Usage: java TestInsertionSorter num1 num2 ...");
        }
        else {
            // Create an array for storing the number
            int[] numbers = new int[args.length];
            // Convert each program parameter into
            // number of type int
            for (int i=0; i < args.length; i++) {
                numbers[i] = Integer.parseInt(args[i]);
            }

            // Create the InsertionSorter object for sorting
            InsertionSorter sorter = new InsertionSorter();
            // Sort the numbers by supplying the array object
            sorter.sort(numbers);

            // Show the sorted numbers on the screen
            System.out.println("Sorted numbers:");
            for (int i=0; i < numbers.length; i++) {
                System.out.print(numbers[i] + "\t");
            }
            System.out.println();
        }
    }
}
```

Figure 6.27 TestInsertionSorter.java

Compile the above two class definitions and execute the `TestInsertionSorter` program with a few numbers as program parameters, such as:

```
java TestInsertionSorter 12 32 4 87 22
```

The program executes and shows the following output:

```
Sorted numbers:
4       12      22      32      87
```

In order to understand the operations of the insertion sort, you can add some statements to the `sort()` method of the `InsertionSorter` class so that the operations carried out during sorting are shown. A class `InsertionSorter2` as shown in Figure 6.28 is derived based on the `InsertionSorter` by adding statements for showing the operational details.

```
// Definition of class InsertionSorter2
public class InsertionSorter2 {

   // Sort an array of numbers of type int
   public static void sort(int[] numbers) {
      // For each value of the last subscript of the sorted region
      for (int sorted = 0; sorted < numbers.length - 1; sorted++) {
         // Show sorted and unsorted region
         System.out.print("DEBUG: sorted region = ");
         for (int i=0; i <= sorted; i++) {
            System.out.print(numbers[i] + "\t");
         }
         System.out.print("\nDEBUG: unsorted region = ");
         for (int i=sorted + 1; i < numbers.length; i++) {
            System.out.print(numbers[i] + "\t");
         }
         System.out.println();

         // Store first number from the unsorted region
         int storage = numbers[sorted + 1];
         // Show the number to be moved from unsorted region
         // to sorted region
         System.out.println("DEBUG: Number to be moved = " +
            storage);

         // The first number to be compared is the last number in
         // the sorted region
         int forCompare = sorted;
         // While there are numbers to be compared and the number
         // to be compared is greater than that of storage
         while (forCompare >= 0 && numbers[forCompare] > storage) {
            // Copy the compared number to the next one
            numbers[forCompare + 1] = numbers[forCompare];
            // Show the number copied to the next element
            System.out.println("DEBUG: " + numbers[forCompare] +
                " is copied to the next element");
            // Decrease forCompare to test next number
            forCompare--;
         }

         // The next element of the last compared number is the
         // position to store the number maintained by storage
         numbers[forCompare + 1] = storage;
         // Show the destination element of the number from
         // unsorted region
         System.out.println("DEBUG: " + storage +
            " is stored in numbers[" + (forCompare + 1) + "]\n");
      }
   }
}
```

Figure 6.28 InsertionSorter2.java

Another driver program `TestInsertionSorter2` is written that is exactly the same as the `TestInsertionSorter` except the object created for sorting is an `InsertionSorter2` object instead of an `InsertionSorter` object. It is therefore not shown here. Please find it in the CD-ROM or the course website.

Compile the `InsertionSorter2` and `TestInsertionSorter2` classes, and execute the `TestInsertionSorter2` class with a few numbers as program parameters, such as:

```
java TestInsertionSorter2 12 32 4 87 22
```

The following output is shown on the screen:

```
DEBUG: sorted region = 12
DEBUG: unsorted region = 32     4       87      22
DEBUG: Number to be moved = 32
DEBUG: 32 is stored in numbers[1]

DEBUG: sorted region = 12       32
DEBUG: unsorted region = 4      87      22
DEBUG: Number to be moved = 4
DEBUG: 32 is copied to the next element
DEBUG: 12 is copied to the next element
DEBUG: 4 is stored in numbers[0]

DEBUG: sorted region = 4        12      32
DEBUG: unsorted region = 87     22
DEBUG: Number to be moved = 87
DEBUG: 87 is stored in numbers[3]

DEBUG: sorted region = 4        12      32      87
DEBUG: unsorted region = 22
DEBUG: Number to be moved = 22
DEBUG: 87 is copied to the next element
DEBUG: 32 is copied to the next element
DEBUG: 22 is stored in numbers[2]

Sorted number:
4       12      22      32      87
```

The output message contains the messages for debugging or tracing. Please study the above messages to verify the operations we discussed.

Activity 6.1

Execute the `TestInsertionSorter2` program with different sets of numbers as program parameters. Study the program output to verify the operations involved.

Example — sorting a `String` array

The idea of insertion sort is not only applicable to sorting a numeric array of primitive type `int` but is also applicable to sort an array object with non-primitive element types.

For numeric values, it makes sense that a sequence of numeric values is in order if the values are ordered in either ascending or descending order. With respect to objects, it is not always easy to define how a sequence of objects is considered to be in order. In other words, it is necessary to define how to determine whether an object is ‘less than’, ‘equal to’ or ‘greater than’ another object.

The `String` class, the class you have encountered the most up to now, defines the comparison between two `String` objects so that it is possible to tell if a `String` object is less than or greater than another `String` object. The comparison is lexicographic — that is, if the occurrence of the content of a first `String` object alphabetically or numerically precedes that of a second `String` object, it is considered to be less than the second `String` object. Here are a few examples:

```
"apple" < "boy"
"book" < "booklet"
"marry" < "merry"
```

The above ordering scheme for strings is called lexicographical ordering. More exactly, the comparison between two `String` objects is a sequence of character-by-character comparisons. The first characters of the two `String` objects are compared. If they are the same character, the next characters of both `String` objects are compared. This process is repeated until a pair of characters from both `String` objects is found to be different. Then, the comparison of the two different characters according to their *codes* determines the overall `String` comparison. The codes for alphanumeric characters are the same as those specified in the American Standard Code for Information Interchange (ASCII). Please refer to Appendix B of the textbook (p. 668) for a table of the ASCII codes and the corresponding characters. For example, the comparison of the two String objects `"marry"` and `"mary"`.

m	=	m
a	=	a
r	=	r
r	<	y
y		

The first three characters of the two `String` objects are the same; the fourth characters of the two `String` objects are different. As the code of character `'r'` is less than that of character `'y'`, the `String` object `"marry"` is considered to be less than the `String` object `"mary"`.

Another `String` comparison example is to compare the `String` objects `"book"` and `"booklet"`. The characters in the two `String` objects are compared one by one. The `String` object `"book"` contains four characters that are the same as those in the `String` object `"booklet"`. For the fifth character comparison, the `String` object `"book"` does not have a fifth character, whereas the `String` object `"booklet"` does. The fifth character comparison compares *no* character with character `'l'`, and *no* character is less than *any* character. Therefore, the overall `String` comparison result `String` object `"book"` is less than `String` object `"booklet"`.

b	=	b
o	=	o
o	=	o
k	=	k
	<	l
		e
		t

The `String` class defines a `compareTo()` method to help us compare two `String` objects. It returns a value of type `int` to indicate the order of the `String` objects referred by two variables. For example, for `String` objects referred by variable `str1` and `str2`, `str1.compareTo(str2)` returns a negative number if the `String` object referred by `str1` is lexicographically less than that referred by `str2`. But, if `str1.compareTo(str2)` returns a positive number, the `String` object referred by variable `str1` is lexicographically greater than that referred by `str2`. If a zero value is returned by `str1.compareTo(str2)`, the contents of the `String` objects referred by `str1` and `str2` are the same.

Using the same idea of insertion sort discussed in the previous section and example, you can modify the `sort()` method so that it accepts a reference of a `String` array, as the items to be sorted are not numeric values but are `String` objects. Then, the type of the temporary storage is a variable of type `String`. Based on the `InsertionSorter` class, a `StringSorter` class is shown in Figure 6.29.

```
// Definition of class StringSorter
public class StringSorter {

   // Sort an array of strings of type int
   public static void sort(String[] strings) {
      // For each value of the last subscript of the sorted region
      for (int sorted = 0; sorted < strings.length - 1; sorted++) {
         // Store first String from the unsorted region
         String storage = strings[sorted + 1];
         // The first String to be compared is the last String in
         // the sorted region
         int forCompare = sorted;
         // While there are strings to be compared and the String
         // to be compared is greater than that of storage
         while (forCompare >= 0 &&
              strings[forCompare].compareTo(storage) > 0) {
            // Copy the compared String to the next one
            strings[forCompare + 1] = strings[forCompare];
            // Decrease forCompare to test next String
            forCompare--;
         }
         // The next element of the last compared String is the
         // position to store the String maintained by storage
         strings[forCompare + 1] = storage;
      }
   }
}
```

Figure 6.29 StringSorter.java

In Figure 6.29, the differences between the definitions of the `StringSorter` class and the `InsertionSorter` class are highlighted. The type of the parameter is modified to be an array of `String` objects, and the name is changed to reflect the fact that the items to be sorted are `String` objects. The core change to the method is the comparison between numbers is changed to be:

```
strings[forCompare].compareTo(storage) > 0
```

The above `boolean` expression compares the `String` array element with subscript `forCompare` with the `String` object referred by the temporary storage, which is the variable `storage`. If the return value is positive, the `String` array element is copied to the next array element by the statement:

```
strings[forCompare + 1] = strings[forCompare];
```

To test the `StringSorter` class, a `TestStringSorter` class is shown in Figure 6.30.

```
// Definition of class TestStringSorter
public class TestStringSorter {

    // Main executive method
    public static void main(String args[]) {
        // Check if no program parameter is provided, show usage
        message
        if (args.length == 0) {
            System.out.println(
              "Usage: java TestStringSorter string1 string2 ...");
        }
        else {
            // Create a StringSorter object for sorting String
            objects
            StringSorter sorter = new StringSorter();
            // Sort the array that maintains program parameters
            sorter.sort(args);

            // Show the sorted String objects
            System.out.println("Sorted strings:");
            for (int i=0; i < args.length; i++) {
                System.out.println(i + 1 + ":" + args[i]);
            }
        }
    }
}
```

Figure 6.30 TestStringSorter.java

Compile the `StringSorter` and `TestStringSorter` classes and execute the `TestStringSorter` program with program parameters such as:

```
java TestStringSorter Open University of Hong Kong
```

The following output is shown on the screen:

```
Sorted strings:
1:Hong
2:Kong
3:Open
4:University
5:of
```

Note that the sorting of the strings above is not truly alphabetical. Can you remember why? The answer lies in what you saw on page 668 in King (if you actually checked that out). The ASCII codes for uppercase (capital) letters are lower than those for lowercase letters, so the capitalized words appear first (sorted) and then the lowercase words appear (sorted as well, though there's only the one word here). So, the sorted order of the strings is logical — 72(H), 75(K), 79(O), 85(U) and 111(o).

Multidimensional arrays

You have learned what arrays and their applications are. In a few words, an array enables you to maintain a group of entities of the same type, which can be primitive or non-primitive values and objects. The entities are accessed with an array subscript, and the subscripts range from `0` to `length-1` where `length` is the number of elements maintained by the array object, or simply the array size. In this way, you can visualize that each array element is a storage element numbered by a subscript.

An analogy is the rooms on the same floor of a hotel; each room has a room number and each room is specified in the same way. With respect to the Java programming language, each room can be considered an array element that is accessible by the expression

```
rooms[roomNumber]
```

in a program. A step further is whether it is possible to refer to a room with a particular room number on a particular floor. It would be handy if an expression existed that looks like

```
rooms[floor][roomNumber]
```

to specify a room on the specified floor (specified by the variable `floor`) with the specified room number (specified by the variable `roomNumber`). Then, if the element type of the array referred by variable `rooms` is `String`, we can have a statement like

```
rooms[floor][roomNumber] = "Peter CHAN";
```

to indicate that the room with a particular room number (specified by variable `roomNumber`) on a particular floor (specified by the variable `floor`) is currently occupied by a guest named “Peter CHAN”.

The Java programming language supports arrays having the above format, known as two-dimensional arrays. The format of the arrays you learned in *Unit 5* can be considered one-dimensional arrays in which elements are considered arranged as a row. Then, the elements of a two-dimensional array can be considered as arranged in a table, such as:

		Room number (specified by variable `roomNumber`)									
		0	1	2	3	4	5	6	7	8	9
Floor (specified by variable `floor`)	0										
	1										
	2										
	3										

Please use the following reading to learn the details of using two-dimensional arrays in the Java programming language. We elaborate on some ideas later on.

Reading

King, section 13.1 ‘Multidimensional arrays’, pp. 542–45, covering subsections ‘Creating and initializing a multidimensional array’, ‘Subscripting a multidimensional array’ and ‘Processing the elements in a multidimensional array’.

The above reading provides a general discussion of multidimensional arrays. Two-dimensional arrays are a specific type of multidimensional array. Once you have acquired the ideas behind two-dimensional arrays, you can easily proceed to three- or four-dimensional arrays.

In *Unit 5* you learned that there were three steps in using an array. They are:

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 the method parameter) so that the entire array object can be accessed as a whole.

The above steps are also applicable to two-dimensional or multidimensional arrays. First of all, it is necessary to create a variable of array type. To declare a variable that can refer to a two-dimensional array, a general format of the variable declaration is:

```
int[][] twoDimArray;
```

You can treat it as an informal rule that the number of pairs of square brackets in the array variable declaration indicates its dimensions. Therefore, the following two declarations declare a one-dimensional array and a two-dimensional array respectively:

```
int[] oneDimArray;
int[][] twoDimArray;
```

As mentioned in the reading, the pairs of square brackets can be placed on either side of the variable name. However, a declaration can be unclear if it declares more than one array variable, especially if they are of different dimensions. For example, the following array variable declaration

```
int[] oneDimArray, twoDimArray[];
```

declares a one-dimensional array variable `oneDimArray` and a two-dimensional array variable `twoDimArray`.

Now, you can create a two-dimensional array object. The general format, as mentioned in the reading, is:

```
int[][] twoDimArray = new int[5][10];
```

You can then visualize the array object referred by the array variable `twoDimArray` is a table with 5 rows and 10 columns. The contents of each array element are initialized in the same way as a one-dimensional array (as shown in Table 5.1 in *Unit 5*). In a few words, elements with primitive numeric types are initialized with zero values, elements of type `boolean` are initialized to be `false,` and elements of non-primitive types are initialized to be `null`.

The created two-dimensional array object referred to by a suitable array variable can now be processed. Like a one-dimensional array in which an element is accessed using a subscript, specifying two subscripts in square brackets allows access to an element of a two-dimensional array.

For example, in a two-dimensional array created with the following statements,

```
int[][] twoDimArray = new int[5][10];
```

the possible subscripts for the two subscripts are 0 to 4 and 0 to 9 respectively. Therefore, to access an array element with subscripts 1 and 2, the expression is

```
twoDimArray[1][2]
```

and you know it is a variable of type `int`, such as

```
twoDimArray[1][2] = 5 + 8;
```

or

```
int storage = twoDimArray[1][2];
```

If either one of the two subscripts is out of the feasible range, a runtime error will occur. For example, with the above array object referred by the array variable `twoDimArray`, the following expressions will cause runtime errors:

```
twoDimArray[5][0] = 0;  // first subscript out of bounds
twoDimArray[0][10] = 0; // second subscript out of bounds
twoDimArray[5][10] = 50;// both subscripts out of bounds
```

With a basic understanding of the steps in using two-dimensional arrays, we can further investigate how two-dimensional arrays are implemented internally in the JVM. Please use the following reading to learn how two-dimensional arrays are stored in the JVM.

Reading

King, 'How multidimensional arrays are stored', p. 545

From the above reading, you learned that two-dimensional arrays (and hence arrays of higher dimensions) are implemented as single-dimensional arrays in the JVM. For the statement discussed in the reading

```
int[][] a = new int[5][10];
```

the variable `a` is declared to be a two-dimensional variable with element type `int`. The expression

```
new int[5][10]
```

creates following a hierarchy of array objects as shown in Figure 6.31.

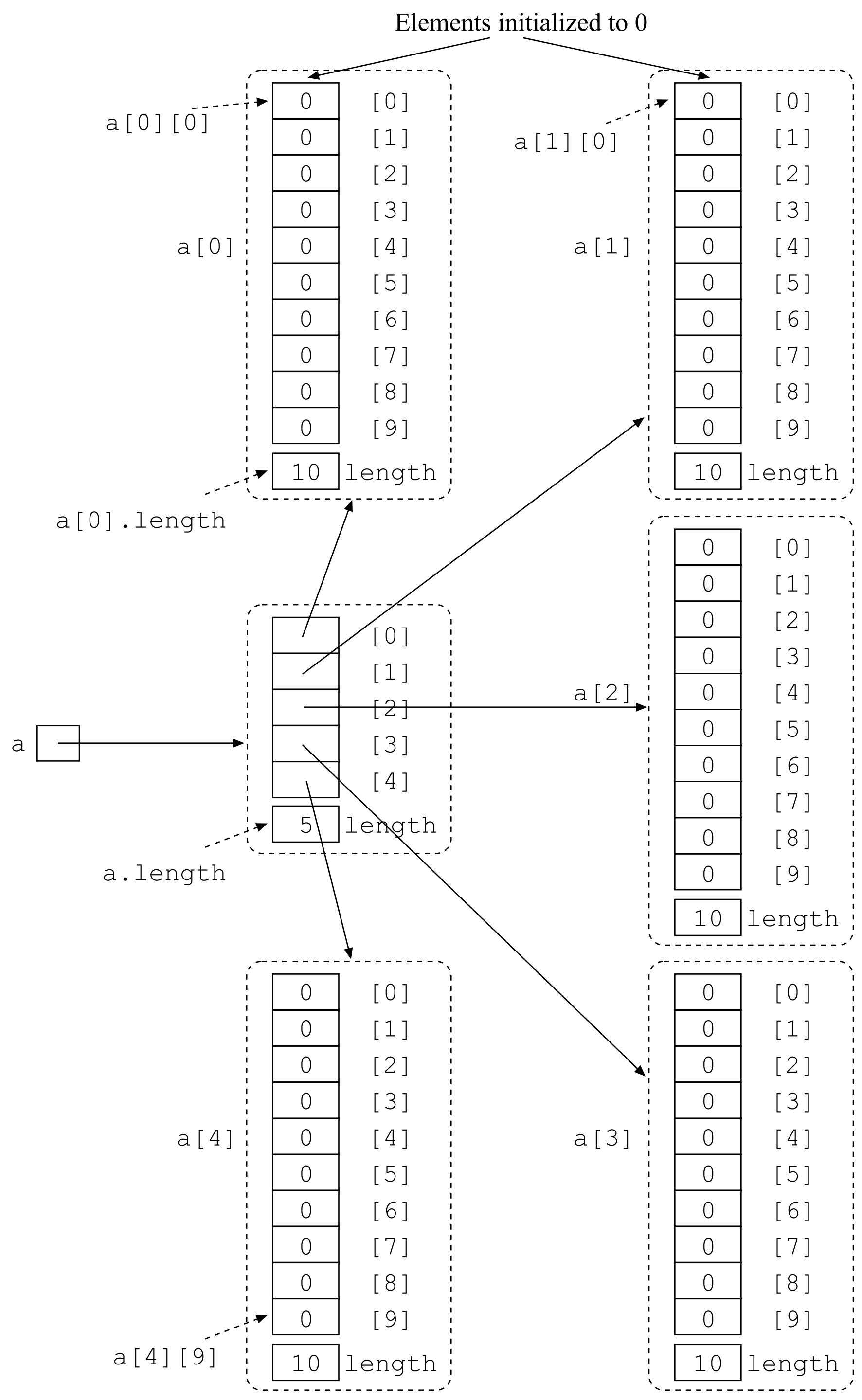


Figure 6.31 The array objects created by the statement `int[][] a = new int[5][10]`

In Figure 6.31, the statement creates *six* array objects

```
int[][] a = new int[5][10]
```

and the array object referred by the variable `a` only maintains the references to the associated array objects with array elements for storing values of type `int`. Furthermore, all values of the array elements of the five array objects are initialized to be zero.

Variable `a` refers to the array object with five elements that are referring to the associated array objects. You can treat it as the first-level array object. Then, the expressions `a[0]`, `a[1]`, `a[2]`, `a[3]` and `a[4]` are referring to the array objects with ten elements for storing values of type `int` and they are considered to be second-level array objects. Therefore, the expression `a.length` refers to the attribute `length` of the array object with five elements, whereas `a[0].length` denotes the attribute `length` of the array object referred by the expression `a[0]`.

Figure 6.31 can also help you understand the interpretation of expressions that have the format `a[i][j]`. For example, the expression `a[0][0]` denotes the first element of the array referred by `a[0]`, and the expression `a[4][9]` specifies the last element of the array object referred by `a[4]`.

The statement

```
int[][]a = new int[5][10];
```

is actually a shorthand form of the following program segment:

```
int[][] a = new int[5][];
a[0] = new int[10];
a[1] = new int[10];
a[2] = new int[10];
a[3] = new int[10];
a[4] = new int[10];
```

The first statement of the above program segment first creates the first-level array objects for associating the other array objects that maintain values of type `int`. The subsequent five statements create the second-level array objects with element type `int` one by one, and their references are assigned to the suitable elements of the first-level array object. After executing the above program segment, the resulting scenario is the same as illustrated in Figure 6.31.

Besides using the `new` statement to create the array object explicitly, it is possible to initialize an array variable with an initializer (as mentioned in your second-to-last reading (pp. 542–45). For example, the statement

```
int[][] square = {{8, 3, 4}, {1, 5, 9}, {6, 7, 2}};
```

declares a two-dimensional array variable `square` and it is initialized with a first-level array object with three elements. Each refers to a second-level array object with three elements with values of type `int`. Check page 543 again if you don't remember how the initializer works.

Now, you have sufficient background knowledge of the use of two-dimensional arrays. The following reading gives some case studies that need to use two-dimensional arrays.

Reading

King, 'Using two-dimensional arrays as matrices', 'Using two-dimensional arrays to store images', 'Ragged arrays' and 'Using multidimensional arrays as parameters and results', pp. 546–48; 550

The reading provides an excellent example of the use of two-dimensional arrays — matrices. They are used extensively in engineering and scientific calculations. A matrix maintains numbers in rows and columns, and the numbers of rows and columns can be any positive number. For example, the following is a matrix with three rows and two columns.

$$B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix}$$

Such an arrangement of numbers is best maintained by a two-dimensional array such as:

```
double[][] b = new double[3][2];
```

Furthermore, it is possible to declare a two-dimensional array variable and initialize it with an initializer for a matrix such as:

```
double[][] b = {
  {1, 2},
  {3, 4},
  {5, 6}
};
```

In a mathematical sense, the row and column numbers start with `1` whereas the subscripts of a two-dimensional array start with `0`. Therefore, the subscripts are off by one. That is, the number denoted by b_{11} in the above matrix is stored in the array element `b[0][0]`. In general, the number denoted by b_{ij} is stored in the array element `b[i-1][j-1]`.

The reading discusses several program segments and methods for handling matrices that are implemented as two-dimensional arrays. Furthermore, it is possible to supply a method to the references of two-dimensional arrays so that the method can manipulate the number of the corresponding matrix, and the reference of a matrix as a two-dimensional array can be returned as the return value of a method. By consolidating the methods discussed in the reading and adding a few methods, a class

`MatrixHandler` is written in Figure 6.32 so that you can test the use of two-dimensional arrays.

```
// Definition of class MatrixHandler
public class MatrixHandler {
    // Show the contents of a matrix
    public void showMatrix(double[][] matrix) {
        for (int row=0; row < matrix.length; row++) {
            for (int col=0; col < matrix[row].length; col++) {
                System.out.print(matrix[row][col] + "\t");
            }
            System.out.println();
        }
    }

    // Find the sum of all elements of a matrix
    public double sumElements(double[][] matrix) {
        // Declare a cumulative total
        double total = 0.0;

        // Add the value in each row and column
        for (int row=0; row < matrix.length; row++) {
            for (int col=0; col < matrix[row].length; col++) {
                total += matrix[row][col];
            }
        }

        // Return the total
        return total;
    }

    // Create an identity matrix of the specified rank
    public double[][] createIdentityMatrix(int rank) {
        // Create a two-dimensional array representing a two-
        // dimensional array to be returned. All elements of
        // the two-dimensional array are initially zeroes.
        double[][] matrix = new double[rank][rank];

        // Set the diagonal elements to be 1.0
        for (int i=0; i < rank; i++) {
            matrix[i][i] = 1.0;
        }

        // Return the created identity matrix
        return matrix;
    }

    // Find the product matrix of the supplied matrices a and b
    public double[][] multiply(double[][] a, double[][] b) {
        // Create a two-dimensional array representing the
        // product matrices
        double[][] c = new double[a.length][b[0].length];

        // Create the value of each element in the product matrix
        for (int row = 0; row < a.length; row++) {
            for (int col = 0; col < b[0].length; col++) {
                c[row][col] = 0.0;
                for (int i=0; i < b.length; i++) {
                    c[row][col] += a[row][i] * b[i][col];
                }
            }
        }

        // Return the product matrix
        return c;
    }
}
```

Figure 6.32 MatrixHandler.java

The `showMatrix()` method is quite straightforward, so a nested `for` loop is used to show the elements of the two-dimensional arrays on the screen. The numbers in the same row are shown delimited with a tab character `'\t'` (by inserting a `String` object `"\t"`).

The `sumElements()` method finds the sum of all numbers of the matrix. The variable `total` is used as a cumulative total. By adding the elements of each column and each row using a nested `for` loop, the total of all elements is obtained and returned.

The `createIdentityMatrix()` method accepts a parameter of the rank of the required identity matrix. If the value of the supplied `rank` is two, the identity matrix contains two rows and two columns, and the values of the diagonal elements are `1.0`. In the last reading (p. 550), the `createIdentityMatrix()` method presented uses a nested `for` loop with an `if` statement to set the array element to `1.0` if the row and column of the array element are the same — that is, the diagonal elements. However, for the `createIdentityMatrix()` method implemented in the `MatrixHandler` class, a simple `for` loop suffices to set all the diagonal elements.

The `multiply()` method enables you to obtain the product matrix of two supplied matrices. First of all, it creates a necessary two-dimensional array object for storing the values of the product matrix. Then, a three-deep nested `for` loop is used to calculate each value of the product matrix as presented in the reading. Finally, the resultant two-dimensional array is returned as the required product matrix.

To test the `MatrixHandler` class, the driver program `TestMatrixHandler` is shown in Figure 6.33.

```
// Definition of class TestMatrixHandler
public class TestMatrixHandler {

   // Main executive method
   public static void main(String args[]) {
      // Declare and initialize two matrices
      double[][] a = {
         {1, 2, 3},
         {4, 5, 6}
      };
      double[][] b = {
         {1, 2},
         {3, 4},
         {5, 6}
      };

      // Create a MatrixHandler for manipulating matrices
      MatrixHandler handler = new MatrixHandler();

      // Show the contents of the two matrices
      System.out.println("The matrix A:");
      handler.showMatrix(a);
      System.out.println("\nThe matrix B:");
      handler.showMatrix(b);

      // Find and show the product matrix of matrices A and B
      double[][] c = handler.multiply(a, b);
      System.out.println(
         "\nThe product matrix of matrices A and B:");
      handler.showMatrix(c);

      // Find and show the sum of all elements of the product
      matrix
      System.out.println(
         "\nThe sum of all elements of the above product matrix=" +
         handler.sumElements(c));

      // Create and show an identity matrix of rank 4
      double[][] d = handler.createIdentityMatrix(4);
      System.out.println(
         "\nAn identity matrix of rank 4");
      handler.showMatrix(d);
   }
}
```

Figure 6.33 TestMatrixHandler.java

Compile the classes `MatrixHandler` and `TestMatrixHandler`, and execute the `TestMatrixHandler` program. The following output is shown on the screen:

```
The matrix A:
1.0     2.0     3.0
4.0     5.0     6.0
```

```
The matrix B:
1.0     2.0
3.0     4.0
5.0     6.0

The product matrix of matrices A and B:
22.0    28.0
49.0    64.0

The sum of all elements of the above product
matrix=163.0

An identity matrix of rank 4
1.0     0.0     0.0     0.0
0.0     1.0     0.0     0.0
0.0     0.0     1.0     0.0
0.0     0.0     0.0     1.0
```

Please use the following self-test to experiment with using a two-dimensional array for storing a multiplication table.

Self-test 6.8

Write the definition of a class `TableMaker` that can create a multiplication table and can display it on the screen. The design of the class `TableMaker` is:

TableMaker
table : int[][]

createTable(row : int, column : int)
printTable()

To create a multiplication table, the method `createTable()` is called with numbers of rows and columns required to create a two-dimensional array for storing the multiplication table and is referred by the attribute `table`. Then, the method `printTable()` is called to show the table on the screen. For example, the definition of class `TestTableMaker`

```
// Definition of class TestTableMaker to test a TableMaker
// object
public class TestTableMaker {

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

        // Create the table with 4 rows and 5 columns
        maker.createTable(4, 5);
        // Show the table on the screen
        maker.printTable();
    }
}
```

shows the following output on the screen:

```
1        2        3        4        5
2        4        6        8        10
3        6        9        12       15
4        8        12       16       20
```

Ragged arrays

The Java programming language implements two-dimensional arrays as two levels of one-dimensional arrays. If you imagine the elements of a two-dimensional array as a table, the first-level array determines the number of rows, and the associated second-level arrays determine the number of columns in the rows. As the second-level array objects are mutually independent, it is possible for these second-level array objects to have different sizes. Such a two-dimensional array is known as a non-rectangular array or ragged array.

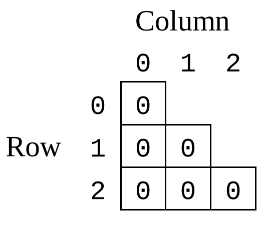
The shorthand you learned in the previous section, such as

```
int[][] a = new int[5][10];
```

creates rectangular two-dimensional arrays only. To create a ragged array, it is usually necessary to create the two-level array objects that constitute a two-dimensional array with a few statements. For example, the following program segment creates a ragged two-dimensional array:

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

The above program segment creates a two-dimensional array that can be visualized as the following non-rectangular table:



Internally, such a ragged array is stored as shown in Figure 6.34:

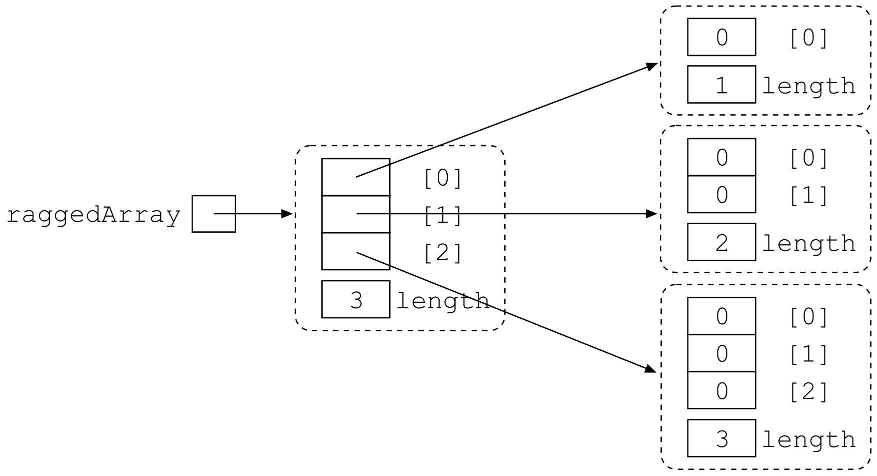


Figure 6.34 The storage of a ragged array in JVM

Another possible way to create a ragged two-dimensional array is to use an array initializer, such as:

```
int[][] raggedArray = {
  {0},
  {0, 0},
  {0, 0, 0}
};
```

Recursion — an advanced control structure

The problems you have encountered so far can be solved by using software objects, and the object behaviours have been implemented using *branching* or *looping* structures. However, instead of simply using these two structures, some problems can preferably be solved using another approach. For example, if you have built a software application and each staff member in the company is modelled by a `Staff` object in the software, how can you determine whether a staff member, say Staff A, is directly or indirectly under the supervision of another staff member, say Staff B?

For example, Figure 6.35 shows a sample organization chart of a company.

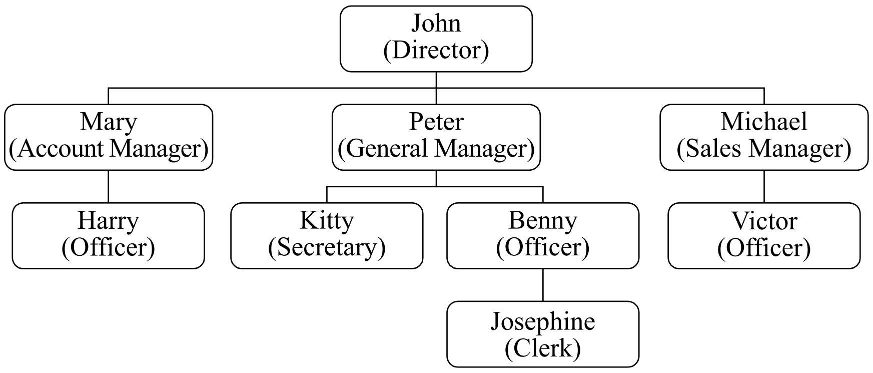


Figure 6.35 An organizational structure of a company

Figure 6.35 illustrates a typical organizational chart of a company. It is obvious from the chart that all staff members are under the supervision of the director, John. However, is the clerk Josephine under the supervision of the sales manager, Michael? We can easily figure out that the answer is ‘no’, but how can we solve such a problem in a program?

You might wonder whether it is possible for every `Staff` object to have an array object that maintains the references of all other `Staff` objects that are under his or her supervision. It is feasible, but it is not a preferable solution, because maintaining the array objects of all `Staff` objects is tedious and error-prone.

In reality, what we can tell about a staff member is his or her name and his or her direct boss or immediate supervisor. With such relationships among the staff members, it is possible to solve the aforementioned problem in the following way:

The answer to the question, ‘is Staff A under the supervision of Staff B’ is

- no, if Staff A has no direct boss, or
- yes, if Staff B is the direct boss of Staff A, or
- yes, if the direct boss of Staff A is under the supervision of Staff B.

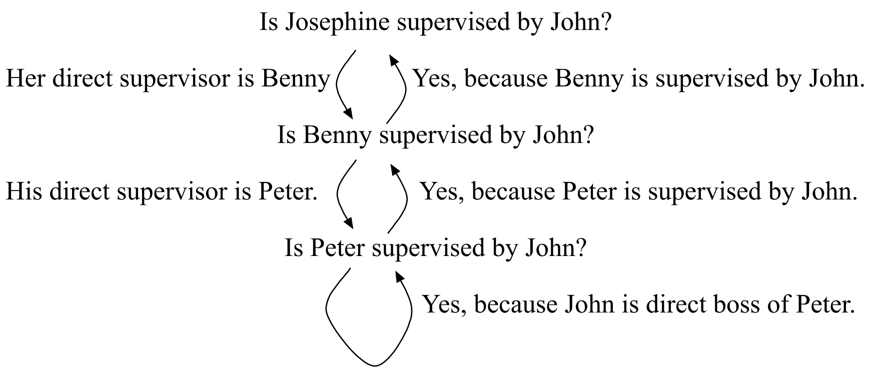
For example, with respect to organization structure shown in Figure 6.35, how can we answer the question ‘is Josephine supervised by John’?

To answer the question with the above definition, we know that John is not Josephine’s *direct* boss. John supervises Josephine only if John supervises Benny, who is Josephine’s direct boss. Then, the problem is resolved to be whether John supervises Benny. If Benny is under John’s supervision, it implies that Josephine is also supervised by John.

Subsequently, to answer the question of whether Benny is under John’s supervision, it is necessary to determine whether Peter, Benny’s direct boss, is under John’s supervision, as John is not Benny’s direct boss. That is, another problem of whether Peter is under John’s supervision is created.

The answer to the question of whether Peter is supervised by John is ‘yes’, because John is Peter’s direct boss, and Peter is under John’s supervision. Then, it implies that Benny is supervised by John, as Peter is supervised by John. It also implies that Josephine is supervised by John, because Benny is supervised by John.

Visually, the sequence in determining the answer to the question ‘is Josephine supervised by John’ is:


Is Josephine supervised by John?
Her direct supervisor is Benny
Yes, because Benny is supervised by John.
Is Benny supervised by John?
His direct supervisor is Peter.
Yes, because Peter is supervised by John.
Is Peter supervised by John?
Yes, because John is direct boss of Peter.

The above approach of problem solving is different from what we have seen in *MT201* so far. In every step of the determination, a *similar* problem is created that must be solved until a concrete answer can be obtained. Such an approach is known as *recursion*. We’ll discuss this interesting technique now.

What is recursion?

We have just discussed the problem of determining whether a staff member is under the supervision of another staff member. A recursive approach was used to determine the answer.

Recursion is a problem-solving technique that, if given a problem, is resolved to another identical problem but with a smaller size. Such an approach is repeated until a well-defined answer is obtained. Afterwards, the solutions to the smaller problems can be used to synthesize the solutions to the larger problems and ultimately the original problem. For example, in our previous example, to determine whether a staff member, say Staff A, is under the supervision of another staff member, say Staff B, the problem is resolved through the determination of whether the direct boss of Staff A is under the supervision of Staff B. Such determinations are repeated until the ultimate boss is reached. In every step of resolving a problem, a problem of the same type is created that needs to be solved. In our example, the same problem is created for the determination of another staff member of a higher grade who is closer to the ultimate boss of the organization. The problem is finally resolved to a well-defined problem for which the answer is defined. Such a well-defined problem or scenario is known as the base case. Then, the solution to the base case is used to synthesize the solution of a larger problem that is closer to the original problem. Such a process repeats until the solution to the original problem is eventually obtained.

Recursion is an amazing way to handle a problem. Instead of detailing every step to solve the problem, a recursive method defines base cases, which can give solutions right away. All cases other than the base case are resolved to the same problem but with a smaller size that is closer to the base case. The process repeats until the base case is reached. The solution is passed back from the simpler cases to the original case, and hence the solution to the first problem is reached.

The previous problem of defining whether a staff member is under the supervision of another staff member is an everyday example that can be solved in a recursive way. We will now discuss how to deal with some mathematical problems using such a technique.

A simple recursion example — calculating the factorial

We came across the calculation of factorial in *Unit 4*. The calculation is straightforward. A simple loop can be used to find the result, like the `factorial()` method of the class `FactorialCalculator` discussed in *Unit 4*.

You know that the definition of the factorial of a non-negative integer n is

$$n! = 1 \times 2 \times 3 \times \ldots\ldots \times (n\text{-}1) \times n$$

The above formula can be rewritten as:

$$n! = \begin{cases} 1, \text{ if } n = 0 \\ n \times (n-1)!, \text{ otherwise} \end{cases}$$

By definition, the factorial of zero is one. According to the above definition, to determine the factorial of n, if the value of n is greater than 0, it is necessary to find the factorial of n-1 and the factorial of n is the product of n and the factorial of n-1.

Based on the above definition, it is possible to write the method `factorial()` in a recursive way in the definition of class `FactorialCalculator` as shown in Figure 6.36.

```
// The class definition of FactorialCalculator
public class FactorialCalculator {

    // The method for calculating the factorial of a given
    // number via the parameter
    public int factorial(int number) {
        if (number == 0) {
            // The base case for number equals 0
            return 1;
        }
        else {
            // Otherwise, find the factorial recursively
            return number * factorial(number - 1);
        }
    }
}
```

Figure 6.36 FactorialCalculator.java

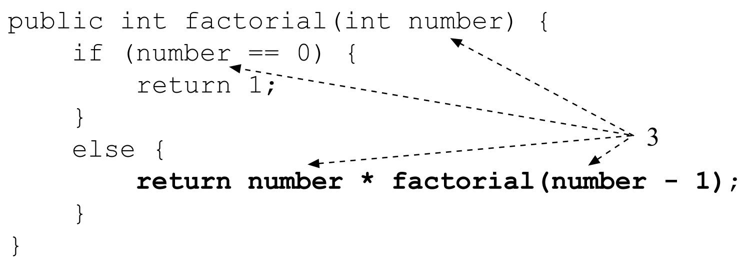
To calculate the factorial of `3`, the following driver program `TestFactorialCalculator` is written as Figure 6.37.

```
// The class definition of FactorialTester for setting up the
// environment and testing the FactorialCalculator object
public class TestFactorialCalculator {

    // The main executive method
    public static void main(String args[]) {

        // Create a FactorialCalculator object and use variable
        // calculator to refer to it
        FactorialCalculator calculator = new FactorialCalculator();

        // Calculate the factorial of 3
        System.out.println(calculator.factorial(3));
    }
}
```

Figure 6.37 TestFactorialCalculator.java

There are only two statements in the `main()` method of the `TestFactorialCalculator` class. It first creates an object of the class `FactorialCalculator` and it then calls its `factorial()` method with parameter `3`. The return value of calling the `factorial()` method will be shown on the screen.

First, a message `factorial` with supplementary data `3` is sent to the `FactorialCalculator` object. The `FactorialCalculator` object then executes its `factorial()` method. According to the method, the value of the parameter, `number`, is compared with zero.

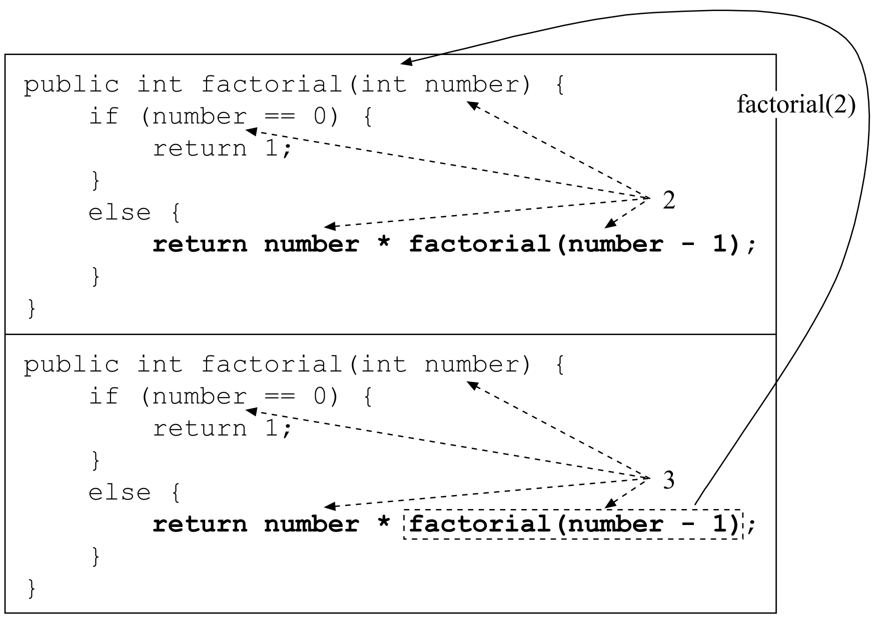


```
public int factorial(int number) {
    if (number == 0) {
        return 1;
    }
    else {
        return number * factorial(number - 1);
    }
}
```

(In the above program segment, the comments are removed for a shorter listing, and the dashed arrows indicate the value of the variable.)

As the value of the parameter `number` is `3`, the condition is `false` and the `else` part of the `if/else` statement is executed. Therefore, the value of the expression `factorial (number-1)`, `number–1` (which is `2`), is determined and is sent with the message `factorial` to the `FactorialCalculator` object itself.

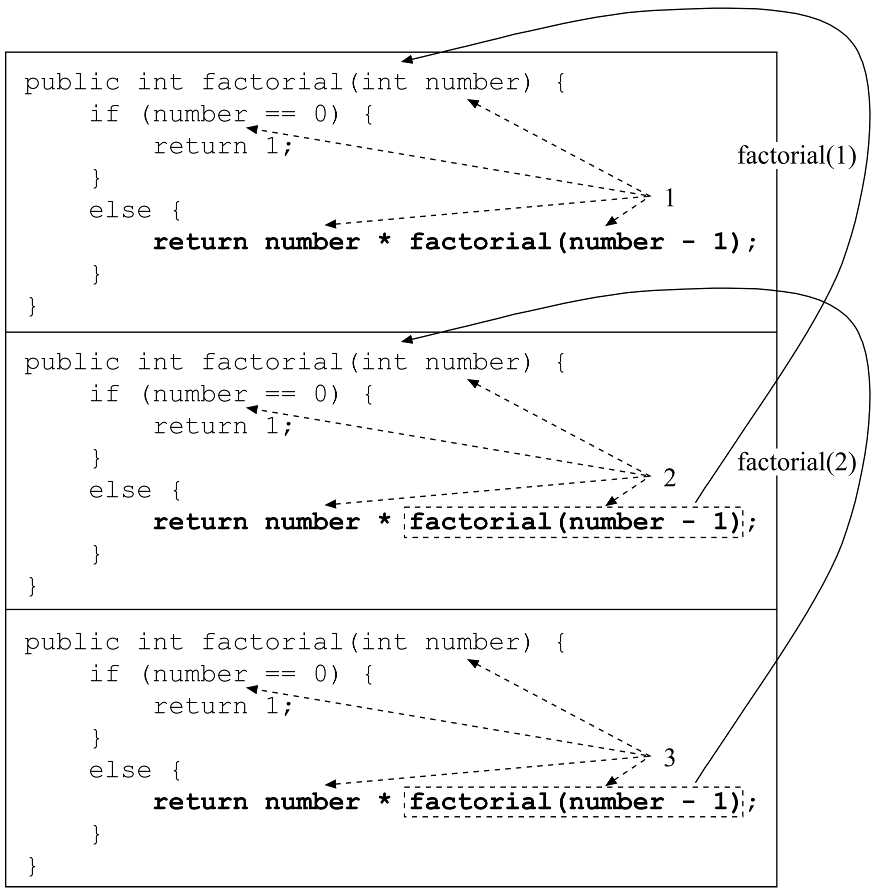
When the method `factorial()` is executed for the second time, its parameter takes the value `2`. Please notice that every time a method is called, new storage locations in the memory are allocated for the parameters and the local variables. As a result, the parameters and local variables can take different values for an individual method call. The scenario becomes:



```
public int factorial(int number) {
    if (number == 0) {
        return 1;
    }
    else {
        return number * factorial(number - 1);
    }
}
```

```
public int factorial(int number) {
    if (number == 0) {
        return 1;
    }
    else {
        return number * factorial(number - 1);
    }
}
```

The parameter `number`, with value of `2`, is compared with zero. The condition is `false`, and the `else` part of the `if/else` statement is executed. Therefore, the expression of `number–1` is evaluated and gives `1` and the method `factorial()` of the object is executed again with parameter `1`.

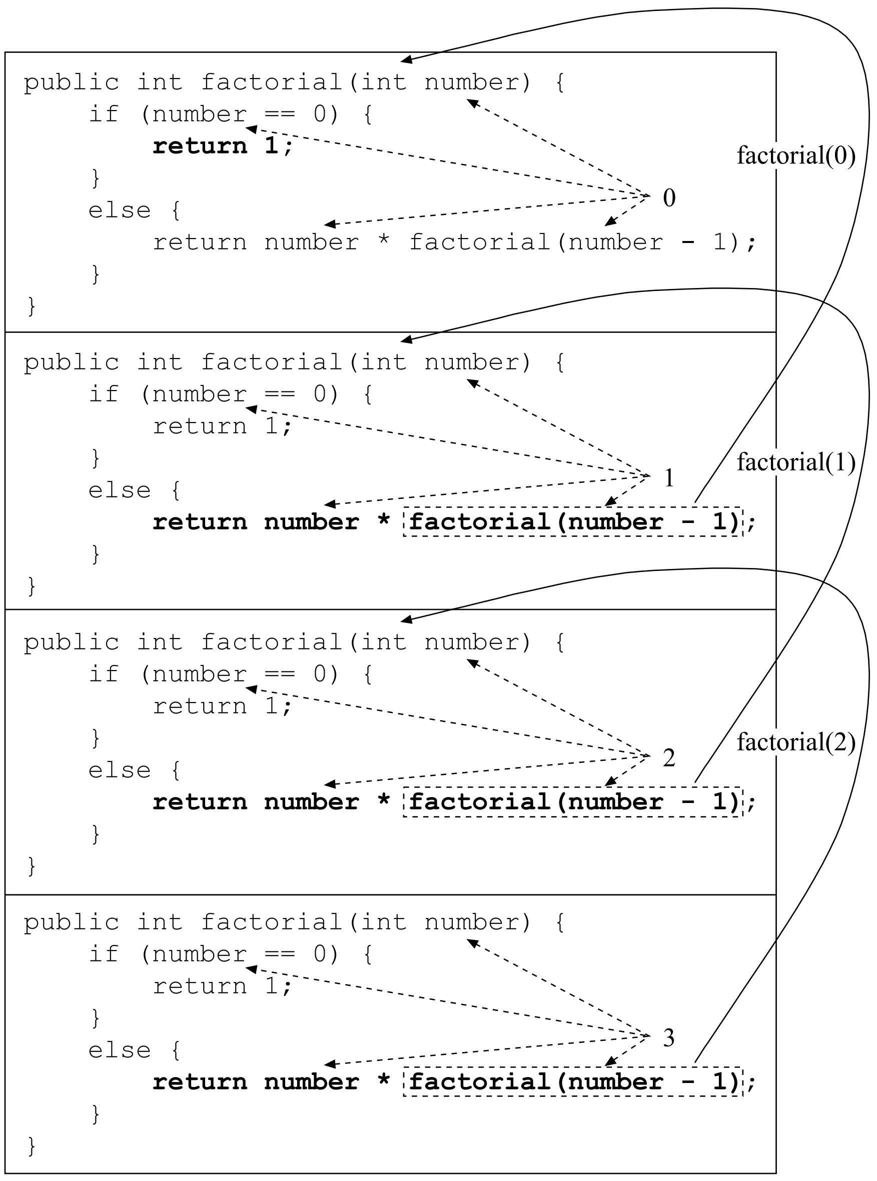


```
public int factorial(int number) {
    if (number == 0) {
        return 1;
    }
    else {
        return number * factorial(number - 1);
    }
}
```

```
public int factorial(int number) {
    if (number == 0) {
        return 1;
    }
    else {
        return number * factorial(number - 1);
    }
}
```

```
public int factorial(int number) {
    if (number == 0) {
        return 1;
    }
    else {
        return number * factorial(number - 1);
    }
}
```

Once again, the method `factorial()` is executed and the value of the parameter `number` is 1. Therefore, the condition is still `false`, and the `else` part of the `if/else` statement is executed. The expression `number-1` is evaluated that gives `0` and the method `factorial()` is called with parameter `0`.



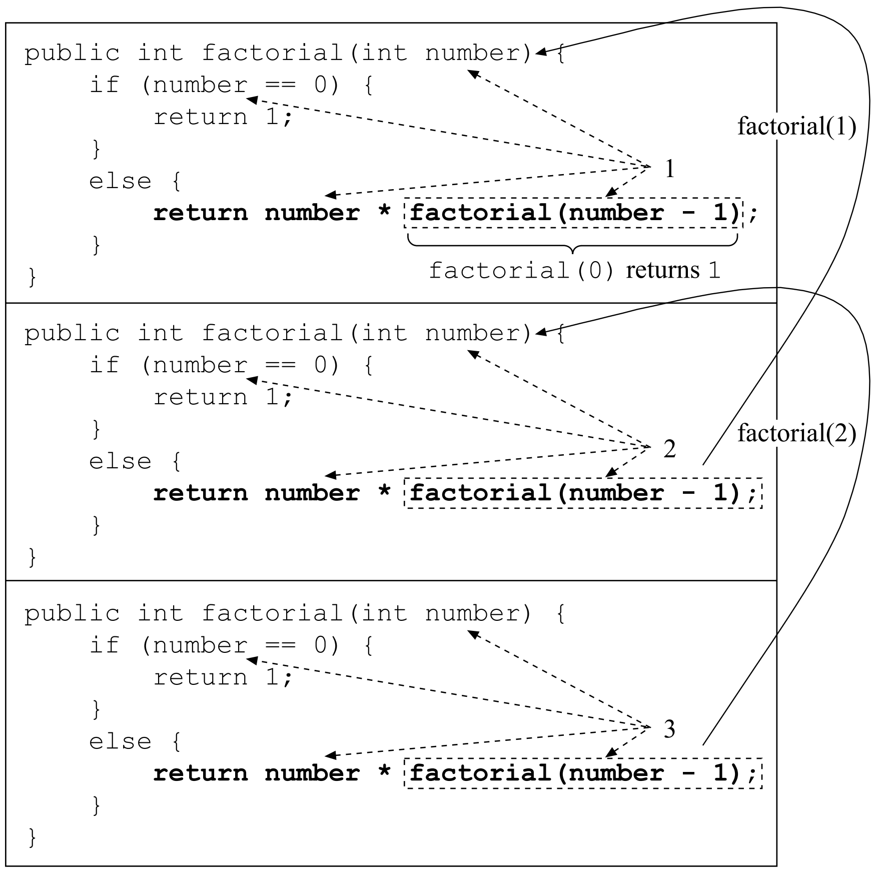
```
public int factorial(int number) {
    if (number == 0) {
        return 1;
    }
    else {
        return number * factorial(number - 1);
    }
}
```

```
public int factorial(int number) {
    if (number == 0) {
        return 1;
    }
    else {
        return number * factorial(number - 1);
    }
}
```

```
public int factorial(int number) {
    if (number == 0) {
        return 1;
    }
    else {
        return number * factorial(number - 1);
    }
}
```

```
public int factorial(int number) {
    if (number == 0) {
        return 1;
    }
    else {
        return number * factorial(number - 1);
    }
}
```

Now, the value of the parameter number is `0` and the condition is `true`. The `if` part of the `if/else` statement is executed which returns a value of `1` and the value is returned to the caller method; that is:

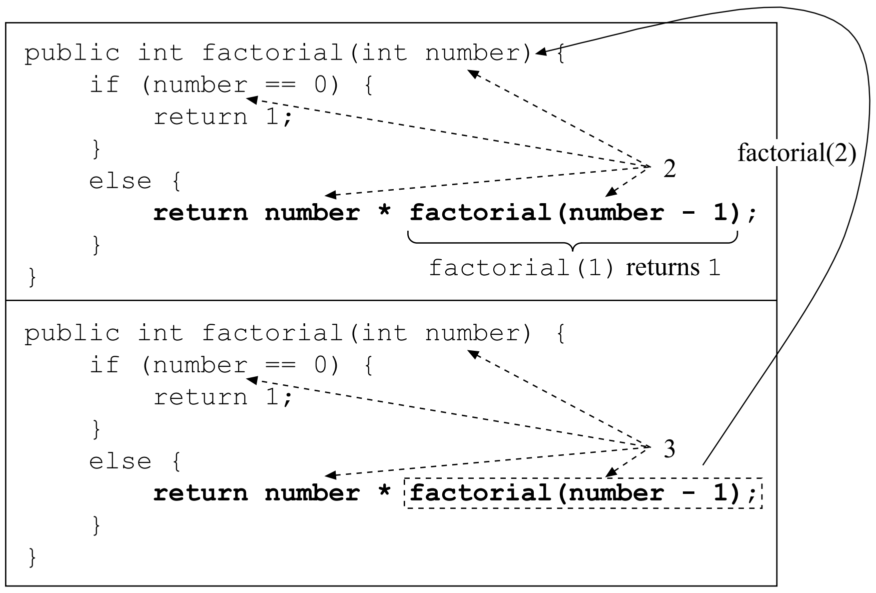


```
public int factorial(int number) {
    if (number == 0) {
        return 1;
    }
    else {
        return number * factorial(number - 1);
    }
}
```

```
public int factorial(int number) {
    if (number == 0) {
        return 1;
    }
    else {
        return number * factorial(number - 1);
    }
}
```

```
public int factorial(int number) {
    if (number == 0) {
        return 1;
    }
    else {
        return number * factorial(number - 1);
    }
}
```

In the method `factorial()`, the expression in the `else` part continues execution and the product of `number` and `factorial(number-1)` is evaluated. As the value of the parameter `number` is `1` and the result of `factorial(number-1)` — that is `factorial(0)` — is `1`, the result of the expression is `1`. This value is returned to the caller method; that is:

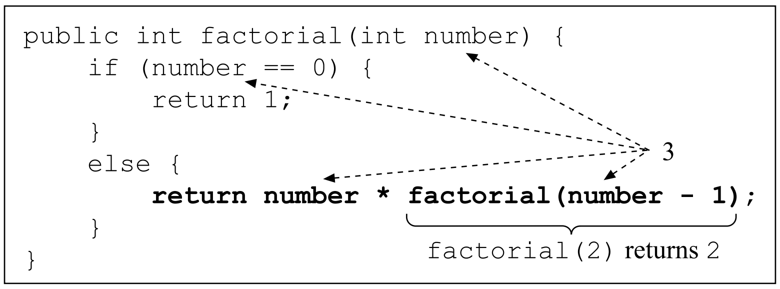


```
public int factorial(int number) {
    if (number == 0) {
        return 1;
    }
    else {
        return number * factorial(number - 1);
    }
}
```

```
public int factorial(int number) {
    if (number == 0) {
        return 1;
    }
    else {
        return number * factorial(number - 1);
    }
}
```

The value of parameter `number` is `2` and the result of `factorial(number –1)`, `factorial(1)`, is `1`. Therefore, the result of `number * factorial(number –1)` is `2` times `1`, which is `2`. Therefore, the result of `factorial(2)` equals `2`.

The flow of control returns to the caller method as well, and the scenario becomes:



```
public int factorial(int number) {
    if (number == 0) {
        return 1;
    }
    else {
        return number * factorial(number - 1);
    }
}
```

Finally, for the execution of `factorial()` method with parameter `3`, the method call `factorial(number – 1)`, which is `factorial(2)`, returns a value of `2`. Then, the expression `number * factorial(number – 1)` that is `3` times `2` which is `6`. As a result, the return value of the method call `factorial(3)` in the `main()` method of `TestFactorialCalculator` is `6` and the result is displayed on the screen.

In the previous few pages, we discussed the detailed sequence of executions during a method call that is recursive. Please use the following activity to reinforce your understanding of the operational sequence of the above `factorial()` method.

Activity 6.3

Modify the `factorial()` method of the `FactorialCalculator` class by adding statements that show the value of the parameter `number`, the `return` value from a recursive method call and the value to be returned. Please verify the output messages and the above descriptions of the operation sequence and the values during the execution.

Another simple recursion example — the Euclidean algorithm

A Greek mathematician and geometer Euclid (circa 325–circa 270 BCE) wrote an important textbook on geometry (called *Elements*) that was used for centuries in Western Europe. In one of the chapters, he discussed an algorithm for finding the greatest common divisor (GCD) of two numbers. For example, the GCD of 18 and 24 is 6, and the GCD of 5 and 12 is 1. This algorithm is known as the Euclidean algorithm.

The idea of the algorithm is that given two numbers *a* and *b*, the greatest common divisor of the two numbers, gcd(*a*, *b*) is determined to be

$$\gcd(a,b) = \begin{cases} a, \text{ if } b = 0 \\ \gcd(b,\ a \,\%\, b), \text{ otherwise} \end{cases}$$

where *a* % *b* is the remainder of the division of *a* by *b*. You can see that the format of the algorithm is similar to the one for the calculation of the factorial. There is a base case in which the value of the number *b* is zero. Otherwise, the greatest common divisor is determined to be that of the values *b* and the remainder of the division of *a* by *b*. For example, for the numbers `39304` and `5100`, the steps to determine the GCD divisor are:

		Remarks
gcd(39304, 5100)	= gcd(5100, 3604)	39304 % 5100 = 3604
	= gcd(3604, 1496)	5100 % 3604 = 1496
	= gcd(1496, 612)	3604 % 1496 = 612
	= gcd(612, 272)	1496 % 612 = 272
	= gcd(272, 68)	612 % 272 = 68
	= gcd(68, 0)	272 % 68 = 0
	= 68	

Therefore, the greatest common divisor of the two numbers `39304` and `5100` is `68`.

The elegance of writing the recursive method is that you can write the method in the way the formula is defined. For example, for the Euclidean algorithm, the method `gcd()` can be written as:

```
public int gcd(int a, int b) {
    if (b == 0) {
        return a;
    }
    else {
        return gcd(b, a % b);
    }
}
```

You can see that the method definition is pretty much the same as the mathematical formula. The complete definition of the class `GCDCalculator` that has the `gcd()` method is shown in Figure 6.38.

```
// Definition of class GCDCalculator
public class GCDCalculator {

    // Calculate the GCD of the numbers a & b
    public int gcd(int a, int b) {
        // Check if the value of b equals 0
        if (b == 0) {
            // If so, a is the GCD
            return a;
        }
        else {
            // Otherwise, determine the GCD recursively
            return gcd(b, a % b);
        }
    }
}
```

Figure 6.38 GCDCalculator.java

To determine the `gcd()` method of the class `GCDCalculator`, the driver program in Figure 6.39 is written to accept two numbers from the program parameters.

```
// Definition of class FindGCD
public class FindGCD {

    // Main executive method
    public static void main(String args[]) {
        // Check whether sufficient program parameters are provided
        if (args.length < 2) {
            System.out.println("Usage: java TestGCD number1 number2");
        }
        else {
            // Get the numbers from program parameters
            int n1 = Integer.parseInt(args[0]);
            int n2 = Integer.parseInt(args[1]);

            // Create the GCDCalculator object and have it referred
            // by variable calculator
            GCDCalculator calculator = new GCDCalculator();

            // Calculate the GCD and show the result to the screen
            System.out.println("The GCD of " + n1 + " and " + n2 +
                " is " + calculator.gcd(n1, n2));
        }
    }
}
```

Figure 6.39 FindGCD.java

Compile the above two class definitions and execute the `FindGCD` with two numbers as the program parameters, such as:

```
java FindGCD 72 54
```

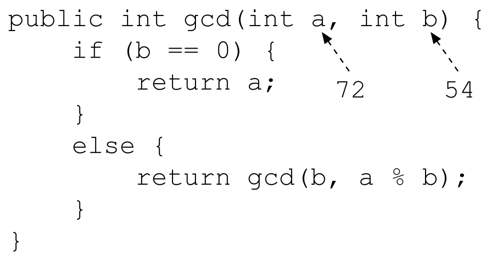
The program displays the following message:

```
The GCD of 72 and 54 is 18
```

Now, let's trace the operations of the `gcd()` method of the class `GCDCalculator` with parameters `72` and `54`.

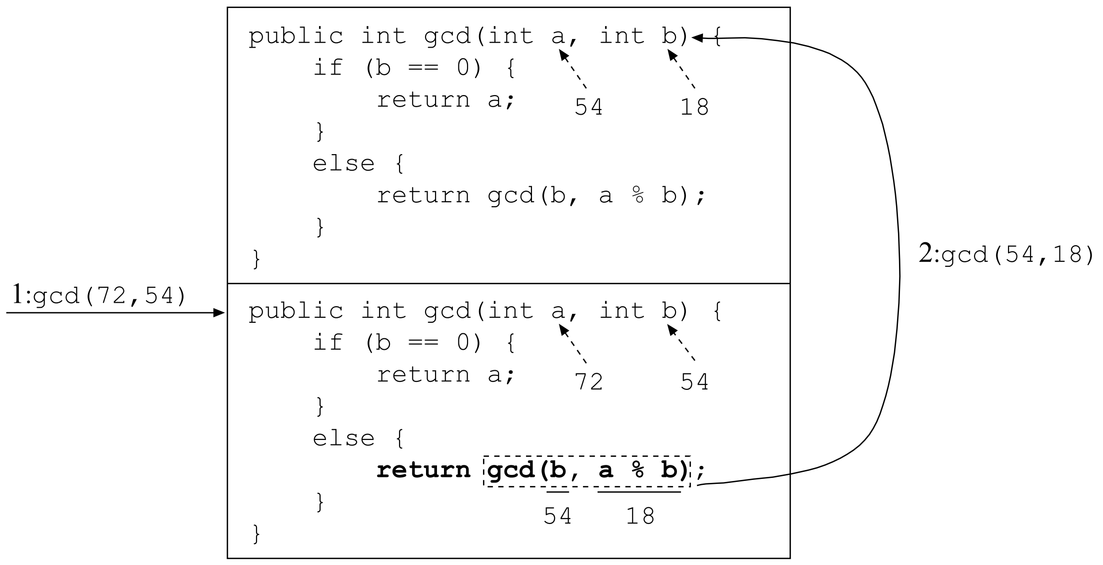
If the `FindGCD` program is executed with program parameters `72` and `54`, the `if/else` statement ensures that the necessary parameters are provided. Then, an object of the class `GCDCalculator` is created that is referred by the variable `calculator`. Then, the numbers are derived from the program parameters and stored in local variable `n1` and `n2`. The two numbers are finally used to call the `gcd()` method of the `GCDCalculator` object.

First, the `gcd()` method is called with parameters `72` and `54`. That is:



```
public int gcd(int a, int b) {
    if (b == 0) {
        return a;     72      54
    }
    else {
        return gcd(b, a % b);
    }
}
```

As the value of variable `b` is not zero, the `else` part of the `if` statement is executed, which calculates the expression `a % b` that gives `18`, and the `gcd()` method is called again with parameters `54` and `18`.

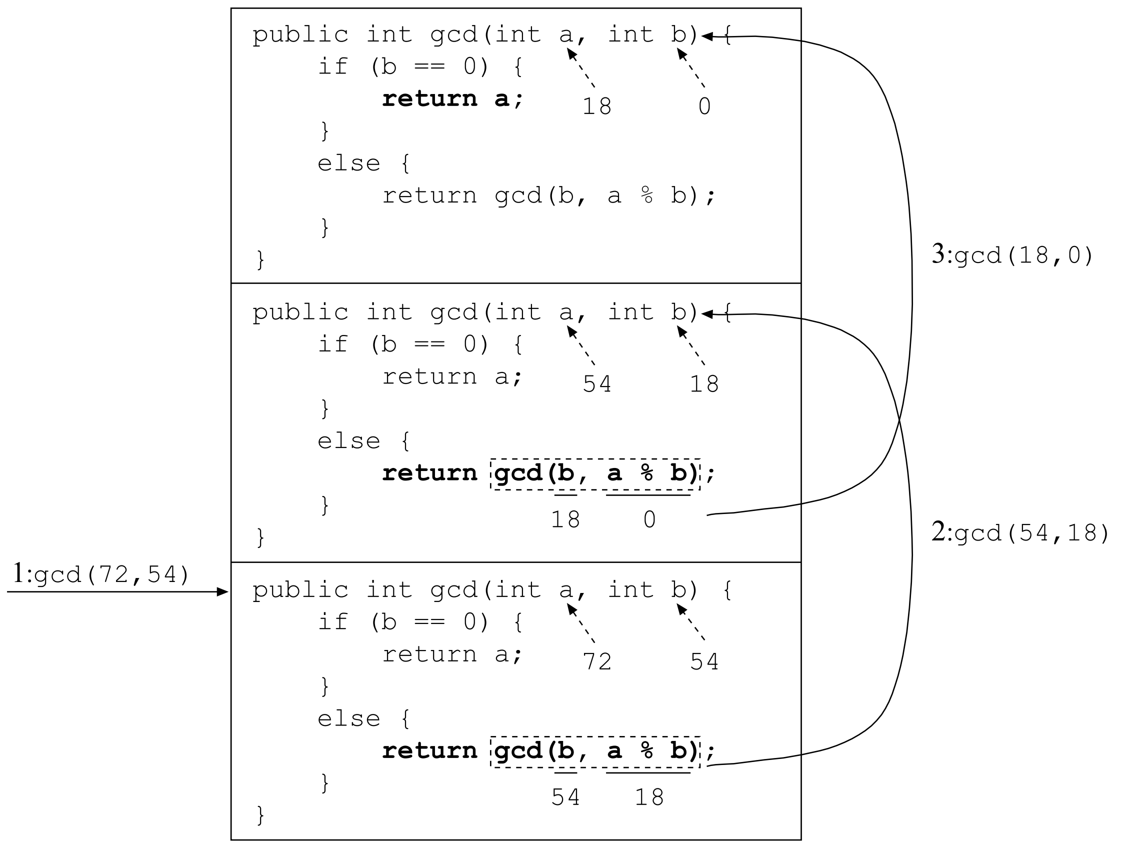


```
public int gcd(int a, int b) {
    if (b == 0) {
        return a;     54      18
    }
    else {
        return gcd(b, a % b);
    }
}
```

```
public int gcd(int a, int b) {
    if (b == 0) {
        return a;     72      54
    }
    else {
        return gcd(b, a % b);
    }
}
```

(In the above and in subsequent diagrams, the dashed arrows with the numbers specify the values of the variables. The solid arrows and the numbers with colons indicate the sequence of method calls and the return values.)

For the second call to the `gcd()` method, the values of the parameters are `54` and `18`. The condition is evaluated to be `false` and the `else` part of the `if/else` statement is executed. Then, the expression `54 % 18` is calculated and gives `0`. The method `gcd()` is called for the third time with parameters `18` and `0`.

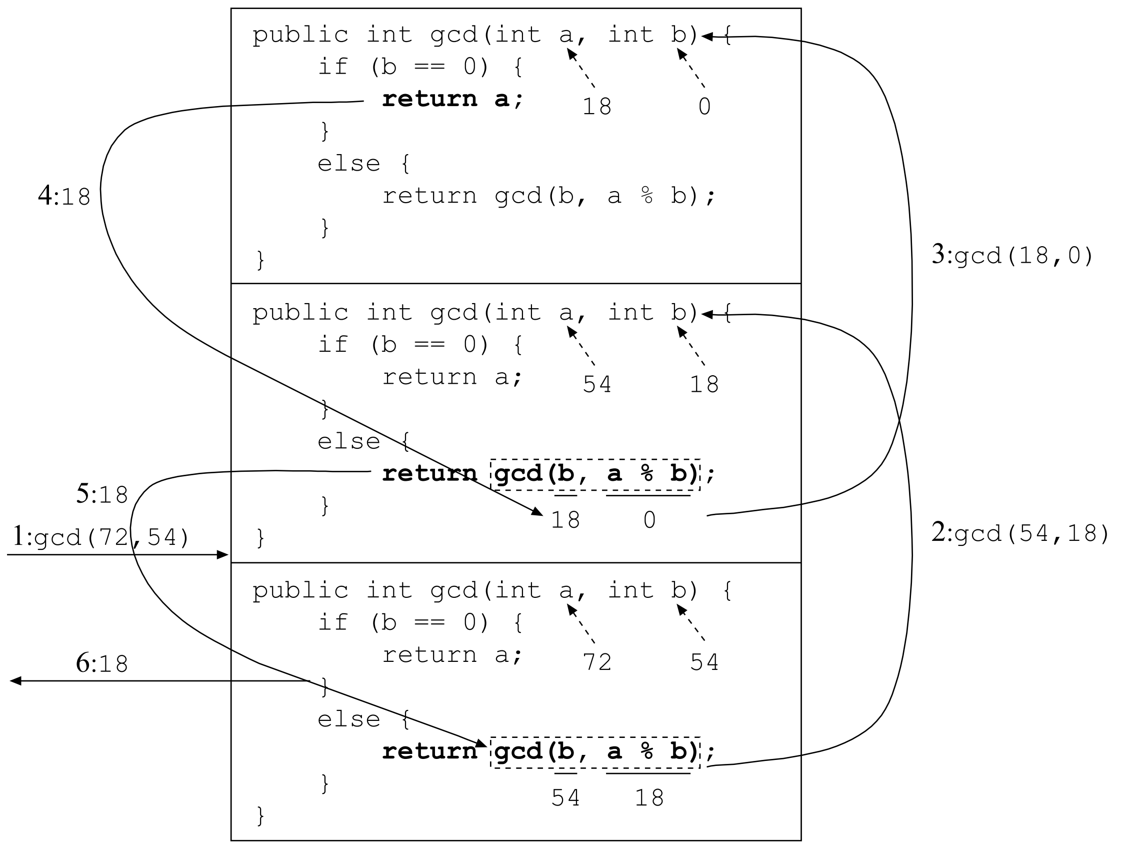


```
public int gcd(int a, int b) {
    if (b == 0) {
        return a;
    }
    else {
        return gcd(b, a % b);
    }
}
```

```
public int gcd(int a, int b) {
    if (b == 0) {
        return a;
    }
    else {
        return gcd(b, a % b);
    }
}
```

```
public int gcd(int a, int b) {
    if (b == 0) {
        return a;
    }
    else {
        return gcd(b, a % b);
    }
}
```

Therefore, the value `18` is returned from the third call to the second call and hence from the second call to the first call. Finally, the call of `gcd()` method with parameters `72` and `54` returns the value `18`. The statement in the `main()` method then shows the result `18`.



```
public int gcd(int a, int b) {
    if (b == 0) {
        return a;
    }
    else {
        return gcd(b, a % b);
    }
}
```

```
public int gcd(int a, int b) {
    if (b == 0) {
        return a;
    }
    else {
        return gcd(b, a % b);
    }
}
```

```
public int gcd(int a, int b) {
    if (b == 0) {
        return a;
    }
    else {
        return gcd(b, a % b);
    }
}
```

Recursion is a powerful programming technique, but it might appear strange to you at first sight. This unit discusses three examples — the determination of an organizational hierarchy, the calculation of a factorial, and the Euclidean algorithm. Please review the descriptions if you cannot trace the operation sequences in the examples.

In conclusion, the following points are provided for you to recap your understanding of recursion.

1 The nature of a recursive algorithm is that a problem can be resolved to a problem of the same type but the problem size is smaller. Such a process repeats until the problem is resolved to be a trivial or base case.

2 From the programming point of view, a characteristic of a recursive method is that it calls itself directly or indirectly.

Therefore, if you want to solve a problem in a recursive way, you should first determine whether the problem can be resolved into the same problem(s) but with a smaller size. Furthermore, it is impossible to create a problem of the same type with a smaller size infinitely. The problem must eventually be resolved to a trivial or base case so that no new problem of the same type is created and the solution is used to obtain the same problem with a larger size until the solution of the original problem is obtained. In other words, the general format of a method that implements recursion is:

```
public ... solveProblem(... problemParameter) {
  if (problemParameter == baseCaseParameter) {
    // return the solution for base case
    return solutionForBaseCase;
  }
  else {
    // Finding the solution recursively
    // Derive the parameters of the problem with a smaller size
    newProblemParameter = ...;
    // Call the method itself for getting the solution
    // of the problem with a smaller size
    solutionForProblemWithSmallerSize =
        solveProblem(newProblemParameter);
    // Construct the solution of the problem of the current size
    // according to the solution obtained for the problem with a
    // smaller size
    solutionForCurrentProblem = ...;
    // Return the solution for the current problem
    return solutionForCurrentProblem;
  }
}
```

For example, the method for finding the factorial with recursion is:

```
public int factorial(int number) {
  if (number == 0) {
    // The base case of factorial is 0! = 1
    return 1;
  }
  else {
    // Get the parameter of the same problem with a smaller size
    number1 = number – 1;
    // Get the solution of the same problem with a smaller size
    result1 = factorial(number1);
    // Construct the solution of the current problem
    result = number * result1;
    // Return the obtained solution
    return result;
  }
}
```

The above method definition is equivalent to the one defined in the `FactorialCalculator` class, but the details of each step are clearly illustrated, whereas the one defined in the `FactorialCalculator` class is a simplified version.

A difficulty you might encounter when you are tracing the execution sequence of a method, especially for a recursive method, is that you are uncertain about the values of the parameters and the local variables. You should bear in mind that every time a method is called, new storage in the memory is allocated for the parameters and local variables so that they can take different values in individual method calls.

Please use the following self-test to verify your understanding of recursion and experience writing recursive methods.

Self-test 6.9

This question is to design a program to calculate the following expression

$$1 + 2 + 3 + ... + n$$

where n is a positive integer.

1 If the expression, $1 + 2 + 3 + ... + n$, is denoted by sum(n), what is the relationship between sum(n) and sum(n-1)?

2 What is the base case of the function sum(n)? That is, what is the value of n so that the expression, $1 + 2 + 3 + ... + n$, is determined without any further evaluation of sum(n-1)?

3 Write a `Summation` class with a `sum()` method that finds the result of the expression, $1+2+3+...+n$ using a recursive approach. Then, write a driver program `TestSummation` that accepts a program parameter as the value of *n* and the value of the expression, $1+2+3+...+n$, is determined by the `sum()` method of the `Summation` class.

This is the end of the discussion of recursion. You might wonder what other problems can be solved using such an approach. As you gain more experience and encounter more problems to be solved by programming, you will find that the algorithms of some problems can be presented in recursive ways and hence recursive methods can be written to solve those problems.

Summary

This has been a rather lengthy unit that discusses many advanced programming features in the Java programming language.

The first part of the unit discusses how you can construct complex logical expressions, which involves multiple sub-expressions that are connected by some logical operators such as the && operator and & operator for logical And operation, and the `||` operator and `|` operator for logical Or operation. Among these four operators, the && operator and `||` operator are known as short-circuit operators in that the second operand is not evaluated if the entire results are already determined. Two other logical operators are discussed: the `^` operator and `!` operator for logical Exclusive Or and Not operations.

Then, we discussed nested branching structures and nested looping structures. The `break` and `continue` statements were introduced to terminate a looping structure immediately or to skip the statements in the current iteration respectively. For nested structures, it is possible to use a label to specify the outer branching or looping structure that the `break` and `continue` statements will apply to.

We then discussed another important operation — sorting — and a simple sorting algorithm, insertion sort. You learned how to apply this sorting method to numeric arrays and `String` arrays.

You learned the use of one-dimensional arrays in *Unit 5*. This unit elaborated on them to implement multidimensional arrays in the Java programming language. For example, with a two-dimensional array, it is possible to consolidate a table of data by using two subscripts to specify a data location (and therefore specify a data item).

Finally, we discussed an interesting and challenging programming technique — recursion. Three examples were used to illustrate its concepts and applications. The core idea is that a problem to be solved by recursion is resolved to a similar and simpler problem to be solved. Such a process repeats until it reaches the base case and the solutions to the original problem are constructed accordingly.

Suggested answers to self-test questions

Self-test 6.1

There are many equivalent ways to construct the conditions. The following is a list of possible ones.

1 `a <= c && c <= b`

2 `a * b >= 0`, or
`a >= 0 ^ b >= 0`

3 `a == b && b == c && c == d`

Self-test 6.2

1 For the condition, `a++ >= 0 || b++ >= 0`, the first sub-condition is evaluated first and gives `true` and the value of variable `a` is increased by 1. As the double pipe operator is a short-circuit operator, the second sub-condition is not evaluated, which means that `b++` is not executed. The result of the entire condition is determined to be `true` because of the short-circuit nature of the double pipe operator, and the `if` part of the `if/else` statement is executed, which is, `c++`. Therefore, after the program segment is executed, statements `a++` and `c++` are executed. The values of variables `a`, `b` and `c` are 1, 0 and 1 respectively.

2 The first sub-condition `a++ >= 0` is executed and gives `true` and the value of variable `a` is increased by one. It is necessary to evaluate the second sub-condition to determine the result of the entire condition. Therefore, the sub-condition `b++ >= 0` is also evaluated and that gives `true` and the variable `b` is increased by one. As a result, the entire condition is evaluated to be `true` and the `if` part of the `if` statement is executed and the statement `c++` is executed. As a result, the values of the variables `a`, `b` and `c` after executing the program segment are all 1.

Self-test 6.3

1 With the bitwise `|` operator, the value of the expression
`(Enrolment.COURSE_MT201|`
`Enrolment.COURSE_MT210|`
`Enrolment.COURSE_MT268)` is,

	TM222	T225	T223	T222	T202	S271	MT269	MT268	MT260	MT258	MT210	MT201	MST207	MST204	MT261	MT246	M245	M221	M205
`COURSE_MT201`	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
`COURSE_MT210`	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
`COURSE_MT201|` `COURSE_MT210`	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
`COURSE_MT268`	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
`COURSE_MT201|` `COURSE_MT210|` `COURSE_MT268`	0	0	0	0	0	0	0	1	0	0	1	1	0	0	0	0	0	0	0

Therefore, the bits for the courses MT201, MT210 and MT268 are set to be `1`. In the `addCourse()` method, the corresponding bits of the attribute `coursesTaking` are set to be `1` by the `|` operator. Therefore, the single method call with expression `(Enrolment.COURSE_MT201| Enrolment.COURSE_MT210| Enrolment.COURSE_MT268)` is equivalent to three different method calls to the `addCourse()` method. Each is supplied with an integer value for a particular course.

Similarly, the above argument applies to the `removeCourse()` method as well. It is possible to set the bits of the attribute `coursesTaking` for more than one course by a single method call with similar parameter value.

2 Suppose that an `Enrolment` object is added. Courses MT201 and MT210 and its attribute `coursesTaking` become:

	TM222	T225	T223	T222	T202	S271	MT269	MT268	MT260	MT258	MT210	MT201	MST207	MST204	MT261	MT246	M245	M221	M205
`coursesTaking`	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0

Then, calling the `contains()` method with a value, `Enrolment.COURSE_MT201|Enrolment.COURSE_MT210 | Enrolment.COURSE_MT268`, passed via parameter `course` the following expression is evaluated:

```
(coursesTaking & course) != 0
```

For the expression, `(coursesTaking & course)`, the value is obtained as

	TM222	T225	T223	T222	T202	S271	MT269	MT268	MT260	MT258	MT210	MT201	MST207	MST204	MT261	MT246	M245	M221	M205
`coursesTaking`	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
`course`	0	0	0	0	0	0	0	1	0	0	1	1	0	0	0	0	0	0	0
`coursesTaking &` `course`	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0

which is non-zero. Therefore, the expression

```
(coursesTaking & course) != 0
```

is evaluated to be `true`. This means that the interpretation of the above method call with a parameter value that contains more than one non-zero bit is to verify whether the `Enrolment` object contains at least one course that is specified by the parameter `course`.

If the real intention is to verify whether an `Enrolment` object contains all specified courses, the expression used in the `contains()` method should be modified to be:

```
(coursesTaking & course) == course
```

Then, the `contains()` method will verify that the courses added to the `Enrolment` object contain all the course(s) specified by the parameter course and it returns `false` in our case.

3 Enrolment.java

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

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

    // Immutable array object maintaining course names
    public static final String[] courseNames = {
        "Fundamentals of Computing",
        "Mathematical Methods",
        "Probability and Statistics",
        "Elements of Statistics",
        "Mathematics for Computing",
}       "Mathematical Models and Methods",
        "Mathematical Methods, Models and Modelling",
        "Computing Fundamentals with Java",
        "Computing Fundamentals",
        "Programming and Database",
        "Computer Architecture and Operating Systems",
        "Commercial Information Processing",
        "Commercial Information Systems and Programming",
        "Discovering Physics",
        "Analogue and Digital Electronics",
        "Electronics Principles and Digital Design",
        "Microprocessor-based Computers",
        "Analogue Circuits",
        "The Digital Computer"
    };

    // An array object is used to maintain the course selected.
    // Initially, all array elements are by default false.
    private boolean[] coursesTaking = new boolean[19];

    // Add a course to the Enrolment
    public void addCourse(int course) {
        coursesTaking[course] = true;
    }

    // Remove a course from the Enrolment
    public void removeCourse(int course) {
        coursesTaking[course] = false;
    }

    // Toggle a course in the Enrolment
    public void toggleCourse(int course) {
        coursesTaking[course] = !coursesTaking[course];
    }

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

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

As the method declarations are the same as the one discussed in the unit, it is possible to use the same `TestEnrolment` class to test the above class definition.

Compared with the `Enrolment` class that uses the bits of an integer value to represent the courses selected, the above class definition that uses an array object for storage is easier to understand. However, it is not possible to add, remove, or verify more than one course with a single method call with such an implementation, which is not as flexible as the one we discussed.

Self-test 6.4

The method `findState()` can be written as:

```
public String findState(int temp) {
    String state;
    if (temp < 0) {
        state = "ice";
    }
    else if (temp > 100) {
             state = "stream";
         }
         else {
             state = "water";
         }
    return state;
}
```

Another solution is to use the ternary conditional operators, such as:

```
public String findState(int temp) {
    return (temp < 0) ? "ice" :
           (temp <= 100) ? "water" : "steam";
}
```

Self-test 6.5

```
switch (mark / 10) {
    case 10:
    case 9: grade = "A"; break;
    case 8: grade = "B"; break;
    case 7: grade = "C"; break;
    case 6: grade = "D"; break;
    default: grade = "F";
}
```

Self-test 6.6

1

a
```
for (int i=0; i < 5; i++) {
    for (int j=0; j <= i; j++) {
        System.out.print("*");
    }
    System.out.println();
}
```

b

```
for (int i=0; i < 5; i++) {
    for (int j=0; j < 5 - i; j++) {
        System.out.print("*");
    }
    System.out.println();
}
```

c

```
for (int i=0; i < 5; i++) {
    for (int j=0; j < 5; j++) {
        if (j < i) {
            System.out.print(" ");
        }
        else {
            System.out.print("*");
        }
    }
    System.out.println();
}
```

d

```
for (int i=0; i < 5; i++) {
    for (int j=0; j < 4 - i; j++) {
        System.out.print(" ");
    }
    for (int j=0; j < i * 2 + 1; j++) {
        System.out.print("*");
    }
    System.out.println();
}
```

2 MultiplicationTable.java

```
// Definition of class MultiplicationTable
public class MultiplicationTable {

    // Show a nine by nine multiple table
    public void show() {
        // Outer loop that displays lines of multiples
        for (int i=0; i < 81; i++) {
            // Determine the row and column number
            int row = i / 9 + 1;
            int col = i % 9 + 1;
            System.out.print(row * col + "\t");

            // If it is the last number of the row,
            // skip to next new line
            if (col == 9) {
                System.out.println();
            }
        }
    }
}
```

Self-test 6.7

Both `break` statements and `continue` statements can be used. If the `break` statement is used, it terminates the `switch/case` structure. As there is no statement following the `switch/case` structure, the update part of the `for` loop is executed immediately. However, if the `continue` statement is used, it skips all the statements in the loop body and the next statement to be executed is the update part of the `for` loop. Therefore, the two statements can be used in the program segment.

Self-test 6.8

TableMaker.java

```
// Definition of class TableMaker
public class TableMaker {
    // Attribute to refer to a two-dimensional array
    private int[][] table;

    // Create a table according to the given row and column
    public void createTable(int row, int column) {
        // Create the two-dimensional array
        table = new int[row][column];

        // A nested loop to store the values in a
        // multiplication table
        for (int i=0; i < row; i++) {
            for (int j=0; j < column; j++) {
                table[i][j] = (i + 1) * (j + 1);
            }
        }
    }

    // Print the values of the table on the screen
    public void printTable() {
        // Check whether the table is created
        if (table != null) {
            // A nested for loop to display the values
            for (int i=0; i < table.length; i++) {
                for (int j=0; j < table[i].length; j++) {
                    System.out.print(table[i][j] + "\t");
                }
                System.out.println();
            }
        }
    }
}
```

Self-test 6.9

1 Mathematically, the function sum(*n*) can be interpreted to be:

$$\begin{aligned} sum(n) &= 1 + 2 + 3 + ... + (n-1) + n \\ &= sum(n-1) + n \end{aligned}$$

Therefore, sum(*n*) = *n* + sum(*n-1*).

2 The base case of the expression $1 + 2 + 3 + ... + n$ is that when the value of *n* is `1`. That is, the expression is simply `1` only. That is:

sum(1) = 1.

3 **Summation.java**

```
// Definition of class Summation
public class Summation {

    // The recursive method for finding the sum of
    // values from 1 to n
    public int sum(int n) {
        // Check if it is the base case for n = 1
        if (n == 1) {
            // If it is the base case, return 1
            return 1;
        }
        else {
            // Derive the parameter for the problem of size n-1
            int nMinus1 = n - 1;
            // Find the result of the problem of size n-1
            int resultForMinus1 = sum(nMinus1);
            // Derive the result of the current problem of size n
            int result = n + resultForMinus1;
            // Return the result
            return result;
        }
    }
}
```

TestSummation.java

```
// Definition of class TestSummation
public class TestSummation {

    // Main executive method
    public static void main(String args[]) {
        // Check if a number is provided as program parameter
        if (args.length == 0) {
            // If no program parameter is provided, show usage message
            System.out.println("Usage: java TestSummation number");
        }
        else {
            // Create a Summation object
            Summation finder = new Summation();
            // Get the number for getter the summation
            int number = Integer.parseInt(args[0]);
            // Find the result of the summation
            int result = finder.sum(number);
            // Show the result to the screen
            System.out.println("sum(" + number + ") = " + result);
        }
    }
}
```