MT201

Unit 4

Control structures

Course team

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

The previous three units discuss some fundamental concepts and ideas in computing, including the relationship between hardware and software, object-oriented software development and problem analysis using the object-oriented approach.

Unit 2 discusses how to analyse a problem and derive list of classes and their attributes and behaviours. You can use the class definitions and primitive types introduced in *Unit 3* to model an object and the object's attributes. Furthermore, methods were introduced for modelling the object's behaviours.

Unit 3 mentions that the object attributes should be retrieved or updated via its methods instead of directly accessing its attributes, because the object 'knows' how to maintain its own status or state. Whenever the object is sent a message to update its state, the object should verify whether it is right to continue the action or just ignore it. As a result, the method should have some intelligence to determine whether to do or not to do, or to do this or do that, in some other ways. In programming language, this can be done with *branching*.

Another common phenomenon of object behaviours is that an object has to perform the same (or mostly similar) operations repeatedly for a predetermined number of times or until a particular condition is met. An operation that is performed repeatedly forms a loop, known as *looping* in programming language. Branching and looping are two common building blocks in programming languages. This unit discusses how the Java programming language implements these two structures.

While you are compiling and executing your program, you can be sure that it will encounter problems. This unit also discusses the way to identify and resolve these problems through a process known as *debugging*.

You can learn programming more easily and understand it better by editing, compiling and executing programs rather than just reading the materials. Therefore, you should have your computer ready and execute the example programs while you studying this unit. Otherwise, you should follow the instructions provided in *Unit 1* to install the Java Software Development Kit (Java SDK) properly.

Please feel free to modify the programs to see what they output and then try to figure out the reasons why. This will make your programming learning much more challenging, more fun and more rewarding.

Objectives

By the end of *Unit 4*, you should be able to:

1. *Explain* the order of executing program statements.
2. *Write* simple relational expressions.
3. *Apply* different Java branching control structures without nesting.
4. *Apply* different Java looping control structures without nesting.
5. *Apply* simple debugging techniques.
6. *Describe* the importance of the single-entry-single-exit principle.

Program execution order

In the simplest form of programming, program statements (or instructions) are executed one-by-one, that is, sequentially. The order of the program execution is also known as the *flow of control*.

The flow of control can be visualized through the flowchart in Figure 4.1:

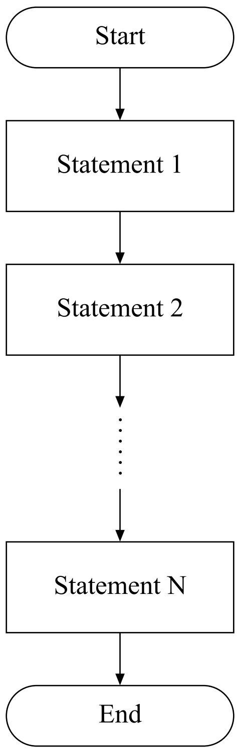


Figure 4.1 Sequential flow of control

The computer sequentially executes the statements according to their order in the program. However, if the instructions in a program are executed sequentially only, the program’s functionalities are quite limited. In some situations, it is essential that a program can make some decisions, such as:

1 It is much more reliable to check that the divisor is non-zero beforehand, because dividing a number by zero is undefined.

2 To update an object state by sending a message with a value to it, the object should verify whether the value is a proper one before assigning it to the attribute.

Furthermore, simply executing a fixed sequence of statements cannot solve some problems. For most algorithms for solving problems, it is necessary to determine the statements that should be executed based on some criteria or whether to execute a sequence of statements repeatedly.

Therefore, a program can be made more powerful and more reliable if it can test some conditions and take some appropriate action and if it can execute a sequence of statements repeatedly.

Simple relational expressions

A computer can test if certain conditions are true or false by performing comparisons, such as comparing two numerical values (e.g. `3 < 5`, which is `true`). To construct a relational expression, some relational operators are needed to determine whether the whole condition is `true` or `false`.

Please use the following reading to get a basic idea of how to perform comparisons in the Java programming language. The materials presented in the reading are reviewed and elaborated on.

> ***Reading***
>
> King, Section 4.1, pp. 128–32. Don't forget to try the review questions on p. 132. You can find the answers on p. 178.

Comparing primitive types of data

From the reading, you now know that there are six relational operators. They are shown in Table 4.1.

Table 4.1 The relational operators

Relational operator	**Meaning**
`<`	Less than
`>`	Greater than
`<=`	Less than or equal to
`>=`	Greater than or equal to
`==`	Equal to
`!=`	Not equal to

All six relational operators above can be used for all numeric primitive types; that is, for `byte, short, char, int, long, float` and `double`. These numeric types can be placed on either side of a relational operator in a Java language expression.

For example, if `a`, `b` and `c` are variables of any numeric primitive type and `getReading()` is the method of an object of class `TicketCounter` that returns a numeric value referred by a variable `tc`, the following relational comparisons are all legal:

1 `a < 0`

2 `a >= b`

3 `a + b <= c – 10`

4 `tc.getReading() > 0`

The results of all relational operations are either `true` or `false`. Please remember that both `true` and `false` are reserved words in the Java programming language. The compiler determines whether either one or both operand(s) of the relational operator need to be converted so that an ‘apple-to-apple’ comparison can be made. An example is presented in the reading. It shows that before a numeric `int` type value is compared with another of type `double`, the value of the `int` type is converted to `double`. Otherwise, directly comparing an `int` value to a `double` value would be comparing ‘apples to oranges’.

Arithmetic operators (such as `+` and `/`) take precedence over all relational operators. Therefore, if an expression includes both arithmetic operators and relational operators, arithmetic operators are always performed before relational operators. However, you can make your program more readable by adding parentheses.

In Java, there are two operators that look similar: `=` (single = character) and `==` (double = character). The `=` operator is used to perform an *assignment* operation that assigns the value of the expression on its right-hand side to the variable on its left-hand side. The `==` operator is a relational operator that checks whether the value of the expression on one side is equal to that on the other side.

The exclamation mark (`!`) in Java always means ‘negation’ or ‘not’. Therefore, the relational operator, *not equal to*, is written `!=`.

Do you think `true` is greater than `false` or `false` is greater than `true`? You can see that it does not make sense to give an order to the two Boolean values `true` and `false`. Therefore, for `boolean` types, only `==` (Equal to) and `!=` (Not equal to) are applicable, but `boolean` values are rarely put to an equality test. For example, what do you think about the following relational comparison?

```
(a > b) == false
```

The above comparison is equivalent to the following:

```
(a <= b)
```

The latter format is shorter and easier to understand. If you should ever write a relational comparison in the former format, you should think again whether there is an equivalent and shorter option.

We said in *Unit 3* that integral values are stored *exactly* in computers, but computers only store numeric values with fractional parts (floating-point) using approximations. Integral division gives an integer result only, which might involve truncation (ignoring the decimal part). Floating-point calculation results are also just approximations, as they might involve rounding-off during the calculation. You should be careful that the following condition,

```
a / b * b == a
```

which is true in mathematics, is not necessarily `true` in computers no matter what types of variable `a` and `b` are. For floating-point calculations, your relational comparisons should allow round-off errors by checking whether the results are close to your expected values. For example, in order to check whether a variable `a` of type `double` equals `0.85`, instead of checking

```
a == 0.85
```

you might consider using the following expression instead

```
Math.abs(a – 0.85) < 0.0000001
```

where the class method `abs()` of the `Math` class determines and returns the absolute value of the method argument (by ignoring the sign). It is equivalent to the following mathematical expression:

$$|a - 0.85| < 0.0000001$$

The value `0.0000001` is an arbitrary value of the tolerable round-off error of the equality comparison, though you should choose a suitable value case-by-case.

Comparing objects

Unit 3 said that any variable of a type other than the eight primitive types is a non-primitive type, which can only store a reference to an object. The analogy is that a non-primitive type variable can be considered a little box that stores the location of an entity. For example

```
TicketCounter counter1, counter2;
```

declares two variables `counter1` and `counter2`, which means that there are two little boxes named `counter1` and `counter2` that can store the locations of ticket counters. Then, how can you answer the following two questions?

1 Do the two little boxes store the location of the same ticket counter?

2 Are the readings of the ticket counters referred to by the two little boxes the same?

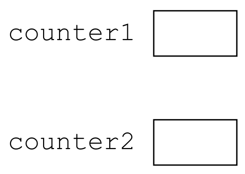
You can use the following relational expression to test the first question:

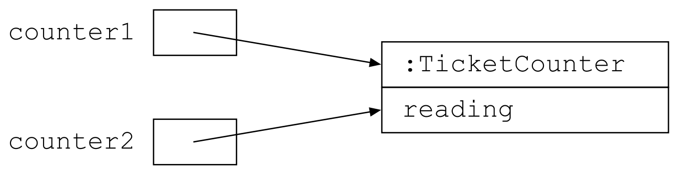
```
counter1 == counter2
```

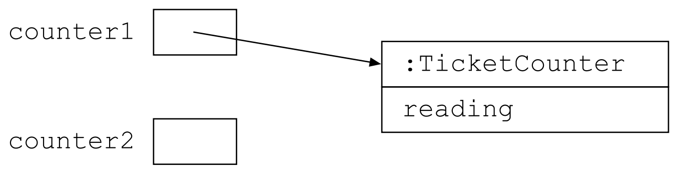
The interpretation of the above expression is that it tests whether the contents of two variables are the same. As the content of a non-primitive variable can be considered the message in the little box that tells the location of a ticket counter, they are the same. This means that they are referring to the same ticket counter. You should notice that if neither little box refers to any ticket counter (that is, the contents of both variables are `null`), such a relational expression result is still `true`.

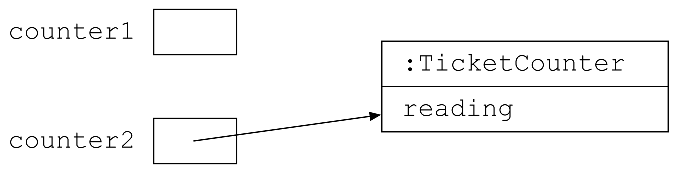
The five possible scenarios are shown in Figure 4.2.

Scenario	**Result**


Program segment that creates the scenario:
`counter1 = null;`
`counter2 = null;`
	`true`


Program segment that creates the scenario:
`counter1 = new TicketCounter();`
`counter2 = counter1;`
	`true`


Program segment that creates the scenario:
`counter1 = new TicketCounter();`
`counter2 = null;`
	`false`


Program segment that creates the scenario:
`counter1 = null;`
`counter2 = new TicketCounter();`
	`false`

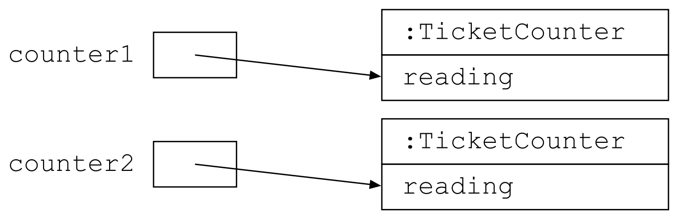
 counter1
counter2
:TicketCounter
reading
:TicketCounter
reading
Program segment that creates the scenario:
`counter1 = new TicketCounter();`
`counter2 = new TicketCounter();`
	`false`

Figure 4.2 Five possible scenarios of checking if `counter1` == `counter2`

For the second question, how can you determine whether the readings of the ticket counter(s) referred to by two different reference variables are the same?

First of all, there are two possible situations:

1 The reference variables are referring to the same ticket counter.

2 They are referring to different ticket counters.

If they are referring to the same ticket counter, the readings (determined via variables `counter1` and `counter2`) are the same.

For the latter, they are referring to different counters (see Figure 4.3 for an example), and you should get their readings for comparison.

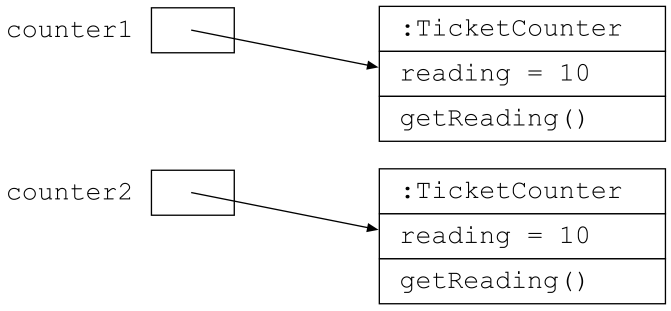


Figure 4.3 Example of two counter variables referring to two different counter objects

You can use the following relational expression to do so:

```
counter1.getReading() == counter2.getReading()
```

The readings of the ticket counters that are referred to by `counter1` and `counter2` are obtained by the methods on the left-hand side and right-hand side respectively, and the equality check is carried out afterwards. The result of the relational expression is `true` even if they are different ticket counters.

You should notice that if `counter1` and `counter2` are actually referring to the same ticket counter, the result is definitely `true`. That is the case in Figure 4.4:

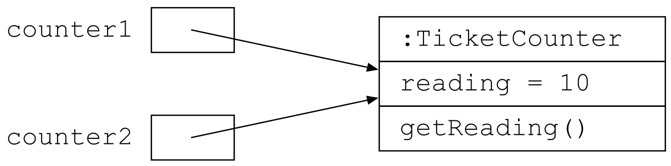


Figure 4.4 Example of two counter variables referring to the same counter object

If you are only concerned about the readings of the ticket counter(s) that are referred to by the two variables, you should use

```
counter1.getReading() == counter2.getReading()
```

instead of

```
counter1 == counter2
```

What do you think makes two objects of the same class 'equal'? In the ticket counter example, two ticket counters with the same reading can be considered 'equal'. For two bank account objects, however, do you think they are 'equal' if the account owners are the same person or their account balances are the same? It is up to the programmer (or designer) of the class, not the users of the class, to determine the meaning of 'equal'.

In Java, you can provide an `equals()` method to the class `TicketCounter` so that you can use either of the following two equivalent statements to determine whether two objects are equal, with respect to your own intention.

```
counter1.equals(counter2)
counter2.equals(counter1)
```

In the Java programming language, some classes in the library that comes with the Java SDK have their own `equals()`method pre-defined. A typical example is the class `String`. Therefore, you can, and you should, always use the following statement to determine whether the contents of the two `String` objects referred to by variables `s1` and `s2` are the same:

```
s1.equals(s2)
```

If you use the usual `==` (double = characters) operator,

```
s1 == s2
```

is `false` even if the contents of the `String` objects referred to may be equivalent (as in Figure 4.5). That is:

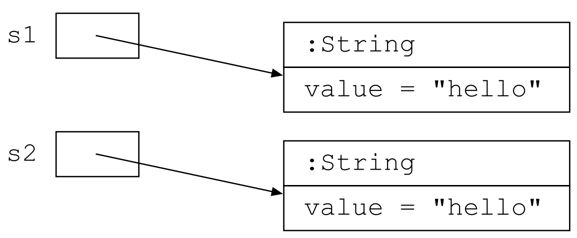


Figure 4.5 Example of two string variables referring to two different string objects with the same content

Therefore, the former statement `s1.equals(s2)` is more suitable for comparing two `String` objects.

Precedence of relational operators

We said that a relational expression could include arithmetic expressions on either side of the relational operators. For example,

```
a * b – c > 0
```

or even involve methods that return values, such as:

```
tc1.getReading() > tc2.getReading() + 1
```

You know that multiplication should always be performed before addition in an expression, and the evaluation of the expression is done from left to right. In programming language, such order is governed by precedence and associativity. The following reading can help you understand how calculations are really carried out. Furthermore, you will learn what operator precedence and associativity are and how they govern the way that a calculation is done.

Reading

King, Section 2.7, pp. 46–52. Try the few review questions on p. 52 when you've finished the reading. The answers are on p. 83.

The reading provides you with valuable information about how a calculation is done. You should bear in mind that integer operations, especially division, involve truncations, whereas there are round-off errors in floating-point calculations.

You also came across the terminology 'precedence' and 'associativity' in the reading. First of all, the operators of higher precedence are performed before those with lower precedence. If there is more than one operator with the same precedence, their orders are determined using associativity.

For example, in the following expression

```
a * b / c
```

both `*` and `/` are of the same precedence, as they are all left-associative (operations on the left-hand side are done first). The expression is interpreted as:

```
(a * b) / c
```

(If operators `*` and `/` were right-associative, the expression `a * b / c` would have been interpreted as `a * (b / c)`.)

Among the operators introduced in the reading, there are right-associative operators — unary `+` and `-` operators — that even take precedence over the `*` and `/` operators. Therefore, the following expression

```
a * -b / c
```

is interpreted as

```
(a * (-b)) / c
```

Table 4.2 summarizes all operators discussed so far.

Table 4.2 Precedence of operators

Precedence	**Associativity**	**Remarks**
`+ -`	Right	Unary operators
`* / %`	Left	
`+ -`	Left	
`>= > < <=`	Left	
`== !=`	Left	
`= += -= *= /= %=`	Right	Assignment operators

You can see from Table 4.2 that arithmetic operators take precedence over all relational operators. Therefore, you can have the following relational expression

```
a + b * c > 10
```

and, without using any parentheses, it is interpreted as:

```
((a + (b * c)) > 10)
```

The calculation `(a + (b * c))` is done before comparing the result with the literal (the constant) `10` and the overall result is therefore either `true` or `false`.

The last row in the above table contains several assignment operators. We frequently use the assignment (operator `=`) to update the content of a variable. The others are compound assignment operators; the interpretation is that an expression `a += 2` is the same as `a = a + 2`. That means adding `2` to the variable `a`. The cases for other compound assignment operators are the same.

As the precedence of assignment operators is the lowest among all operators, the following expression

```
a *= b + c
```

is considered to be

```
(a *= (b + c))
```

Therefore, it is equivalent to

```
a = a * (b + c)
```

instead of

```
a = (a * b) + c
```

Self-test 4.1

1 Evaluate the following expressions:

a `5 + 4 / 3 * 2`

b `(3 + 2) * 4`

c `33 * 3 < 99`

d `(4 + 3) * 2 != 12`

2 Convert the following mathematical expressions into valid arithmetic expressions in the Java programming language:

a $\frac{x1 - x2}{y1 - y2}$

b mc^2

c $ut + \frac{at^2}{2}$

3 Determine the values stored in the variable `n` after executing the following sets of statements:

a
```
int n = 10;
n % = 3;
```

b
```
int n = 15;
n *= n - 5;
```

Branching

The previous section says that most programs need to evaluate some conditions so that they can perform different actions depending on the outcome of the evaluation. This is the reason why you have to understand how to construct different conditions in the Java programming language.

A program can be written to conditionally perform a group of instructions based on a Boolean expression result or the result of a condition (so that there is a different path or flow of control). Such a control structure is known as branching. The overall result of a Boolean expression is either `true` or `false`. The previous section also says that simple relational expressions are Boolean expressions. Also, there are logical expressions that give either `true` or `false` results. These are discussed in *Unit 6*.

`if` statements

In the Java programming language, you can perform an extra sequence of operations if the result of a condition (in an `if` statement) is `true`. An `if` statement is written in the following format:

```
if (condition) {
   statement(s)    // statement(s) to be executed if condition is true
}
```

The *condition* in the parentheses must be a Boolean expression that gives a Boolean result; that is, either `true` or `false`. For example, it can be a simple relational expression, discussed in the previous section. If the result of the condition is evaluated to be `true`, the *statements* enclosed in the curly brackets (`{` and `}`) are executed.

The functionality of an `if` statement is shown in Figure 4.6.

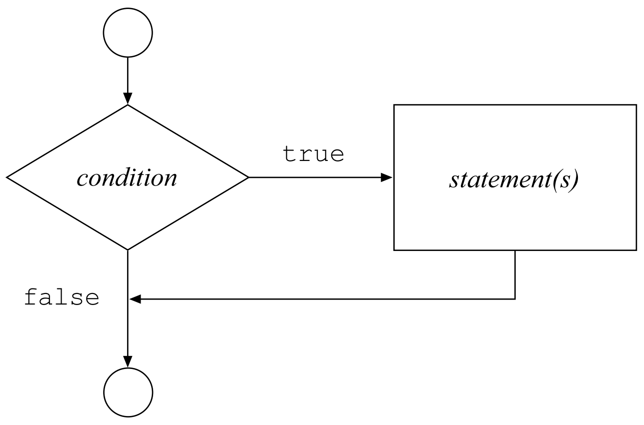


Figure 4.6 if statement

If there is only one statement to be executed if the condition is evaluated to be `true`, the pair of curly brackets is optional. The `if` statement becomes:

```
if (condition)
   statement    // statement to be executed if the condition is true
```

The above format is not preferable, because if you later want to add statements to be executed, you may forget to add the pair of curly brackets. The program can be compiled without any error message, but the execution result will be wrong. For example,

```
if (condition)
   statement-1;
   statement-2;
```

is interpreted as

```
if (condition)
   statement-1;
statement-2;
```

The indentation of *statement-2* in the first program segment above may just mislead you.

Therefore, it is always a good habit (convention) to use a pair of curly brackets even if there is only one statement to be executed.

Let's revisit the ticket counter you came across in *Unit 3*. Its class definition is shown in Figure 4.7.

```
// The class definition of a ticket counter
class TicketCounter {
    private int reading; // The ticket counter reading

    // To increase the reading by one
    public void increase() {
        reading = reading + 1;
    }

    // To increase the reading specified by the parameter, amount
    public void increaseByAmount(int amount) {
        reading = reading + amount;
    }

    // To get the ticket counter reading
    public int getReading() {
        return reading;
    }

    // To set a value to the ticket counter reading
    public void setReading(int newReading) {
        reading = newReading;
    }
}
```

Figure 4.7 TicketCounter.java (from *Unit 3*)

Suppose that you want to model a ticket reader with a three-digit reading. Therefore, the largest number it can represent is 999. The next number to be shown when it is incremented again is 000. The methods `increase()` and `increaseByAmount()` have to be modified then.

The logic of the `increase()` method is modified to:

> Increase the reading by one.
> If the reading is greater than 999, it is reset to zero.

The above steps can be implemented in the Java programming language as:

```
// To increase the reading by one
public void increase() {
    reading = reading + 1;
    if (reading > 999) {   // verify whether reading > 999
        reading = 0;       // if true, reset reading to 0
    }
}
```

In the Java programming language, the pair of curly brackets is used to construct a *statement block*. Although it is not a requirement of the programming language, we usually indent the statements in a block so that it is easier to identify the block properly; hence, the readability of the program is enhanced. For example:

```
                    void increase() {
                      ┌   reading = reading + 1;
                      │   if (reading > 999) {
statement block ──────┤       reading = 0;  ┐── statement block
                      │   }                 ┘
                      └
                    }
```

The above method can be visualized as in Figure 4.8.

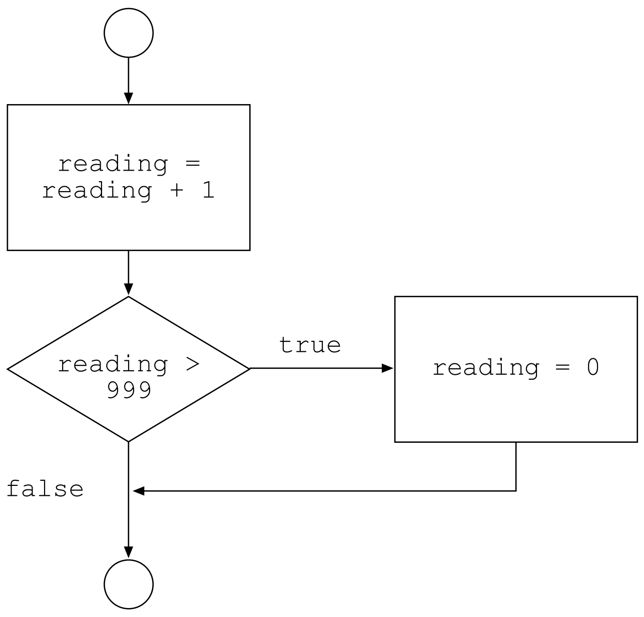


Figure 4.8 Visualization of the increase() method

For example, when the current value of the `reading` attribute of the ticket counter is `123`, a message is sent to it to make it perform the `increase` operation. The statement

```
reading = reading + 1;
```

is executed. The result `124` is obtained and is assigned to the attribute `reading` afterwards. The statement that verifies the condition then follows:

```
if (reading > 999) {
   ... // statement
}
```

Since `124` is not greater than `999`, the condition is `false` and the *statement* in the block is skipped.

By contrast, when the current value of the `reading` attribute of the ticket counter is already `999`, it performs the `increase` operation when it receives a message `increase`. Then, the following statement

```
reading = reading + 1;
```

assigns a new value `1000` to the attribute `reading`. Then, the condition `reading > 999` is `true` because `1000` is greater than `999`, and the block of the statement that follows the condition is now executed:

```
…… {
    reading = 0;
}
```

The variable `reading` is now updated to zero.

The `increaseByAmount()` method is similar. The difference is that after the `reading` is increased by the amount, the reading is not necessarily `1000` but might be `1001` or `1002`. The reading can be rectified by performing a remainder operation (`%`) with `1000` from the current value of the `reading`. Therefore, the method `increaseByAmount()` can be modified as:

```
// To increase the reading specified by the parameter, amount
public void increaseByAmount(int amount) {
    reading = reading + amount;
    if (reading > 999) {          // verify whether reading > 999
        reading = reading % 1000; // if true, rectify reading
    }
}
```

For example, when the current reading of the ticket counter is `997`, it receives an `increaseByAmount` message with supplementary data `5` (via a parameter). The statement

```
reading = reading + amount;
```

is executed and a value of `1002` is assigned to the attribute `reading`. Then, the condition `reading > 1000` is verified. The result is `true` and the statement block that follows the condition is executed, which means that the following statement

```
reading = reading % 1000;
```

is executed, and a new value `2` is assigned to the attribute `reading`. The above statement is a generic way to rectify the value of the attribute `reading`; this statement can be used to replace the statement in the `increase()` method. That is, the `increase()` method can also be implemented as:

```
// To increase the reading by one
public void increase() {
    reading = reading + 1;
    if (reading > 999) {          // verify whether reading > 999
        reading = reading % 1000; // if true, rectify the reading
    }
}
```

Unit 3 states that updating an object attribute via a method is much safer and more reliable; the object knows whether it is appropriate to update the state or just to ignore it. The `setReading()` method of the `TicketCounter` class can be modified to:

```
public void setReading(int theReading) {
    if (theReading >= 0) {
        reading = theReading % 1000;
    }
}
```

Before the attribute `reading` is updated with the value of the parameter `theReading`, the value of `theReading` is checked to see whether it is non-negative. If so, the attribute is updated with the remainder from dividing `theReading` by `1000`, as the ticket counter supports three digits only. For example, if the given value in the parameter `theReading` is `1234`, the attribute `reading` will be updated as `1234 % 1000`, which yields `234`.

If the value of the parameter `theReading` is negative, the result of the condition of the `if` statement is `false` and no statement is executed. This means that the value of the attribute `reading` is kept unchanged. It is preferable to display an error message to notify the user, but it needs a variation of the `if` statement (discussed in the next section). Therefore, for the time being, the `setReading()` method is written to update the attribute if the value passed via the parameter is valid.

The class definition of the ticket counter becomes as shown in Figure 4.9.

```
// The class definition of a ticket counter
public class TicketCounter {
    private int reading;     // The ticket counter reading

    // To increase the reading by one
    public void increase() {
        reading = reading + 1;
        if (reading > 999) {          // verify whether reading > 999
            reading = 0;              // if true, reset reading to 0
        }
    }

    // To increase the reading specified by the parameter, amount
    public void increaseByAmount(int amount) {
        reading = reading + amount;
        if (reading > 999) {          // verify whether reading > 999
            reading = reading % 1000; // if true, rectify reading
        }
    }

    // To get the ticket counter reading
    public int getReading() {
        return reading;
    }

    // To set a reading to the ticket counter reading
    public void setReading(int theReading) {
        // Verify the parameter to be non-negative before updating
        // the attribute reading
        if (theReading >= 0) {
            reading = theReading % 1000;
        }
    }
}
```

Figure 4.9 A new version of TicketCounter.java

By using `if` statements in the three methods, the object can be modelled closer to a real-world ticket counter.

To illustrate the use of the modified `TicketCounter` class, the class `TestCounter` is shown in Figure 4.10.

```
// The class definition of a ticket counter tester.
public class TestCounter {
    // The main executive
    public static void main(String args[]) {
        // Create the ticket counter and have it accessible via counter1
        TicketCounter counter1 = new TicketCounter();

        // Set the initial reading
        counter1.setReading(124);
        System.out.println("The initial reading is: " +
                           counter1.getReading());

        // Increase the counter reading
        counter1.increase();
        System.out.println("The reading after 'increase' is: " +
                           counter1.getReading());

        // Set the reading to a specific value
        counter1.setReading(997);
        System.out.println("The reading is set to be " +
                           counter1.getReading());

        // Increase the counter reading by an amount
        counter1.increaseByAmount(5);
        System.out.println("The reading after 'increaseByAmount' is: " +
                           counter1.getReading());
    }
}
```

Figure 4.10 TestCounter.java

The sole purpose of the `main()` method of the `TestCounter` class is to set up an initial environment in which a ticket counter object is created and is accessible via the reference variable `counter1` by the following statement:

```
TicketCounter counter1 = new TicketCounter();
```

It can send a *message* with supplementary data *parameter* to the ticket counter object referred by `counter1` using:

`counter1`.*message*(*parameter*);

The following two statements first update the value of the attribute `reading` of the ticket counter object to `124` and display its value on the screen:

```
counter1.setReading(124);
System.out.println("The initial reading is: " +
                   counter1.getReading());
```

The program displays the following message on the screen:

```
The initial reading is: 124
```

Then, the subsequent two statements

```
counter1.increase();
System.out.println("The reading after 'increase' is: " +
                   counter1.getReading());
```

increase the attribute `reading` of the ticket counter object by one, by sending a message `increase` to it and show the value of the attribute `reading` on the screen.

```
The reading after 'increase' is: 125
```

The value of the attribute `reading` is then set to `997` by sending a message `setReading` to the ticket counter object again. Its value is shown on the screen by executing the following two statements:

```
counter1.setReading(997);
System.out.println("The reading is set to be  +
                   counter1.getReading());
```

The output shown on the screen is:

```
The reading is set to be 997
```

The value of the attribute `reading` of the object is increased by `5` by sending a message `increaseByAmount` with supplementary data `5`:

```
counter1.increaseByAmount(5);
```

The attribute `reading` is increased from `997` to `1002` and is changed to `2` by the expression (`1002 % 1000`) in the method `increaseByAmount()`. Therefore, after executing the `increaseByAmount()` method with parameter `5`, the value of the attribute `reading` is updated as `2`.

The last statement in the `main()` method

```
System.out.println("The reading after 'increaseByAmount' is: " +
                   counter1.getReading());
```

displays the value of the attribute reading on the screen.

```
The reading after 'increaseByAmount' is: 2
```

This is the end of the `main()` method of the `TestCounter` class, and the program terminates.

After compiling the `TestCounter.java` file by entering the following command at the command prompt

```
javac TestCounter.java
```

the compiler will automatically compile the `TicketCounter`. To execute the program, enter the following command

```
java TestCounter
```

and you will get the following output:

```
The initial reading is: 124
The reading after 'increase' is: 125
The reading is set to be 997
The reading after 'increaseByAmount' is: 2
```

Self-test 4.2

1 Suppose that the ticket reader now supports four digits and the maximum reading is 9999. Please modify the `TicketCounter` class to support such a change.

2 A student typed the source code for the `TicketCounter` class by himself. However, by mistake, the `increase()` method was typed as:

```
// To increase the reading by one
void increase() {
    reading = reading + 1;
    if (reading > 999); {         // verify whether reading > 999
        reading = 0;              // if true, reset reading to 0
    }
}
```

A semi-colon is typed after the parentheses that enclose the condition `reading > 999`. Is this a syntax error that can be detected during compilation? If the program can be compiled successfully and you run the `TestCounter` program by `java TestCounter` again, what are the outputs — and why?

The following two examples further illustrate the use of `if` statements in other situations.

Example: finding the maximum and minimum of a series of numbers

In some applications, it is necessary to find the maximum and minimum of a series of numbers. For example, if you are going to plot a chart of some points, you have to determine the minimum and maximum x-coordinates and y-coordinates respectively so that you can set the ranges and scales of the axes properly.

To solve the problem, you can design an object of class `MinMaxFinder` that has attributes for current minimum and maximum named `currentMin` and `currentMax` respectively. Furthermore, the class needs three behaviours:

1 `testNumber` — to request the `MinMaxFinder` object to perform this behaviour, a supplementary number is presented, and the number is tested against the current minimum and maximum. If the number presented is larger than the current maximum, the current maximum is updated to be the presented number. Similarly, if the presented number is smaller than the current minimum, the current minimum is updated accordingly.

2 `getMinimum` — queries the `MinMaxFinder` object to obtain the current minimum.

3 `getMaximum` — queries the `MinMaxFinder` object to obtain the current maximum.

Based on the design above, the `MinMaxFinder` class can be represented as shown in Figure 4.11.

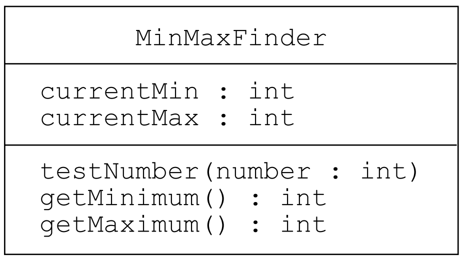


Figure 4.11 The `MinMaxFinder` class

The two methods, `getMinimum()` and `getMaximum()` are trivial. For the `testNumber()` methods, the algorithm is:

```
if the given number is less than the current minimum,
    assign the given number to the current minimum
if the given number is greater than the current maximum,
    assign the given number to the current maximum
```

Therefore, the method `testNumber()` is written as shown in the following Java method:

```
    // Present a number to the object for testing with current
    // minimum and maximum
    public void testNumber(int number) {
        // Compare the given number with current maximum
        if (number > currentMax) {
            // If the given number is greater than current maximum
            // assign the given number to the current maximum
            currentMax = number;
        }

        // Compare the given number with the current minimum
        if (number < currentMin) {
            // If the given number is less than the current mininum
            // assign the given number to the current minimum
            currentMin = number;
        }
    }
```

With other attribute and method declarations, the `MinMaxFinder` is written as shown in Figure 4.12.

```
// The class definition for the minimum/maximum finder
public class MinMaxFinder {
    // the current minimum
    private int currentMin = Integer.MAX_VALUE;
    // the current maximum
    private int currentMax = Integer.MIN_VALUE;

    // Present a number to the object for testing with current
    // minimum and maximum
    public void testNumber(int number) {
        // Compare the given number with current maximum
        if (number > currentMax) {
            // If the given number is greater than current maximum
            // assign the given number to the current maximum
            currentMax = number;
        }

        // Compare the given number with the current minimum
        if (number < currentMin) {
            // If the given number is less than the current mininum
            // assign the given number to the current minimum
            currentMin = number;
        }
    }

    // retrieve the current minimum
    public int getMinimum() {
        return currentMin;
    }

    // retrieve the current maximum
    public int getMaximum() {
        return currentMax;
    }
}
```

Figure 4.12 MinMaxFinder.java

The definitions of the two attributes of the class `MinMaxFinder` — `currentMin` and `currentMax` — are:

```
    // the current minimum
    private int currentMin = Integer.MAX_VALUE;
    // the current maximum
    private int currentMax = Integer.MIN_VALUE;
```

The above two statements declare the two attributes named `currentMin` and `currentMax`; the initial values assigned to them are `Integer.MAX_VALUE` and `Integer.MIN_VALUE` respectively. In the Java programming language, `Integer.MAX_VALUE` and `Integer.MIN_VALUE` represent the largest possible value and the smallest possible value of type `int`.

Such an initialization is required, because all possible values of type `int` must be less than or equal to `Integer.MAX_VALUE`. Then, when the `MinMaxFinder` object verifies the first number, the result of the comparison:

```
(number < currentMin)
```

must be `true` and the first number is assigned to the attribute `currentMin`.

Similarly, `currentMax` is initialized to be the same as `Integer.MIN_VALUE` so that the result of the following comparison to be tested for the first number

```
(number > currentMax)
```

must be `true` and the first number is assigned to the attribute `currentMax`.

Now, the class definition of `MinMaxFinder` is ready. What you need now is a driving program that creates a `MinMaxFinder` object that subsequently sends messages to it and gets the result afterwards. Such a program is actually another class definition but is not used to model any real-world entity. It is just to set up the necessary environment and to send messages to the objects so that they will perform their operations and solve the problem cooperatively. It is like the manager in an office who assigns jobs to subordinates so that they will complete their own jobs and thereby complete (solve) the functions of the office.

The significance of such a driver program is that it must have a special method `main()`; its format is:

```
public static void main(String args[]) {
    // Create the necessary objects
    ......

    // Send messages to the objects so that they perform the
    // desired operations
    ......

    // Send messages to the objects to get the results
    ......
}
```

This method has to be declared as `public` and `static`. *Unit 3* tells us that the keyword `public` governs the access to the method. The method `main()` must be marked as `public`, which means that it must be publicly accessible by all. The `main()` method must be a class method and therefore requires a `static` modifier.

There is no return value from the method `main()` and hence the keyword `void` in the method declaration. It accepts an array of `String` objects, but the issue of arrays is not discussed until *Unit 5*.

When a Java application is started, the statements in such a `main()` method are executed. The usual pattern of the `main()` method is to create the necessary objects first. Afterwards, it sends messages to the created objects so that the objects carry out their own behaviours accordingly. Last, it gets the results from the objects and shows them to the user.

For example, the following is a driver program for the class `MinMaxFinder` named `TestMinMaxFinder` . Its definition is shown in Figure 4.13.

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

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

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

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

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

Figure 4.13 TestMinMaxFinder.java

Before running the above program, you need to compile the two Java source files, `MinMaxFinder.java` and `TestMinMaxFinder.java`, first. If the two source files are ready, you can simply place the two files in the same folder in your hard disk and compile `TestMinMaxFinder.java`. The Java compiler will detect that the source files of the class `TestMinMaxFinder` uses the class definition of `MinMaxFinder` and will compile the file `MinMaxFinder.java` automatically as well.

Therefore, you can start the Command Prompt with your Microsoft Windows and change the current directory to the folder that contains the two Java source files. Enter the following command to compile the files:

```
javac TestMinMaxFinder.java
```

If the two Java source files are free of errors, the compiler will generate two class files named `MinMaxFinder.class` and `MinMaxFinderTester.class`.

To execute the program, enter

```
java TestMinMaxFinder
```

at the command prompt. The program will give you the result:

```
Minimum number = 10
Maximum number = 20
```

Now, let's trace what's going on with the program.

When you start the program with the command `java TestMinMaxFinder`, your computer starts a Java Virtual Machine (JVM). The JVM loads the necessary class definitions, including the `TestMinMaxFinder` and the `MinMaxFinder` class definitions. It starts executing the `main()` method of the `TestMinMaxFinder` class definition.

In the `main()` method, it first declares a local reference variable `finder`, creates an object of class `MinMaxFinder` and then assigns its reference to the local variable `finder`:

```
    // Create the MinMaxFinder object
    MinMaxFinder finder = new MinMaxFinder();
```

The scenario can be visualized in Figure 4.14.

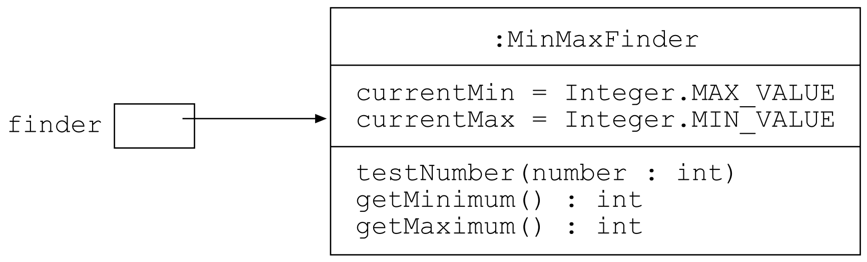


Figure 4.14 A MaxMinFinder object referred to by finder

Then, the `main()` method sends messages to the `MinMaxFinder` object:

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

The first time a message is sent, when the message `testNumber` with supplementary data `15` is sent to the `MinMaxFinder` object, the `MinMaxFinder` object performs its `testNumber()` behaviour and the parameter `number` is assigned a value of `15`.

In the method `testNumber()`, the first condition is tested by:

```
    // Compare the given number with current maximum
    if (number > currentMax) {
         // If the given number is greater than current maximum
         // assign the given number to the current maximum
    currentMax = number;
    }
```

The value stored by the variable (or parameter) `number` is `15`; the value stored by `currentMax`, an attribute of the `MinMaxFinder` object that stores the current maximum value so far, is `Integer.MIN_VALUE`. The value `15` is greater than `Integer.MIN_VALUE` and the comparison result is `true`. As a result, the block that follows the comparison parentheses is executed and the value stored by `number`, which is `15` here, is assigned to the attribute `currentMax` of the `MinMaxFinder` object. Then, the scenario becomes as shown in Figure 4.15.

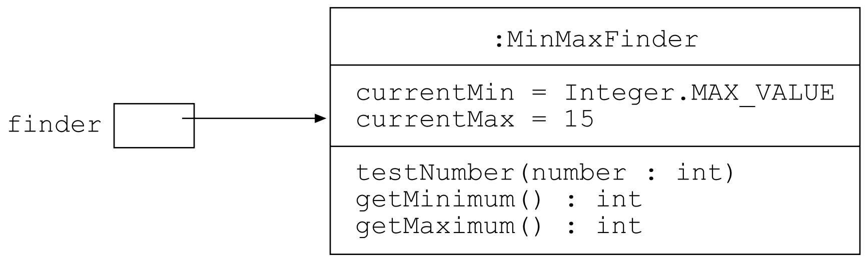


Figure 4.15 Assigning supplementary data to the MinMaxFinder object

Similarly, the value stored by the variable `number`, `15`, is compared with the object attribute `currentMin` and its value is `Integer.MAX_VALUE`.

```
// Compare the given number with the current minimum
if (number < currentMin) {
    // If the given number is less than the current mininum
    // assign the given number to the current minimum
    currentMin = number;
}
```

As the value `15` is less than the value represented by `Integer.MAX_VALUE`, the condition is `true` and the statement block is executed. As a result, the following statement is executed

```
    currentMin = number;
```

and the value stored by the variable `number` is assigned to the object attribute `currentMin`. Then, the scenario of the `MinMaxFinder` object becomes as shown in Figure 4.16.

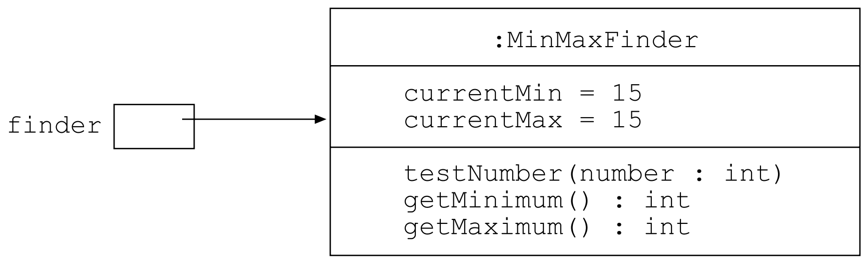


Figure 4.16 Assigning more supplementary data to the MinMaxFinder object

This is the end of the `testNumber()` method, and the `MinMaxFinder` object completes performing the `testNumber` behaviour. You can see that after the `MinMaxFinder` executes the `testNumber()` method for the first time, both the current minimum and maximum numbers are `15`.

The method that sends a message to another object is a *caller* method. We also consider that the caller method calls or invokes the methods of the other object by sending a message to it with the necessary supplementary data via parameters. In our example, the caller method is the `main()` method of `TestMinMaxFinder` and the method `testNumber()` of `MinMaxFinder` was called with a parameter `15`.

The flow of control returns to the `main()` method. The next statement to be executed is the one that follows the statement

```
    finder.testNumber(15);
```

It is

```
    finder.testNumber(20);
```

The above statement sends a message `testNumber` with supplementary data `20` to the `MinMaxFinder` object. The `MinMaxFinder` object starts performing the `testNumber` behaviour again.

This time in the `testNumber()` method, the value of the parameter `number` is assigned a value of `20`. At this time, the values of both object attributes `currentMax` and `currentMin` are `15`. For the two comparisons in the method

```
(number > currentMax)
```

and

```
(number < currentMin)
```

the following two comparisons are carried out:

```
(20 > 15) and (20 < 15)
```

The results of the comparisons are `true` and `false` respectively. Therefore, the statement block that follows the condition `(number > currentMax)` is executed, the value of the `number`, `20`, is assigned to the `currentMax` and the object attribute `currentMin` remains unchanged.

Therefore, after the `MinMaxFinder` object performs the `testNumber` behaviour for the second time, the scenario becomes as shown in Figure 4.17.

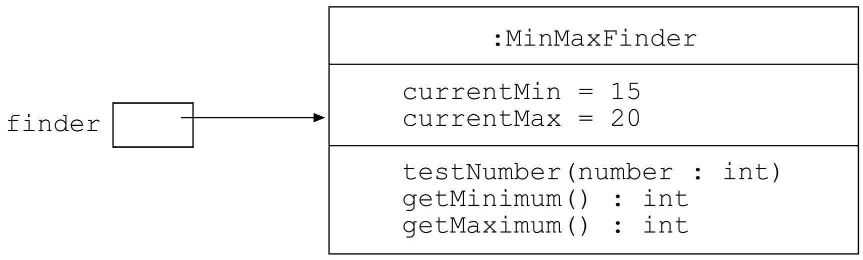


Figure 4.17 Assigning supplementary data to the MinMaxFinder object

Afterwards, the following statement in the `main()` method

```
    finder.testNumber(10);
```

sends the message `testNumber` with supplementary data `10` to the object. The parameter of the method `testNumber()` is assigned a value `10` and, for the two comparisons in the method

```
(number > currentMax)
```

and

```
(number < currentMin)
```

the following two comparisons are carried out:

```
(10 > 20) and (10 < 15)
```

The comparison results are `false` and `true` respectively. Therefore, the value `10` is assigned to the variable `currentMin` and the object becomes as shown in Figure 4.18.

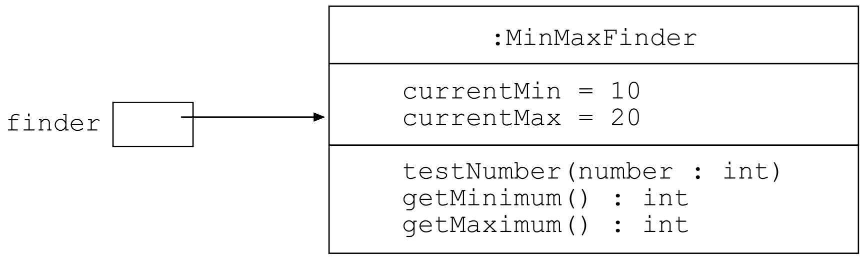


Figure 4.18 Assigning supplementary data to the MinMaxFinder object

The following statements in the `main()` method are then executed:

```
    int minNumber = finder.getMinimum();
```

A new variable named `minNumber` is declared, and the message `getMinimum` is sent to the object. The object returns its current minimum value, and the returned value is assigned to the variable `minNumber`. Therefore, after the above statement is executed, the value stored by the variable `minNumber` is `10`.

Similarly, the following statement in the `main()` method is executed next:

```
int maxNumber = finder.getMaximum();
```

Another new variable named `maxNumber` is declared, and the message `getMaximum` is sent to the object. The current maximum value of the object, which is `20`, is returned and is assigned to the variable `maxNumber`. As a result, the value stored by the variable `maxNumber` is `20`.

Finally, the following two statements in the `main()` method are executed:

```
System.out.println("Minimum number = " + minNumber);
System.out.println("Maximum number = " + maxNumber);
```

The first statement displays the value stored by the variable `minNumber` with prefix `"Minimum number = "` to the screen, that is:

```
Minimum number = 10
```

Whatever you want to display on the screen, you can use the following method to display it:

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

If there is more than one item, which could be a `String` object or any expression, to be displayed in the same line on the screen, use the add (+) operator to link them. If the message fills the current line, the message will continue on the next line. The next time the method `System.out.println()`is used, the message will be displayed on the next line on the screen.

Then, the latter statement executes, and the value stored by `maxNumber` is displayed with the prefix `"Maximum number = "`. Therefore, the following message is displayed on the screen:

```
Maximum number = 20
```

After all statements in the `main()` method are executed, the program terminates.

Example: morning, afternoon, or evening?

According to the hour passed to an instance of the class, the instance can determine whether the hour is in the morning, afternoon or evening. For example, we need an object of class named `GreetingChooser` . It has a method `assignGreeting()`. When the object receives a message `assignGreeting` with supplementary information `hour`, it returns a `String` reference to a `String` object with contents `"morning"`, `"afternoon"` or `"evening"`. Therefore, the class `GreetingChooser` is defined as shown in Figure 4.19.

```
// The class definition for a greeting chooser
public class GreetingChooser {

    // Determine the greeting based on the given hour
    public String assignGreeting(int hour) {
        // A local variable for storing the result temporarily
        String greeting = "";

        // Check whether the hour is in the morning
        if (hour >= 0) {
            greeting = "morning";
        }

        // Check whether the hour is in the afternoon
        if (hour >= 12) {
            greeting = "afternoon";
        }

        // Check whether the hour is in the evening
        if (hour >= 18) {
            greeting = "evening";
        }

        // Return the result to the caller
        return greeting;
    }
}
```

Figure 4.19 GreetingChooser.java

Whenever the object of the class `GreetingChooser` receives a message `assignGreeting` with the supplementary data `hour`, the method `assignGreeting()` is executed and the parameter `hour` has a value that comes with the message. The following explains the method statement by statement:

```
String greeting = "";
```

A local variable named `greeting` is declared (or created). It initially refers to a `String` object with empty contents (a `String` without any characters, called a null string).

```
// Check whether the hour is in the morning
if (hour >= 0) {
    greeting = "morning";
}
```

The value stored by the variable `hour` is checked if it is greater than or equal to zero. If the comparison result is `true`, the statement block enclosed within the pair of curly brackets is executed. That is, the local variable `greeting` is changed to refer to a `String` object with the contents `"morning"`.

This is not the end of the method — the computer keeps on executing the following statements:

```
    // Check whether the hour is in the afternoon
    if (hour >= 12) {
        greeting = "afternoon";
    }
```

Similarly, if the value stored by variable `hour` is greater than or equal to `12`, the comparison result is `true` and the statement block is executed so that the variable `greeting` is changed to refer to a `String` object with the contents `"afternoon"`.

```
    // Check whether the hour is in the evening
    if (hour >= 18) {
        greeting = "evening";
    }
```

The third `if` statement checks whether the value of the variable `hour` is greater than or equal to `18`. It changes the variable `greeting` to refer to a `String` object with the content `"evening"` if the comparison result is `true`.

```
    // Return the result to the caller
    return greeting;
```

Finally, the `String` object that is referred to by the local variable `greeting` is returned.

To test the class `GreetingChooser`, the following program `TestGreeting` is written (some new features have been used, and explanations follow) as shown in Figure 4.20.

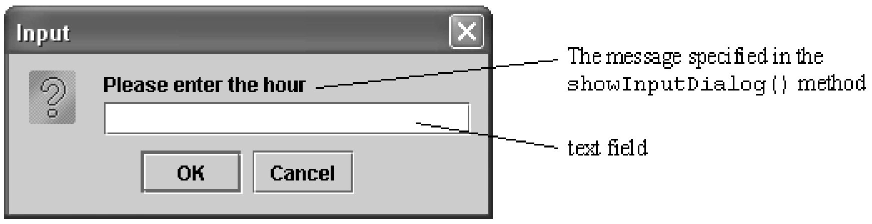
```
// Import statement for resolving the class JOptionPane
import javax.swing.JOptionPane;

// The class definition of TestGreeting for setting up the
// environment and test the GreetingChooser object
public class TestGreeting {

    // The main executive method
    public static void main(String args[]) {
        // Create the GreetingChooser object and use variable
chooser
        // to refer to it
        GreetingChooser chooser = new GreetingChooser();

        // Show a dialog for getting the hour from user
        // The obtained hour is returned as a String object
        String inputHour =
            JOptionPane.showInputDialog("Please enter the hour");

        // Obtain the integer value from the input String object
        int hour = Integer.parseInt(inputHour);

        // Get the greeting from the GreetingChooser object
        String greeting = chooser.assignGreeting(hour);

        // Determine the message to be shown
        String output = "Good " + greeting;

        // Show the greeting to the user with a dialog
        JOptionPane.showMessageDialog(null, output);

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

Figure 4.20 TestGreeting.java

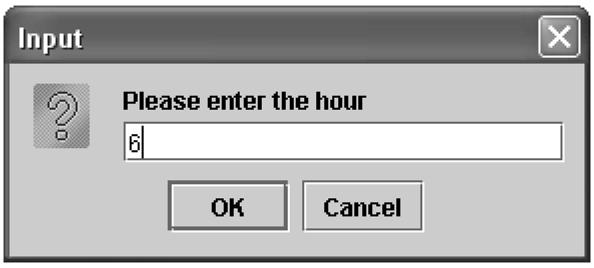
After compilation, execute the above `GreetingTester` program by entering

```
java TestGreeting
```

at the Command Prompt. It displays the following dialog first,



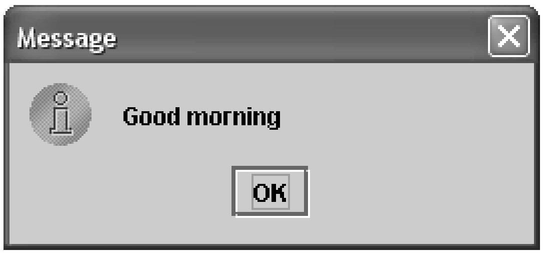
to prompt the user for the hour. Enter a value, say `"6"`, in the text field.


Input

Please enter the hour

6

Then, press <Enter> or click <OK>. The program displays the following dialog:


Message

Good morning

You can try the program with other values; the program will display the greeting accordingly.

Now, let's trace the program to see how it works.

First of all, you compile the source codes and enter

```
java TestGreeting
```

to start the program. Then, the `main()` method of the class `TestGreeting` is executed.

The following statement is executed to create a `GreetingChooser` object:

```
GreetingChooser chooser = new GreetingChooser();
```

The reference of the newly created object is assigned to the variable `chooser` so that the object can be used via this variable.

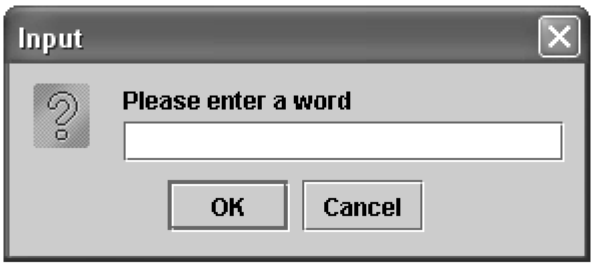
We can instruct the program to accept user input. One way to do so is by using the following method:

```
javax.swing.JOptionPane.showInputDialog(String)
```

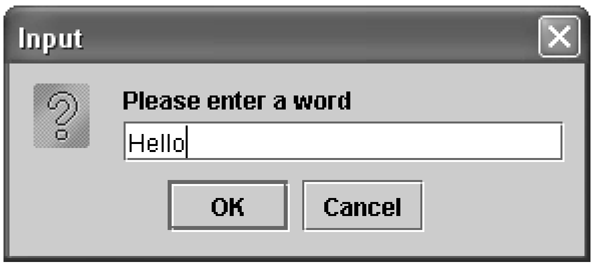
The method `showInputDialog()` is a class method of the class `javax.swing.JOptionPane`. (If you need to refresh your memory concerning packages, please refer to *Unit 3*.) It accepts a `String` object as the parameter, which is the message shown to the user in the dialog. For example, the following statement shows the dialog with the message `"Please enter a word"`

```
javax.swing.JOptionPane.showInputDialog("Please enter a word");
```

and the program displays the following dialog:



The program will wait for the user's input. The user can now type anything in the text field. When the user feels comfortable with the entry, such as when the word `"Hello"` is entered, he or she can press <Enter> or use the mouse pointer to click <OK>.



The method will return the entry in the text field as a `String` object. Therefore, we usually use a variable to store the returned `String` object. For example:

```
String input = javax.swing.JOptionPane.showInputDialog(
    "Please enter a word");
```

The returned `String` object will be assigned to the variable `input`. Therefore, if the above statement is executed and the user entry is `"Hello"`, the variable `input` is referring to a `String` object with content `"Hello"` as shown in Figure 4.21.

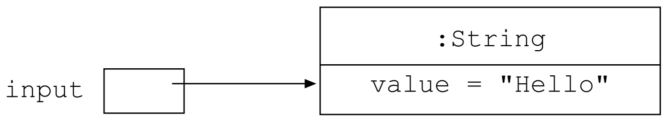


Figure 4.21 State of a string object

In order not to use the fully qualified class name, `javax.swing.JOptionPane`, you can use an `import` statement at the beginning of the source code file

```
import javax.swing.*;
```

or

```
import javax.swing.JOptionPane;
```

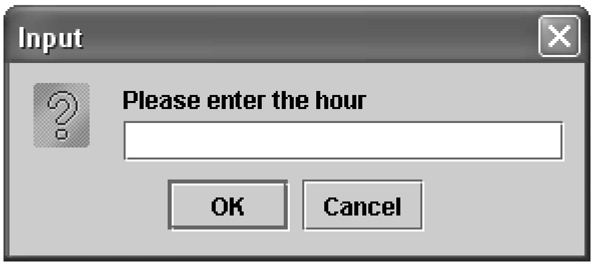
so you can use the following statement instead:

```
String input = JOptionPane.showInputDialog("Please enter a word");
```

In our example program, the following statement prompts the user for a value of an hour in a dialog with the message `"Please enter the hour"`

```
String inputHour =
    JOptionPane.showInputDialog("Please enter the hour");
```

The following dialog appears:



If the user enters `"6"` in the text field and presses <Enter> or clicks <OK>, a `String` object with content `"6"` is returned and is assigned to the variable `inputHour`.

The `assignGreeting()` method can only accept an integer value as a parameter, so it is necessary to derive an equivalent integer value from the `String` object referred to by the variable `inputHour`. In the Java API library, the following method helps us do this:

```
Integer.parseInt(String)
```

The returned value is a value of type `int` and hence the method is usually used in the following format:

```
int number = Integer.parseInt(String);
```

The statement in our example program is:

```
int hour = Integer.parseInt(inputHour);
```

The equivalent integer value is derived from the `String` object referred to by the variable `inputHour`, and the integer value is assigned to a new variable named `hour`. If the content of the `String` object is `"6"`, an integer value `6` is derived and assigned to the variable `hour`.

You have now obtained the input from the user and have it in integer type. It is now possible to use the `GreetingChooser` object to get the proper greeting message.

```
String greeting = chooser.assignGreeting(hour);
```

The message `assignGreeting` with supplementary data (`hour`) is presented to the `GreetingChooser` object that is referred to by the variable `chooser`.

When the `GreetingChooser` object performs its `assignGreeting` behaviour, it gets an extra data item from the parameter `hour` because the method declaration is:

```
public String assignGreeting(int hour) {
......
}
```

The value assigned to the parameter `hour` is `6`. Note that there are two variables with the same name — `hour`. One is declared in the `main()` method of the class `TestGreeting` and one is declared as a parameter of the method `assignGreeting()` of class `GreetingChooser`. Although their names are the same, they are referring to two different memory locations in the memory.

In the `assignGreeting()` method, a local variable `greeting` of type `String` is declared and is assigned a `String` object with empty content:

```
String greeting = "";
```

The value stored in the parameter `hour` is compared with `0`, `12` and `18` respectively:

```
// Check whether the hour is in the morning
if (hour >= 0) {
    greeting = "morning";
}

// Check whether the hour is in the afternoon
if (hour >= 12) {
    greeting = "afternoon";
    }

// Check whether the hour is in the evening
if (hour >= 18) {
    greeting = "evening";
}
```

In our program example in which `hour` is given the value `6`, only the first comparison `hour >= 0` is `true` and the statement in its block is executed. Therefore, the local variable `greeting` is updated to refer to the `String` object with content `"morning"` as in Figure 4.22.

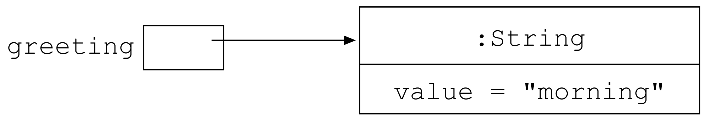


Figure 4.22 State of the variable greeting after program segment execution

Finally, the following statement is executed:

```
// Return the result to the caller
return greeting;
```

Therefore, the reference to the `String` object with the content `"morning"` is returned to the position in which the message `assignGreeting` was sent to the `GreetingChooser` object; that is, where the method `assignGreeting()` of the

`GreetingChooser` object is invoked. In our example program, the following statement is in the `main()` method:

```
    String greeting = chooser.assignGreeting(hour);
```

The reference to the `String` object with `"morning"` as content is assigned to the local variable `greeting` of the `main()` method. The variable name `greeting` is the same as the local variable in the method `assignGreeting()` of class `GreetingChooser`, but they are referring to different memory locations. This is similar to the local variable `hour` in the `main()` method of the class `TestGreeting` and the parameter `hour` of the `assignGreeting()` of the class `GreetingChooser`.

Now, the local variable `greeting` of the `main()` method is referring to the `String` object with content `"morning"` as shown in Figure 4.23.

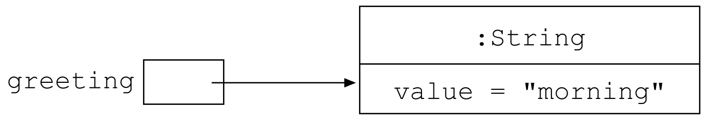


Figure 4.23 State of the variable greeting

Then, the following statement is executed to determine the output message:

```
    String output = "Good " + greeting;
```

The two `String` objects with contents `"Good"` and `"morning"` (referred by the variable `greeting`) are concatenated. A new `String` object with content `"Good morning"` is created and assigned to the variable `output` as shown in Figure 4.24.

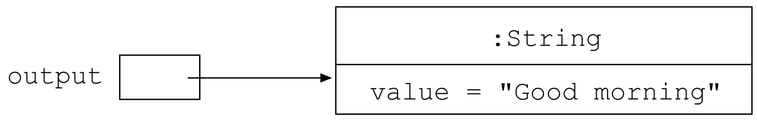


Figure 4.24 State of the variable output

The output message is ready and it is now possible to display it. Up to now, we have only displayed the message to the Command Prompt using the following statement:

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

As our program uses a dialog for getting user input, it is preferable to show the output message in a dialog as well. To do so, you can use the following method:

```
javax.swing.JOptionPane.showMessageDialog(Component, String)
```

The method `showMessageDialog()` is a class method of the class `javax.swing.JOptionPane`. If a suitable `import` statement is used, the method can be written as:

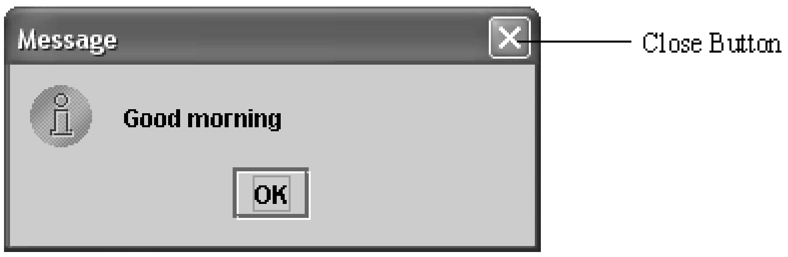
```
JOptionPane.showMessageDialog(Component, String)
```

Two parameters are required to use the method. For now, you can simply use `null` for the first parameter. We elaborate on its usage in *Unit 9*. We usually use a `String` object as the second parameter.

In the class `TestGreeting`, the following statement is used to display the message determined:

```
    // Show the greeting to the user with a dialog
    JOptionPane.showMessageDialog(null, output);
```

As the `String` object referred to by the variable `output` is `"Good morning"`, the following dialog is shown to the user:



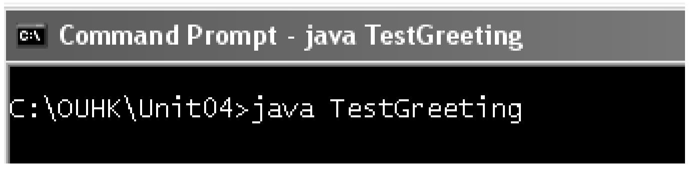
The output message is shown. You can now close the dialog by simply pressing <Enter> or the Close button at the top right corner.

Since the program uses some dialogs, it is necessary to terminate the program explicitly by using the following statement:

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

The method accepts one parameter of type `int` as the program termination status. As a usual practice, a non-zero value indicates an abnormal program termination. Therefore, you can use `0` as the status if the program is terminated properly.

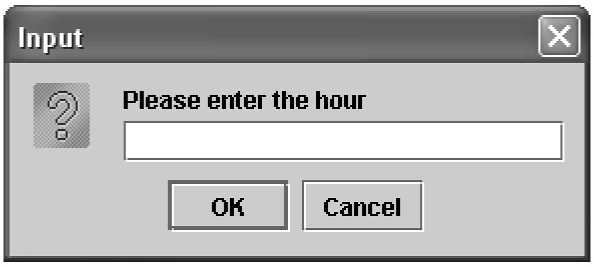
If your program uses the dialogs without the `System.exit()` method as the last statement, the program will not terminate even though you have closed all the dialogs.



```
C:\OUHK\Unit04>java TestGreeting
```

Then, you need to force the termination of the Java Virtual Machine by typing <Ctrl-C> in the Command Prompt.

Let's execute the `TestGreeting` program with another value for `hour`, says `15`. Execute the `TestGreeting` again. The dialog that prompts you for the `hour` appears.



Enter `"15"` and complete the dialog by either clicking <OK> or pressing <Enter>. In the `main()` method, a numeric value `15` is derived. A message `assignGreeting` with supplementary data (`hour`) is sent to the `GreetingChooser` object that is referred to by the variable `chooser`.

```
    // Obtain the integer value from the input String object
    int hour = Integer.parseInt(inputHour);

    // Get the greeting from the GreetingChooser object
    String greeting = chooser.assignGreeting(hour);
```

When the `GreetingChooser` object performs its behaviour `assignGreeting`, or in other words, the object executes its method `assignGreeting()`, the parameter `hour` is set to a value of `15`.

In the `assignGreeting()` method, a local variable `greeting` is declared and is initialized so that it is referring to a `String` object with empty content. Then, the first `if` statement is executed with condition `hour >= 0`, the condition result is evaluated to be `true`, and the following statement is executed that updates the local variable `greeting` to refer to a `String` object with content `"morning"`:

```
    greeting = "morning";
```

Afterwards, the second `if` statement is executed, and the condition `hour >= 12` is evaluated to be `true.` The following statement

```
    greeting = "afternoon";
```

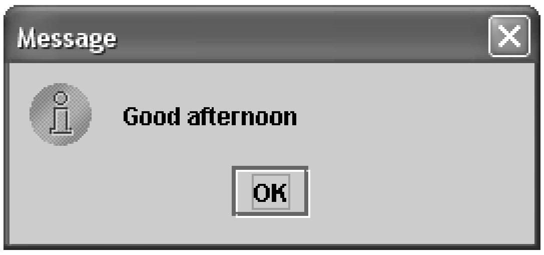
is executed that updates the local variable `greeting` to refer to the `String` object with content `"afternoon"`.

Then, the third `if` statement is executed, but the condition `hour >= 18` is evaluated to be `false` and the local variable greeting remains unchanged. Last, the `return` statement returns the reference to the `String` object with content `"afternoon"`.

Therefore, in the `main()` method, the following statements determine the complete greeting message and show the dialog with the complete greeting message.

```
    // Determine the message to be shown
    String output = "Good " + greeting;

    // Show the greeting to the user with a dialog
    JOptionPane.showMessageDialog(null, output);
```

As a result, the following dialog appears on the screen.



Good afternoon

You can start the program again to try other values of `hour` to see whether the program works as expected.

Self-test 4.3

Execute the `TestGreeting` program and enter `"-1"` as the input hour. What is the output of the program? Why?

`if` statements with `else` clauses

While you are writing a program, you will often encounter situations in which the program has to perform a task if a condition is `true` and perform another task if it is `false`. For example, if you have a variable named `number` and would like to display a message indicating whether it is positive or not, you can write the program segment as:

```
if (number > 0) {
    System.out.println("Number is positive");
}
if (number <= 0) {
    System.out.println("Number is non-positive");
}
```

The above program segment, of course, works properly. However, two conditions have been tested one-by-one. If the result of the first condition is `false`, the result of the second one *must* be `true` but the program will still test the second condition. Therefore, it is better if the second testing can be eliminated. Furthermore, consider the following program segment

```
if (number > 0) {
    System.out.println("Number is positive");
}
if (number < 0) {
    System.out.println("Number is non-positive");
}
```

The programmer who wrote the segment misinterprets non-positive numbers as negative, which is not correct. The value zero should be treated as non-positive as well. Such programming style is error-prone, redundant and tedious.

The Java programming language provides a better way to handle the above situation by adding the `else` structure. The above program segment can be written as:

```
if (number > 0) {
    System.out.println("Number is positive");
}
else {
    System.out.println("Number is non-positive");
}
```

The general syntax of using such `if/else` structures is:

```
if (condition) {
    statement(s)-1 // statement(s) to be executed if the condition is true
}
else {
    statement(s)-2 // statement(s) to be executed if the condition is false
}
```

If the *condition* is evaluated to be `true`, the block that follows the condition parentheses, the *if part*, is executed. Otherwise, the statement block that follows the `else` keyword, the *else part*, is executed. Therefore, either one of the statement blocks *must* be executed. You can visualize the above `if/else` structure in Figure 4.25.

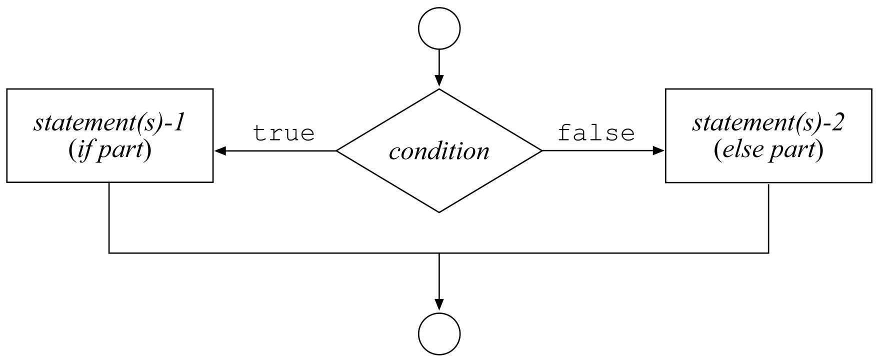


Figure 4.25 Visualization of an if/else statement

If a statement block constitutes a single statement, the pair of curly brackets is optional. For example, the program segment we discussed before can be written as:

```
if (number > 0)
    System.out.println("Number is positive");
else
    System.out.println("Number is non-positive");
```

Although it is shorter and seems to be simpler, it may be error-prone, especially when you add statements to either block of the `if/else` structure. Therefore, it is highly recommended you use the curly brackets even if there is only one statement in the block.

Example: solving quadratic equations

We discussed the quadratic equation in *Unit 1*. For a quadratic equation in the following format

$$ax^2 + bx + c = 0$$

if the coefficients a ($\neq 0$), b and c are given particular values, the possible values of x satisfying the equation (so that the expression on the left-hand side is zero) are given by the following formulas:

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \text{ and } x = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

In mathematics, the two values obtained from the above two formulas are usually known as the roots of the quadratic equation. Therefore, a quadratic equation has real solutions if

$$b^2 - 4ac \geq 0$$

and the expression $b^2 - 4ac$ is called the *determinant* of the quadratic equation. Based on that understanding of quadratic equations, a class `QuadraticEquation` is written and shown in Figure 4.26.

```
// Class definition of QuadraticEquation for a quadratic equation
// with format ax^2 + bx + c = 0
public class QuadraticEquation {
    // The three coefficients of the equation
    private double coeffA = 0.0,
                   coeffB = 0.0,
                   coeffC = 0.0;

    // Set the coefficient a
    public void setCoeffA(double a) {
        coeffA = a;
    }

    // Set the coefficient b
    public void setCoeffB(double b) {
        coeffB = b;
    }

    // Set the coefficient c
    public void setCoeffC(double c) {
        coeffC = c;
    }

    // Get the determinant of the quadratic equation
    public double findDeterminant() {
        return coeffB * coeffB - 4.0 * coeffA * coeffC;
    }

    // Get the first real root
    public double findFirstRealRoot() {
        return (-coeffB - Math.sqrt(coeffB * coeffB -
            4.0 * coeffA * coeffC)) / (2.0 * coeffA);
    }

    // Get the second real root
    public double findSecondRealRoot() {
        return (-coeffB + Math.sqrt(coeffB * coeffB -
            4.0 * coeffA * coeffC)) / (2.0 * coeffA);
    }
}
```

Figure 4.26 QuadraticEquation.java

The class `QuadraticEquation` defines three attributes, `coeffA`, `coeffB` and `coeffC`, representing the three coefficients *a*, *b* and *c* of a quadratic equation respectively. In order to set these three attributes, three methods, `setCoeffA()`, `setCoeffB()` and `setCoeffC()` are provided. Their functions are to assign the value of the method parameter to the corresponding attribute of the object. For example, the method `setCoeffA()` is written as:

```
// Set the coefficient a
public void setCoeffA(double a) {
    coeffA = a;
}
```

Whenever the method `setCoeffA()` is called, a message `setCoeffA` is sent to the object, a value of type `double` is provided and is temporarily stored in parameter `a`. In the method, only one statement stores the value in parameter `a` to the object attribute `coeffA`.

Another method `findDeterminant()` is written so that the determinant is calculated and returned.

```
// Get the determinant of the quadratic equation
public double findDeterminant() {
    return coeffB * coeffB - 4.0 * coeffA * coeffC;
}
```

The remaining two methods provided in the class definition are `findFirstRealRoot()` and `findSecondRealRoot()` that are used to calculate and return the two roots respectively.

```
// Get the first real root
public double findFirstRealRoot() {
    return (-coeffB - Math.sqrt(coeffB * coeffB -
        4.0 * coeffA * coeffC)) / (2.0 * coeffA);
}

// Get the second real root
public double findSecondRealRoot() {
    return (-coeffB + Math.sqrt(coeffB * coeffB -
        4.0 * coeffA * coeffC)) / (2.0 * coeffA);
}
```

The only difference between the two methods is that the values returned correspond to the following two equations respectively:

$$x = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \text{ and } x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

In these two methods, the expression is a little complicated. For example, the expression in the `findFirstRealRoot()` method is:

```
(-coeffB - Math.sqrt(coeffB * coeffB - 4.0 * coeffA * coeffC)) / (2.0 *
coeffA)
```

The `Math.sqrt(double)` method is provided in the standard Java API library for determining the square root of the given number of type `double`.

To illustrate how the above expression is evaluated, suppose that the values stored by the variables `coeffA`, `coeffB` and `coeffC` are `2.0`, `6.0` and `4.0` respectively. Then, the expression can be treated as:

```
(-(6.0) – Math.sqrt(6.0 * 6.0 – 4.0 * 2.0 * 4.0)) / (2.0 * 2.0)
```

In the above expression, an extra pair of parentheses is added to the first occurrence of `coeffB` to highlight that a unary minus operator is preceding it.

First of all, both sides of the division operator are enclosed in parentheses. As the division operator is left associative, the expression enclosed in the parentheses on its left-hand side is evaluated first. That is, the following expression is evaluated:

```
-(6.0) – Math.sqrt(6.0 * 6.0 – 4.0 * 2.0 * 4.0)
```

The first minus character `-` is the unary minus operator. Its precedence is higher than that of the subtraction operator (the second minus character in the expression). It negates the expression or variable that follows it. Therefore the expression becomes:

```
-6.0 – Math.sqrt(6.0 * 6.0 – 4.0 * 2.0 * 4.0)
```

Then, the method `Math.sqrt()` is about to be called, and the value to be presented to the method is an expression. The enclosed expression is evaluated as usual:

```
   6.0 * 6.0 – 4.0 * 2.0 * 4.0
=  36.0 – 32.0
=  4.0
```

The result `4.0` is obtained and the expression becomes:

```
-6.0 – Math.sqrt(4.0)
```

The square root of `4.0` is `2.0` and the expression in the left parentheses is further evaluated to be:

```
-6.0 – 2.0
```

The overall result of the left parentheses is `–8.0`.

The original expression becomes:

```
   -8.0 / (2.0 * 2.0)
=  -8.0 / (4.0)
=  -2.0
```

Therefore, the root determined by the `findFirstRealRoot()` method is `-2.0` and that value will be returned to the place that the `findFirstRealRoot()` method is called.

Please note that the parentheses enclosing `2.0 * coeffA` are mandatory. If such parentheses are missing, that is,

```
(-coeffB - Math.sqrt(coeffB * coeffB - 4.0 * coeffA * coeffC)) / 2.0 *
coeffA
```

the expression denotes the following calculation:

$$x = \left(\frac{-b - \sqrt{b^2 - 4ac}}{2} \right) a$$

If you insist on not using the parentheses, an alternate expression is:

```
(-coeffB - Math.sqrt(coeffB * coeffB - 4.0 * coeffA * coeffC)) / 2.0 /
coeffA
```

The `findSecondRealRoot()` method evaluates another similar expression. With the coefficient values mentioned above, the result obtained by the `findSecondRealRoot()` method is `-1.0`.

The definition of the class `QuadraticEquation` uses no `if/else` structure, but the program `EquationSolver` that is used to test the `QuadraticEquation` class does. This is shown in Figure 4.27.

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

// The class definition of EquationSolver for setting up the
// environment and testing the QuadraticEquation object
public class EquationSolver {

    // The main executive method
    public static void main(String args[]) {
        // Create a QuadraticEquation object and use variable equation
        // to refer to it
        QuadraticEquation equation = new QuadraticEquation();

        // Get the coefficient A and set it to the quadratic equation
        String inputA =
            JOptionPane.showInputDialog("Please enter coefficient A");
        double coeffA = Double.parseDouble(inputA);
        equation.setCoeffA(coeffA);

        // Get the coefficient B and set it to the quadratic equation
        String inputB =
            JOptionPane.showInputDialog("Please enter coefficient B");
        double coeffB = Double.parseDouble(inputB);
        equation.setCoeffB(coeffB);

        // Get the coefficient C and set it to the quadratic equation
        String inputC =
            JOptionPane.showInputDialog("Please enter coefficient C");
        double coeffC = Double.parseDouble(inputC);
        equation.setCoeffC(coeffC);

        // Check if the quadratic equation has real roots based on the
        // determinant, and prepare the message to be shown to the user
        String message;
        if (equation.findDeterminant() >= 0.0) {
            double firstRoot = equation.findFirstRealRoot();
            double secondRoot = equation.findSecondRealRoot();
            message = "The roots of the quadratic equation are:" +
                "\nFirst root = " + firstRoot +
                "\nSecond root = " + secondRoot;
        }
        else {
            message = "No real roots";
        }

        // Show the result to the user with a dialog
        JOptionPane.showMessageDialog(null, message);

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

Figure 4.27 EquationSolver.java

When the `EquationSolver` is compiled and executed, its `main()` method creates the `QuadraticEquation` object first. Then the reference to the newly created `QuadraticEquation` object is assigned to the local variable `equation`.

The definition of class `QuadraticEquation` has three attributes. They are initially assigned a value of `0.0`:

```
// The three coefficients of the equation
private double coeffA = 0.0,
               coeffB = 0.0,
               coeffC = 0.0;
```

Therefore, after the first statement is executed, the following scenario is created, as shown in Figure 4.28.

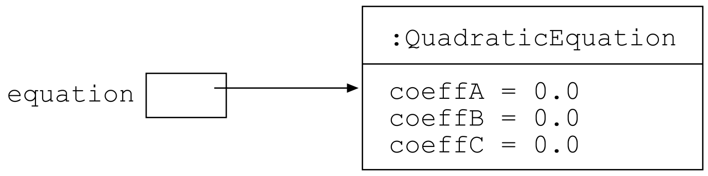
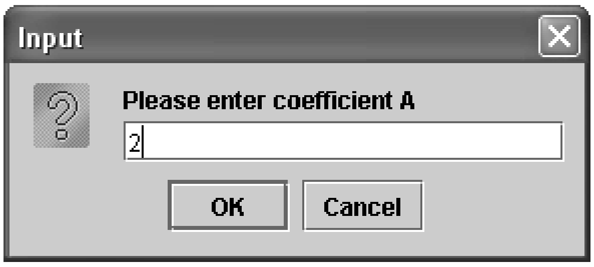


Figure 4.28 Assigning initial values to the quadratic coefficients

Then, the following program segment is executed:

```
// Get the coefficient A and set it to the quadratic equation
String inputA =
    JOptionPane.showInputDialog("Please enter coefficient A");
double coeffA = Double.parseDouble(inputA);
equation.setCoeffA(coeffA);
```

First, a dialog with the message `"Please enter coefficient A"` appears for the user to enter coefficient *a*, and the inputted entry is stored in the variable `inputA` of type `String`. For example, a value of `"2"` is provided.



A `String` object with content `"2"` is assigned to the variable `inputA`, as shown in Figure 4.29.

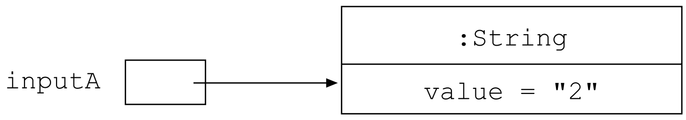


Figure 4.29 Assigning a value to a variable

To obtain an equivalent numeric value of `double` type from the variable `inputA`, the following statement is executed:

```
    double coeffA = Double.parseDouble(inputA);
```

Like the method `Integer.parseInt()`, the above method `Double.parseDouble()` can be used to derive a numeric value from a `String` object. The only difference is the derived numeric value is of type `double`. Therefore, the above statement derives a `double` value from the user input and assigns it to a local variable `coeffA`.

Then, the value of coefficient *a* is ready to be sent to the `QuadraticEquation` object by the following statement:

```
    equation.setCoeffA(coeffA);
```

After the above program segment is executed, the `QuadraticEquation` object becomes as shown in Figure 4.30.

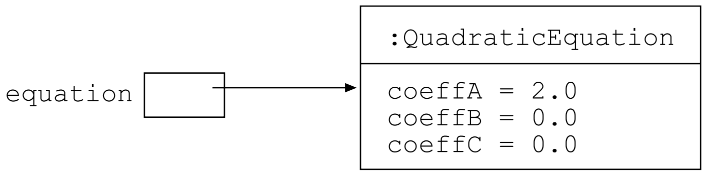
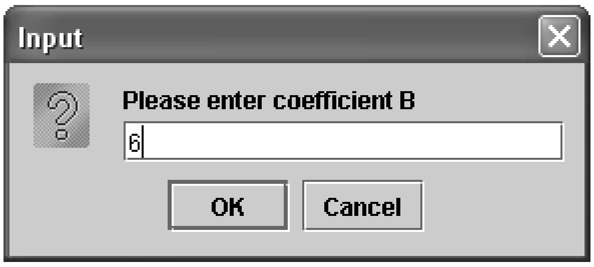


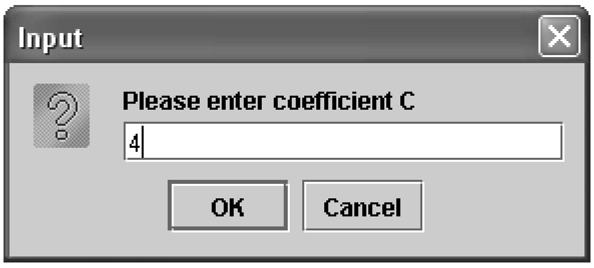
Figure 4.30 The state of the QuadraticEquation object

The subsequent program segments perform similarly for coefficients *b* and *c*:

```
    // Get the coefficient B and set it to the quadratic equation
    String inputB =
        JOptionPane.showInputDialog("Please enter coefficient B");
    double coeffB = Double.parseDouble(inputB);
    equation.setCoeffB(coeffB);

    // Get the coefficient C and set it to the quadratic equation
    String inputC =
        JOptionPane.showInputDialog("Please enter coefficient C");
    double coeffC = Double.parseDouble(inputC);
    equation.setCoeffC(coeffC);
```

Two dialogs appear prompting the user for coefficients *b* and *c*. Suppose that the values `"6"` and `"4"` are provided:





Two numeric values `6.0` and `4.0` of type `double` are derived and are assigned to the `QuadraticEquation` object. The object becomes as shown in Figure 4.31.

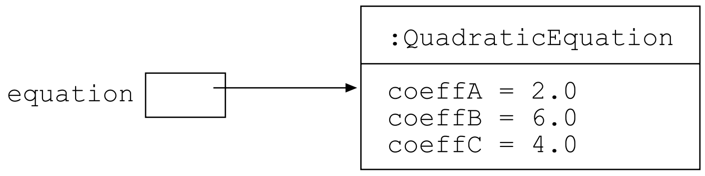


Figure 4.31 The updated QuadraticEquation object

The `QuadraticEquation` is properly set according to the user inputs. It can now determine whether the corresponding quadratic equation

$$2x^2 + 6x + 4 = 0$$

has real roots. Therefore, the following program segment in the `main()` method is used to determine it:

```
// Check if the quadratic equation has real roots based on the
// determinant, and prepare the message to be shown to the user
String message;
if (equation.findDeterminant() >= 0.0) {
    double firstRoot = equation.findFirstRealRoot();
    double secondRoot = equation.findSecondRealRoot();
    message = "The roots of the quadratic equation are:" +
              "\nFirst root = " + firstRoot +
              "\nSecond root = " + secondRoot;
}
else {
    message = "No real roots";
}
```

First, a variable `message` of type `String` is declared for storing the message to be shown to the user.

Then the following `if/else` statement is used to determine whether the quadratic equation has real roots by inquiring about the equation's determinant:

```
if (equation.findDeterminant() >= 0.0) {
......
}
else {
......
}
```

The method `findDeterminant()` is called to get its determinant. With the set coefficients, the determinant calculated and returned is `2.0`. The condition `2.0 >= 0.0` is `true` and the statement block that follows the condition is executed:

```
     double firstRoot = equation.findFirstRealRoot();
     double secondRoot = equation.findSecondRealRoot();
     message = "The roots of the quadratic equation are:" +
               "\nFirst root = " + firstRoot +
               "\nSecond root = " + secondRoot;
```

The first two statements declare two new local variables named `firstRoot` and `secondRoot` of type `double`. The `findFirstRealRoot()` and `findSecondRealRoot()` methods of the `QuadraticEquation` object are called one-by-one to get the roots. The returned values, `–2.0` and `–1.0`, are assigned to the two local variables `firstRoot` and `secondRoot` respectively.

The third statement is long. The right-hand side of the assignment operator `=` is a `String` concatenation expression. The resultant `String` object is assigned to the local variable `message`.

First, the two `String` objects with contents `"The roots of the quadratic equation are:"` and `"\nFirst root = "` are concatenated to give `"The roots of the quadratic equation are:\n First root = "`. You should remember that `\n` is a special character known as *newline* character. Although there are two characters in the program, it represents a single character, which means the characters that follow it should be printed on the next line. With the current `String` contents, the message will be displayed as:

```
The roots of the quadratic equation are:
First root =
```

Then, the `String` with contents `"The roots of the quadratic equation are:\n First root = "` is further concatenated using the value stored by the variable `firstRoot`. The value of the variable `firstRoot` is `–2.0` and hence `"-2.0"` will be appended to the `String` object and gives `"The roots of the quadratic equation are:\n First root = -2.0"`.

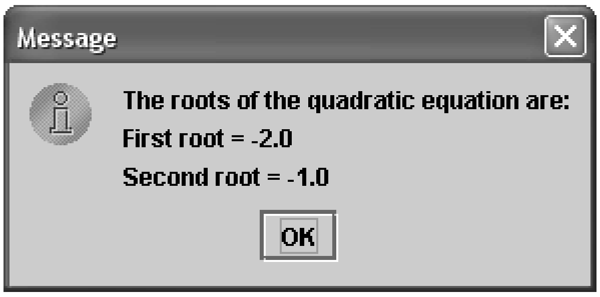
Last, another `String` object `"\nSecond root = "` (the value stored by the local variable `secondRoot`) is appended to the above `String`. The resultant `String` obtained from the expression is therefore `"The roots of the quadratic equation are:\n First root = -2.0\nSecond root = -1.0"`. When such a `String` object is displayed, the message is shown in the following format:

```
The roots of the quadratic equation are:
First root = -2.0
Second root = -1.0
```

The message to be shown is ready. It is now possible to show it to the user by the following statement:

```
    // Show the result to the user with a dialog
    JOptionPane.showMessageDialog(null, message);
```

The corresponding dialog will appear:

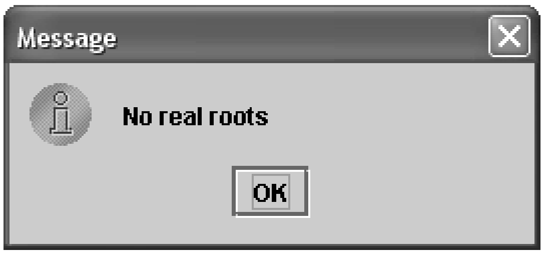


The roots of the quadratic equation are:
First root = -2.0
Second root = -1.0

You can try other combinations of coefficients. If there is no real solution for that set of coefficients, the condition in the `main()` method will be evaluated to be `false`. Then, the program block that follows the keyword `else` will be executed. That is:

```
    message = "No real roots";
```

The following dialog will appear, telling you about this:



No real roots

The expressions in the methods `findFirstRealRoot()` and `findSecondRealRoot()` include the same expression for the determinant. As the `QuadraticEquation` class has the method `findDeterminant()`, the expressions in the two methods for getting real roots can be modified to use the `findDeterminant()` method, such as:

```
// Get the first real root
public double findFirstRealRoot() {
    return (-coeffB – Math.sqrt(findDeterminant())) / (2.0 * coeffA);
}

// Get the second real root
public double findSecondRealRoot() {
    return (-coeffB + Math.sqrt(findDeterminant())) / (2.0 * coeffA);
}
```

The two expressions involve a method call to the `findDeterminant()` method of the `QuadraticEquation` object itself for getting the determinant. This is a preferable way to implement the class because the class is simpler; it is always preferable to reduce duplicated code. However, the expressions used in the current implementation use the complete formulas to highlight the relationships between the methods and the formulas.

Example: determining the grades

Assume that you need an object to determine the grade from a given mark. The ranges for the grades are shown in Table 4.3.

Table 4.3 Grade ranges

Mark	Grade
90 to 100	A
80 to 89	B
70 to 79	C
60 to 69	D
50 to 59	E
0 to 49	F

You can write the following class definition to derive the grade from a given mark, as in Figure 4.32.

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

    // The method for getting the grade based on the given mark
    public String assignGrade(int mark) {
        // The grade is first assumed to be F
        String grade = "F";

        // Set the criteria for the grades one by one
        if (mark >= 50) {
            grade = "E";
        }
        if (mark >= 60) {
            grade = "D";
        }
        if (mark >= 70) {
            grade = "C";
        }
        if (mark >= 80) {
            grade = "B";
        }
        if (mark >= 90) {
            grade = "A";
        }

        // Return the determined grade
        return grade;
    }
}
```

Figure 4.32 GradeConverter.java

The `assignGrade()` method of the `GradeConverter` class requires a parameter of type `int`. Whenever the method is called, a numeric value of type `int` is assigned to the parameter `mark`, as specified by the method declaration:

```
    public String assignGrade(int mark) {
    .. .. ..
    }
```

The method will return a `String` object with the content of the grade determined.

In the method, a local variable named `grade` of type `String` is declared and initialized to be `"F"`:

```
    // The grade is first assumed to be F
    String grade = "F";
```

Then, the value that is stored by the parameter `mark` is involved in several comparisons:

```
// Set the criteria for the grades one by one
if (mark >= 50) {
    grade = "E";
}
if (mark >= 60) {
    grade = "D";
}
if (mark >= 70) {
    grade = "C";
}
if (mark >= 80) {
    grade = "B";
}
if (mark >= 90) {
    grade = "A";
}
```

The `grade` is changed to `"E"` if the content of the variable `mark` is greater than or equal to `50`. Afterwards, the `mark` is tested against `60`, `70`, `80` and `90` sequentially and the grade is assigned accordingly.

For example, if the value stored by the parameter is `75`, the first comparison `mark >= 50` is `true` and the local variable `grade` is assigned an `"E"`. Next, `mark` is tested against `60` in the comparison `mark >= 60` is also `true` and `grade` is assigned a `"D"`. Similarly, the comparison `mark >= 70` is also `true` and the local variable `grade` is updated to `"C"`. The results of the subsequent comparisons `mark >= 80` and `mark >= 90` are both `false` and the local variable `grade` is left unchanged.

After all the conditions are verified and the local variable `grade` is set accordingly, the reference referred to by the variable `grade` is returned by the following statement:

```
// Return the determined grade
return grade;
```

You can verify that the method can be used to determine a grade based on the aforementioned table.

In order to test the class `GradeConverter`, the following class `TestGradeConverter` is written as in Figure 4.33.

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

// The class definition of TestGradeConverter for setting up the
// environment and test the GradeConverter object
public class TestGradeConverter {

    // The main executive method
    public static void main(String args[]) {
        // Create a GradeConverter object and use variable converter
        // to refer to it
        GradeConverter converter = new GradeConverter();

        // Get the mark from the user
        String inputMark =
            JOptionPane.showInputDialog("Input the mark");
        // Derive the mark in numeric format from the input String
        int mark = Integer.parseInt(inputMark);
        // Get the grade using the GradeConverter object
        String grade = converter.assignGrade(mark);

        // Show the grade to the user
        JOptionPane.showMessageDialog(null, "The grade is " + grade);

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

Figure 4.33 TestGradeConverter.java

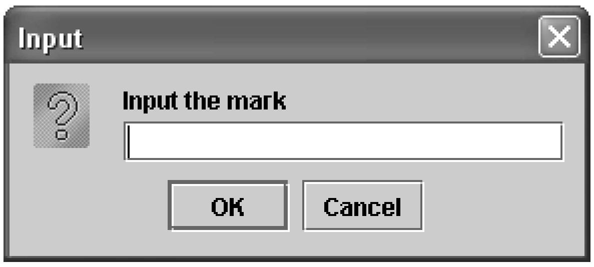
The `main()` method of the class `TestGradeConverter` is quite straightforward. It creates the `GradeConverter` object first and assigns its reference to the local variable `converter`:

```
// Create a GradeConverter object and use variable converter
// to refer to it
GradeConverter converter = new GradeConverter();
```

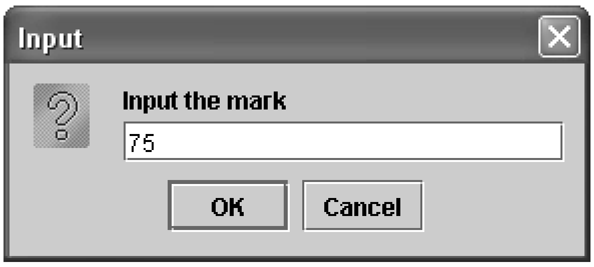
Then, a dialog with `"Input the mark"` is shown to the screen through the following statement:

```
// Get the mark from the user
String inputMark =
    JOptionPane.showInputDialog("Input the mark");
```

That is:



If you enter "75" in the text field, press <Enter> or click <OK>:



A `String` object is created and is assigned to the local variable `inputMark`.

Afterwards, the equivalent numeric value of type `int` is derived

```
    // Derive the mark in numeric format from the input String
    int mark = Integer.parseInt(inputMark);
```

and the derived value, 75 here, is assigned to the local variable `mark`.

Now, the `assignGrade()` method of the `GradeConverter` object is called with parameter 75:

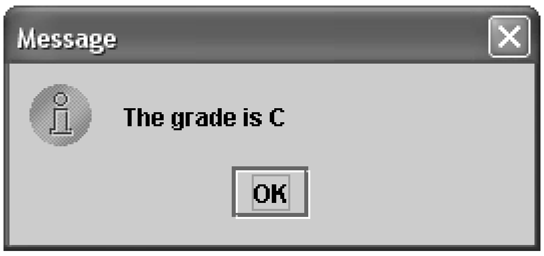
```
    // Get the grade using the GradeConverter object
    String grade = converter.assignGrade(mark);
```

The method will return the `grade` according to the given mark. For a value of 75, a `String` object with content `"C"` is returned and assigned to a local variable `grade`.

Last, the result is shown to the user through the statement:

```
    // Show the grade to the user
    JOptionPane.showMessageDialog(null, "The grade is " + grade);
```

The following dialog appears:



Pressing <Enter> or clicking <OK> closes the dialog. The program is terminated explicitly with:

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

You can test the program with other mark values to see whether it works as expected.

switch/case

Sometimes it is necessary to test the content of a variable against a list of possible values. For example, to determine the English word for numeric value 1 to 9, the following program segment can be used:

```
String word;
if (number == 1) {
    word = "one";
}
if (number == 2) {
    word = "two";
}
if (number == 3) {
    word = "three";
}
if (number == 4) {
    word = "four";
}
if (number == 5) {
    word = "five";
}
if (number == 6) {
    word = "six"
}
if (number == 7) {
    word = "seven"';
}
if (number == 8) {
    word = "eight";
}
if (number == 9) {
    word = "nine";
}
```

You can see that the above program segment compares the content of the variable `number` with integer values from `1` to `9`. The program segment works, but it is a little tedious to place similar conditions one after the other. In Java programming language, another control structure suits the above scenario, the `switch/case` structure. Please use the following reading to acquire some basic knowledge of such a structure.

Reading

King, Section 8.2, pp. 306–14. Try the review questions on p. 314. Check your answers against those on p. 337.

From the reading, you now know that a general format for using the `switch/case` structure is:

```
switch (expression) {
case constant-expression: statement(s)
......
case constant-expression: statement(s)
default: statement(s)
}
```

The following is a list of points that you should note:

The type of *expression* that is enclosed in the parentheses following the keyword `switch` can be `byte`, `short`, `char` or `int`. You cannot use the other primitive types (`long`, `float`, `double`, `boolean`) or non-primitive types as the expression.

1 You are not required to list the *constant-expressions* that are to be tested against the expression in order. However, if you do list the cases in some form of meaningful order, it can help you determine whether there are any missing cases.

2 The `default` part is optional. If there is no *constant-expression* that matches the *expression* in the parentheses following the keyword `switch`, all statements in the `switch/case` structure are ignored.

For example, a program segment with a `switch/case` structure:

```
switch (expr) {
case expr-1: statements-1
case expr-2: statements-2
case expr-3: statements-3
default: statements-default
}
```

can be visualized as in Figure 4.34.

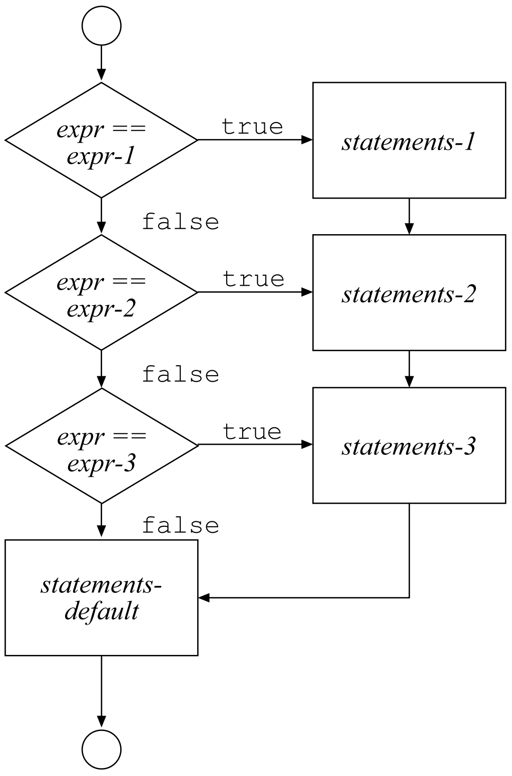


Figure 4.34 An example of a simple switch/case statement

From Figure 4.34, it is clear that if one of the conditions is `true` — that is, there is a matching case — the statements corresponding to that condition will be executed. Furthermore, the statements for the subsequent conditions will be executed as well, but such a structure is not what a programmer usually wants.

A more general format is:

```
switch (expr) {
case expr-1:
    statements-1;
    break;
case expr-2:
    statements-2;
    break;
case expr-3:
    statements-3;
    break;
default:
    statements-default;
    break;
}
```

In each part of the `switch/case` structure, the `break` statement is usually used as the last statement. Whenever a `break` statement is executed, the statements in the subsequent cases are ignored and the next statement to be executed is the first statement that follows the `switch/case` structure. You may find that the `break` statement in the last case is redundant, but it is recommended because you might forget to add the `break` statement when you append a new case to the `switch/case` structure.

Figure 4.35 illustrates the interpretation of the `switch/case` structure above.

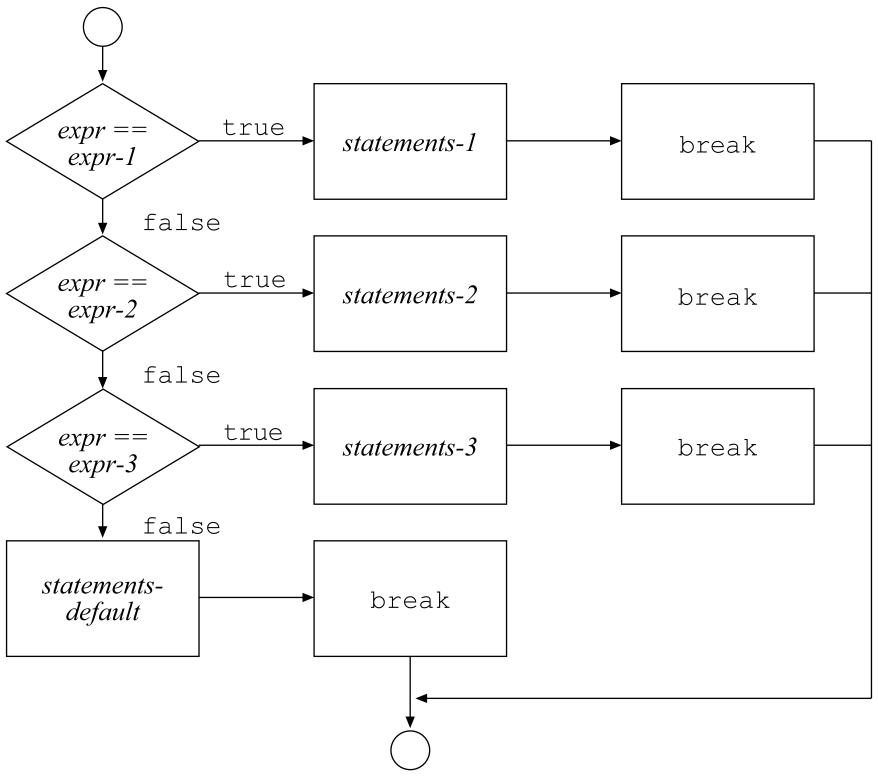


Figure 4.35 An example of switch/case statement with break

The problem of determining the number of days in a month is discussed in the reading, and a sample program was given to show how to use a `switch/case` structure to handle the problem. In the following section, we will solve the problem using different `switch/case` structures starting with a simple format and then using more advanced structures.

Example: number of days in a month

To determine the number of days in a month, you can write a class with a method, say `findDaysInMonth()` that can return the number of days of the given month. Its declaration is:

```
public int findDaysInMonth(int month) {
......
}
```

The first version of the class definition is shown in Figure 4.36.

```
// The class definition of CalendarExpert1
public class CalendarExpert1 {

    // The method for getting the number of days in a month
    public int findDaysInMonth(int month) {
        // Declare a local variable for storing the number of
        // days in the month temporarily
        int days = 0;

        // Determine the number of days based on the given month
        switch (month) {
        case 1: days = 31; break;
        case 2: days = 28; break;
        case 3: days = 31; break;
        case 4: days = 30; break;
        case 5: days = 31; break;
        case 6: days = 30; break;
        case 7: days = 31; break;
        case 8: days = 31; break;
        case 9: days = 30; break;
        case 10: days = 31; break;
        case 11: days = 30; break;
        case 12: days = 31; break;
        }

        // Return the number of days in the month
        return days;
    }
}
```

Figure 4.36 CalendarExpert1.java

In the `findDaysInMonth()` method, the month presented to the method via the parameter is placed in the parentheses following the `switch` keyword. As a result, the content of the parameter `month` is matched with the list of constants, from `1` to `12`. Whenever a constant is matched, the corresponding statements are executed, so that the number of days is assigned to the variable `days`. For example, if the value stored by `month` is `2`, it matches the following case,

```
case 2: days = 28; break;
```

The statement `days = 28;` is executed and the local variable `days` is updated to be the number of days in February. Then, the `break` statement is executed, and the next statement to be executed is:

```
// Return the number of days in the month
return days;
```

As a result, the number of days in the month is returned to the place the method is called.

To use the above `CalendarExpert1` class, the following class is written as shown in Figure 4.37.

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

// The class definition of ExpertTester1 for setting up the
// environment and test the CalendarExpert1 object
public class ExpertTester1 {

    // The main executive method
    public static void main(String args[]) {
        // Create a CalendarExpert1 object and use variable expert
        // to refer to it
        CalendarExpert1 expert = new CalendarExpert1();

        // Get the month from the user
        String inputMonth =
            JOptionPane.showInputDialog("Input the month");

        // Derive the month in numeric format from the input String
        int month = Integer.parseInt(inputMonth);

        // Get the grade using the CalendarExpert1 object
        int daysInMonth = expert.findDaysInMonth(month);

        // Show the number of days in the month to user
        JOptionPane.showMessageDialog(null,
            "Number of days in the month is " + daysInMonth);

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

Figure 4.37 ExpertTester1.java

The first version of the `switch/case` structure is better than writing 12 `if` statements. With some modifications, the `switch/case` structure can be further enhanced. The following second version of the class, `CalendarExpert2`, is shown in Figure 4.38

```
// The class definition of CalendarExpert2
public class CalendarExpert2 {

    // The method for getting the number of days in a month
    public int findDaysInMonth(int month) {
        // Declare a local variable for storing the number of
        // days in the month temporarily
        int days = 0;

        // 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;
        case 1: case 3: case 5:
        case 7: case 8: case 10:
        case 12:
            days = 31;
            break;
        }

        // Return the number of days in the month
        return days;
    }
}
```

Figure 4.38 CalendarExpert2.java

In the second version of the `switch/case` structure (Figure 4.38), the months with the same number of days are grouped. Using the property that execution in a `case` segment continues until a `break` statement is met, the above `switch/case` structure can perform exactly the same as the first version.

To use the second version of the class, `CalendarExpert2`, another program `ExpertTester2` is written. The only difference between the classes `ExpertTester1` and `ExpertTester2` is the following statement:

```
// Create a CalendarExpert2 object and use variable expert
// to refer to it
CalendarExpert2 expert = new CalendarExpert2();
```

Even though the second version is simpler than the first version, it can be further simplified as in the following third version in Figure 4.39.

```
// The class definition of CalendarExpert3
public class CalendarExpert3 {

    // The method for getting the number of days in a month
    public int findDaysInMonth(int month) {
        // Declare a local variable for storing the number of
        // days in the month temporarily
        int days = 0;

        // 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;
        }

        // Return the number of days in the month
        return days;
    }
}
```

Figure 4.39 CalendarExpert3.java

The `switch/case` structure used is similar to the version provided in the reading. As the majority of the months have 31 days, it is simpler to use `default` for months with 31 days.

A class `ExpertTester3` is written for testing the class `CalendarExpert3`. The `ExpertTester3` is similar to the classes `ExpertTester1` and `ExpertTester2` with the following statements modified:

```
// Create a CalendarExpert3 object and use variable expert
// to refer to it
CalendarExpert3 expert = new CalendarExpert3();
```

Now, please use the following self-test to make sure that you are familiar with the `switch/case` structure.

Self-test 4.4

1 Compare the classes `CalendarExpert1`, `CalendarExpert2` and `CalendarExpert3`. Do they function exactly the same? (Hint: The value returned by the `findDaysInMonth()` method of the class `CalendarExpert3` is different in some cases from those of `CalendarExpert1` and `CalendarExpert2`.)

2 Write a class `NumberTranslator` that has a method `assignWordFromNumber()` that returns `String` objects with contents `"one"`, `"two"`, to `"ten"` for numbers `1`, `2` to `10` respectively. The declaration of the `assignWordFromNumber()` method is:

```
public String assignWordForNumber(int number) {
.. .. ..
}
```

Then, write a testing program, `TestTranslator`, to use the `NumberTranslator` class. (Hint: You can use the classes `CalendarExpert1` and `ExpertTester1` as the blueprints.)

Comparisons between `if` / `else` statements and `switch` / case statements

The Java programming language only supports two branching structures, `if/else` statements and `switch/case` statements. One structure may be more suitable for some cases and the other one is more suitable for others. In some cases, they can be used interchangeably.

In this section, we compare these two structures so that you can have a better understanding of them and can choose the more suitable one while you are programming.

The condition of the `if/else` statement is of `boolean` type, whereas the expression type of a `switch/case` statement can only be `byte`, `short`, `char` or `int`. Although it seems that the `if/else` structure is more restricted (only one type is supported), it can actually involve any primitive type and non-primitive type, provided that proper relational operators are used.

The `switch/case` only supports equality checking, but the `if/else` structure can use any relational operator. Furthermore, as the `switch/case` structure can support expression types `byte`, `short`, `char` and `int` only, you must use `if/else` statements to check the equality for `long`, `float`, `double`, `boolean` and non-primitive types. For example, the following `switch/case` statement is invalid:

```
long ln;
......
switch (ln) {
case 1L: ......
case 2L: ......
}
```

Instead, you have to write the program segment as:

```
long ln;
......
if (ln == 1L) {
    ......
}
if (ln == 2L) {
    ......
}
```

Although `switch/case` is more restricted than the `if/else` structure, its structure is more elegant and easier to read. Therefore, it is recommended you choose the `switch/case` statement whenever possible.

Some situations may need some 'tricks' when using the `switch/case` structure. For example, in the problem of determining the grades based on the marks, we used a sequence of `if` statements. Instead, the method `assignGrade()` of the class `GradeConverter` (Figure 4.32) can be modified to use a `switch/case` statement, as in Figure 4.40.

```
// The class definition of GradeConverter2
public class GradeConverter2 {

    // The method for getting the grade based on the given mark
    public String assignGrade(int mark) {
        // The grade to be returned
        String grade;

        // Set the criteria for the grades one by one
        switch (mark / 10) {
        case 9: case 10: grade = "A"; break;
        case 8:          grade = "B"; break;
        case 7:          grade = "C"; break;
        case 6:          grade = "D"; break;
        case 5:          grade = "E"; break;
        default:         grade = "F"; break;
        }

        // Return the determined grade
        return grade;
    }
}
```

Figure 4.40 GradeConverter2.java

In the `assignGrade()`method, the expression in the parentheses that follows the keyword `switch` is not a variable but an expression of integer division. Because integer division only gives an integer through truncation, if the value stored by the parameter `mark` is `95`, integer division `95 / 10` gives `9`. Therefore, the expression result matches `case 9` and the `String` object with content `"A"` is assigned to the local variable `grade` to be returned.

Another class `TestGradeConverter2` is written to test the `GradeConverter2` class. As it is similar to the `GradeTester` class (Figure 4.33), it is not listed here. You can find the source code in the course CD-ROM or website.

Looping

In a method, it may be necessary to execute the same statement several times. For example, if it is necessary to display the message `"Hello World"` three times in a method, it is possible to write the following program segment:

```
System.out.println("Hello World");
System.out.println("Hello World");
System.out.println("Hello World");
```

Then, when that method is called, the message `"Hello World"` is displayed three times in the Command Prompt window. What if it is necessary to print the message `"Hello World"` 10 times? The program is modified, and seven more identical statements are appended to the program segment.

```
System.out.println("Hello World");
System.out.println("Hello World");
System.out.println("Hello World");
System.out.println("Hello World");
System.out.println("Hello World");
System.out.println("Hello World");
System.out.println("Hello World");
System.out.println("Hello World");
System.out.println("Hello World");
System.out.println("Hello World");
```

The modification is tedious and inflexible, because it requires re-compilation. Therefore, we usually construct the program segment to be executed repeatedly as a loop structure, as shown in Figure 4.41. For example:

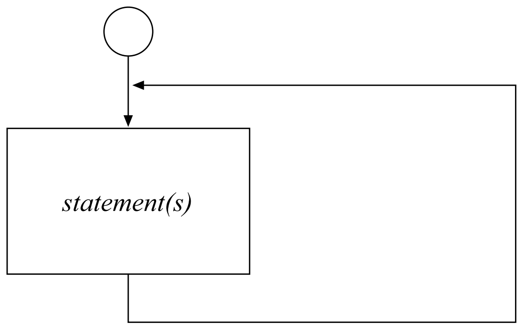


Figure 4.41 A simple (but infinite) loop

The above flow of control is considered a loop. The statement(s) that is (are) executed repeatedly is (are) known as the *loop body*. From the above flowchart, you can see that there is no way out once the loop body is executed. Such a loop is unusual and is considered an *infinite loop*. Except for some particular reasons, such a structure should never be implemented in a program because it will repeat the same operation forever and probably will never generate any useful result. How to prevent your program from executing an infinite loop is discussed later.

Therefore, a usual and proper looping structure has to verify a condition either before or after the loop body is executed. Figure 4.42 is one such looping structure.

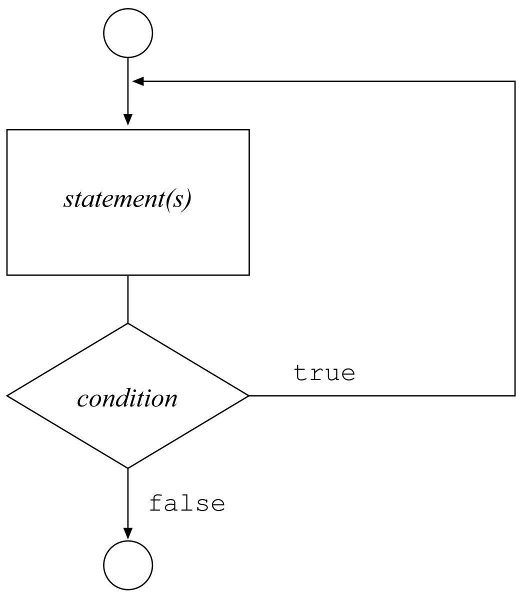


Figure 4.42 A loop with condition checking

The above flowchart indicates a looping structure that performs a loop *condition* checking every time after the loop body is executed. Whenever the loop body is executed once, it is considered an *iteration*.

The Java programming language supports three types of looping structure, which we discuss soon. Afterwards, we'll compare them so that you can determine the best one in different situations.

`while` loops

Please use the following reading to learn about `while` loops.

Reading

King, Section 4.6, pp. 151–56. Answer the review questions on p. 156. Check your answers on p. 179.

From the reading, you now know that the general format of a `while` loop is:

```
while (condition) {
     statement(s)
}
```

The *condition* in the parentheses is a `boolean` expression that gives a `true` or a `false` value. As long as the *condition* is `true`, the *loop body* [the *statement(s)* enclosed in the curly brackets] is executed.

Figure 4.43 should help you understand the `while` loop.

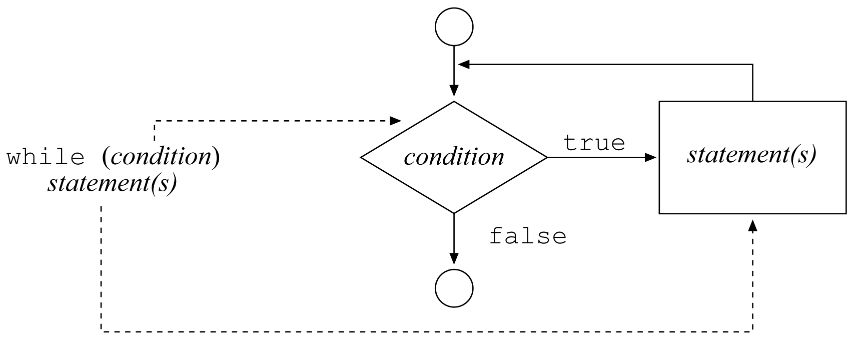


Figure 4.43 while loop

Using Figure 4.43, it is easy to understand that the *condition* is verified before the loop body is executed for the first time. If the *condition* is `true`, the loop body is executed once. Afterwards, the *condition* is tested again, and the loop body is executed again if the *condition* is still `true`. The process repeats until the *condition* is evaluated to be `false` and the loop structure is terminated. If the condition is `false` in the first check, the loop body is not executed.

We mentioned that if an `if` statement block contains only one statement, the pair of curly brackets is optional. Similarly, if there is only one statement in the loop body, the `while` loop can be written as:

```
while (condition)
    statement
```

However, like the `if/else` statements, it is highly recommended you use the pair of curly brackets even if there is only a single statement.

In the reading, the program segment that can be used here to reinforce your understanding of the loop structure is:

```
i = 1;
n = 10;
while (i < n)
    i *= 2;
```

According to our recommendation above, this program segment should be modified to:

```
i = 1;
n = 10;
while (i < n) {
    i *= 2;
}
```

In the above program segment, the assignment operator `*=` in the statement

```
    i *= 2;
```

is one of the assignment operators supported by the Java programming language. The above statement is equivalent to the following statement:

```
    i = i * 2;
```

The above program segment can be visualized in Figure 4.44.

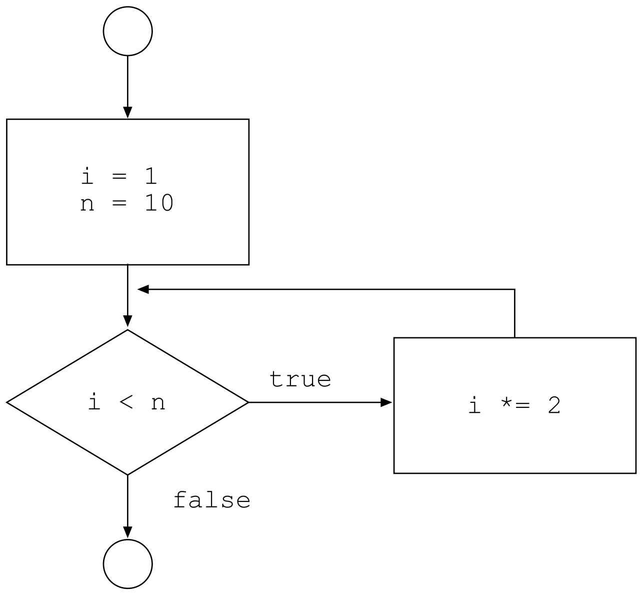


Figure 4.44 Initializing, testing and updating in a loop

In Figure 4.44, there is one more process box that contains two statements:

```
i = 1;
n = 10;
```

This process box is executed before the loop starts; the involved statements are usually used to initialize some variables that are evaluated in the condition. This step is known as *loop initialization*.

Table 4.4 summarizes the execution details of the loop. (The values of `i` and `n` are their values at the time the condition `i < n` is verified.)

Table 4.4 Execution details of a loop

Iteration	Value of `i`	Value of `n`	Value of `i < n`	Action to be taken
1	`1`	`10`	`true`	The loop body is executed, and the value of variable `i` becomes `2`.
2	`2`	`10`	`true`	The loop body is executed, and the value of variable `i` becomes `4`.
3	`4`	`10`	`true`	The loop body is executed, and the value of variable `i` becomes `8`.
4	`8`	`10`	`true`	The loop body is executed, and the value of variable `i` becomes `16`.
5	`16`	`10`	`false`	The loop is terminated

As a result, the value of the variable `i` is 16 when the loop is terminated.

Example: printing a string repeatedly

We said that printing the message `"Hello World"` 10 times by writing a program segment with 10 identical statements is tedious and inflexible. You can use the `while` loop to do this instead. For example, the following program segment can be used to print the message `"Hello World"` 10 times:

```
int i=0;
while (i < 10) {
    System.out.println("Hello World");
    i = i + 1;
}
```

Let's trace the execution of the above program segment.

First, a local variable `i` is declared and initialized as `0`. This is the loop initialization. Then, the condition `i < 10` is verified. As the condition `0 < 10` is `true`, the loop body is executed for the first time. The message `"Hello World"` is printed. Afterwards, the value of variable `i` is increased by one and becomes `1`.

The loop condition is tested again. The condition `1 < 10` is still `true`, and the message `"Hello World"` is displayed for the second time. Then, the value of variable `i` is increased by one again and becomes `2`.

The loop repeats; when the message `"Hello World"` is printed for the tenth time, variable `i` is `9`. Afterwards, the value of variable `i` is increased by one again and becomes `10`.

Finally, the condition `i < 10` is evaluated and `10 < 10` is `false`. Therefore, the loop body is skipped and the loop is terminated.

Besides tracing the `while` loop in our imagination, we use a modified version of the above `while` loop to write a `HelloSpeaker` class that can display the message `"Hello World"` a particular number of times.

The following definition of the `HelloSpeaker` class is shown in Figure 4.45.

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

    // The method for printing the message "Hello World" to
    // the Command Prompt repeatedly for the number of times
    // specified in the parameter
    public void sayHello(int times) {
        // The number of times the message has been spoken, and
        // it is initially zero as nothing has been spoken yet.
        int timeSpoken = 0;

        // While the times spoken is still less than the number
        // of times required, continue to speak
        while (timeSpoken < times) {
            // Speak the message
            System.out.println("Hello World");

            // Increase the number of times the message is spoken
            timeSpoken++;
        }
    }
}
```

Figure 4.45 HelloSpeaker.java

The aforementioned `while` loop was modified so that variable `i` has been renamed something more meaningful (`timeSpoken)`. Furthermore, a new operator is introduced in the program, which is the `++` operator in the following statement:

```
timeSpoken++;
```

The operator `++` is known as the *increment operator* and it increases the value of the associated variable by one. Therefore, the above statement is equivalent to the following statement

```
timeSpoken = timeSpoken + 1;
```

but it is much simpler.

Similarly, the *decrement operator* `--` (two minus characters) can decrease the value of the associated variable by one. For example

```
i--;
```

is equivalent to the following statement:

```
i = i - 1;
```

Another difference between the aforementioned `while` loop and the one in the `sayHello()` method of the `HelloSpeaker` class is that the condition is not compared with a variable with a constant value (`10` in the aforementioned `while` loop) but instead with the parameter `times`.

For example, the parameter `times` is given a value `3`. The local variable `timeSpoken` is initialized as `0` and is increased by one every time the message `"Hello World"` is printed in the Command Prompt. Therefore, after the message `"Hello World"` is displayed for the third time, the variable `timeSpoken` has increased to `3`. When the condition `timeSpoken < times` is verified, the result is `false` and the loop is terminated.

As there is no statement following the `while` loop, the execution of the method `sayHello()` is completed and the flow of control is returned to where the method was called.

To test the `HelloSpeaker` class, the following `HelloTester` class is written as shown in Figure 4.46.

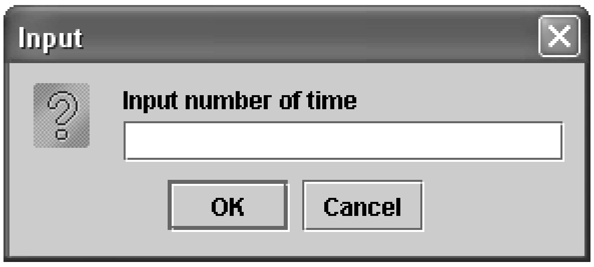
```
// Import statement for resolving the class JOptionPane
import javax.swing.JOptionPane;

// The class definition of HelloTester for setting up the
// environment and test the HelloSpeaker object
public class HelloTester {

    // The main executive method
    public static void main(String args[]) {
        // Create a GradeConverter object and use variable speaker
        // to refer to it
        HelloSpeaker speaker = new HelloSpeaker();

        // Get the number of time required from the user
        String inputTimes =
            JOptionPane.showInputDialog("Input number of time");

        // Derive the times in numeric format from the input String
        int times = Integer.parseInt(inputTimes);

        // Request the HelloSpeaker object to say the message
        // the required number of times
        speaker.sayHello(times);

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

Figure 4.46 HelloTester.java

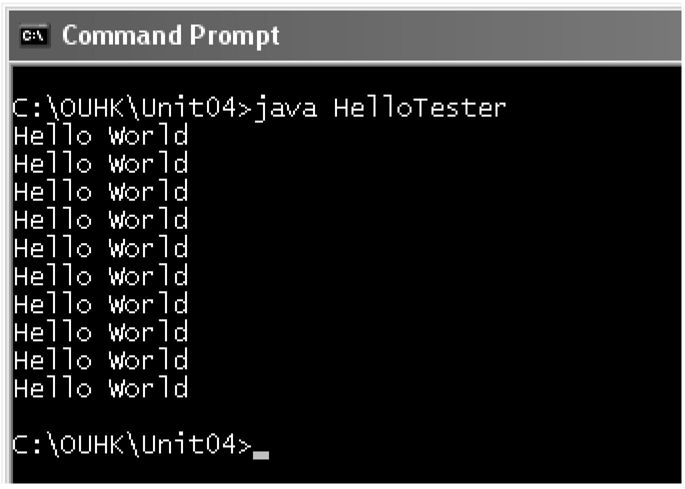
After you have compiled the two classes, `HelloSpeaker` and `HelloTester`, you execute `HelloTester` by:

```
java HelloTester
```

The following dialog appears:



If you enter `"10"` in the text field and press <Enter> or click <OK>, the following message will be shown in the Command Prompt:



```
C:\OUHK\Unit04>java HelloTester
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World

C:\OUHK\Unit04>_
```

do / `while` loops

The second looping structure that the Java programming language supports is the `do/while` loop. The difference between a `while` loop and a `do/while` loop is that the loop condition of the latter is verified after each loop iteration.

Please use the following reading to get some ideas about the `do/while` loop. The control structures are discussed afterwards.

Reading

King, Section 8.3, pp. 314–16. The answers to the review questions on p. 316 are on p. 337.

From the reading, you learned that the syntax of the `do/while` loop structure is:

```
do
    statement
while (condition);
```

The above structure only accepts one statement in the loop body. If the loop body has to contain more than one statement, you are required to use a pair of curly brackets to enclose the statements. Therefore, the usual and recommended `do/while` loop structure is:

```
do {
    statement(s)
} while (condition);
```

Figure 4.47 will help you understand the operations of a `do/while` loop structure.

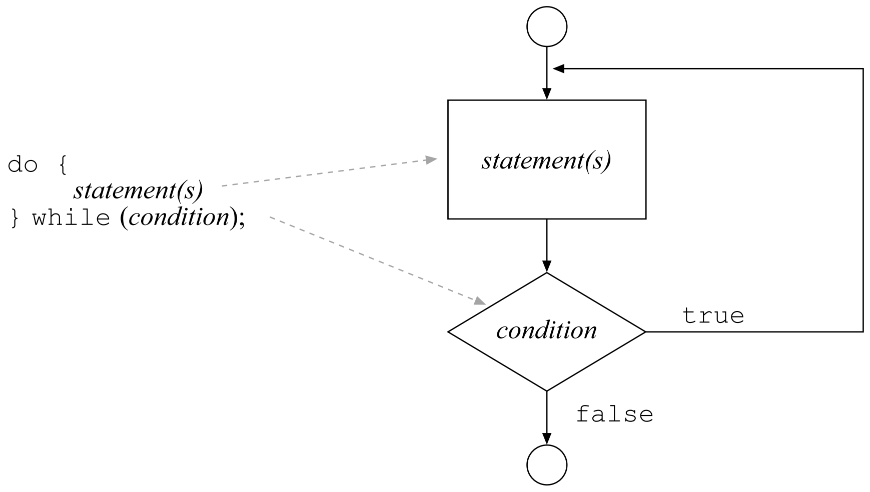


Figure 4.47 Operations in a do/while structure

The expression for the `do/while` loop *condition* is placed after the loop body (*statement(s)*). Therefore, the loop body is executed without verifying any condition for the first iteration. Then, the *condition* is evaluated. If the *condition* is evaluated to be `true`, the loop body will be executed again. Otherwise, it is terminated. You should notice that the loop body of a `do/while` loop must be executed at least once.

Example: calculating the factorial

In mathematics, some calculations involve a sequence of repeated operations, such as repeated multiplications in the following formulas:

$a^n = a \times a \times \ldots\ldots \times a$ (for n times)
$n! = n \times (n\text{-}1) \times \ldots\ldots \times 1$

(The expression *n*! is known as the *factorial* of *n*. By definition, the factorial of zero, i.e. 0!, is 1.) For example:

$2^5 = 2 \times 2 \times 2 \times 2 \times 2$, and
$5! = 5 \times 4 \times 3 \times 2 \times 1$

In the standard Java API library, there is no method to calculate the factorial of an integer; you need to develop your own method to calculate it.

You cannot construct an expression in a program on the fly for a specific value of a number. Therefore, you need to multiply the number one-by-one from one to the given value of the number (or equivalently, to multiply the number from the given value of the number to one).

The following `FactorialCalculator` class definition (Figure 4.48) is written to calculate the factorial of an integer (see Question 2 of Self-test 4.5 about the issue of 0!).

```
// 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) {
        // Declare a local variable for storing the intermediate
        // value during iteration, and its value is initialized to
        // be 1
        int result = 1;
        do {
            // The intermediate result is multiplied by the
            // the current value of variable number
            result *= number;

            // Decrease the value of number by one
            number--;

        // While the number is greater than one,
        // repeat the loop body
        } while (number > 1);

        // Return the finalized result
        return result;
    }
}
```

Figure 4.48 FactorialCalculator.java

The core program section for calculating the factorial in the above class definition is:

```
int result = 1;
do {
   // The intermediate result is multiplied by the
   // the current value of variable number
   result *= number;

   // Decrease the value of number by one
   number--;

// While the number is greater than one,
// repeat the loop body
} while (number > 1);
```

The above program segment can be visualized in Figure 4.49.

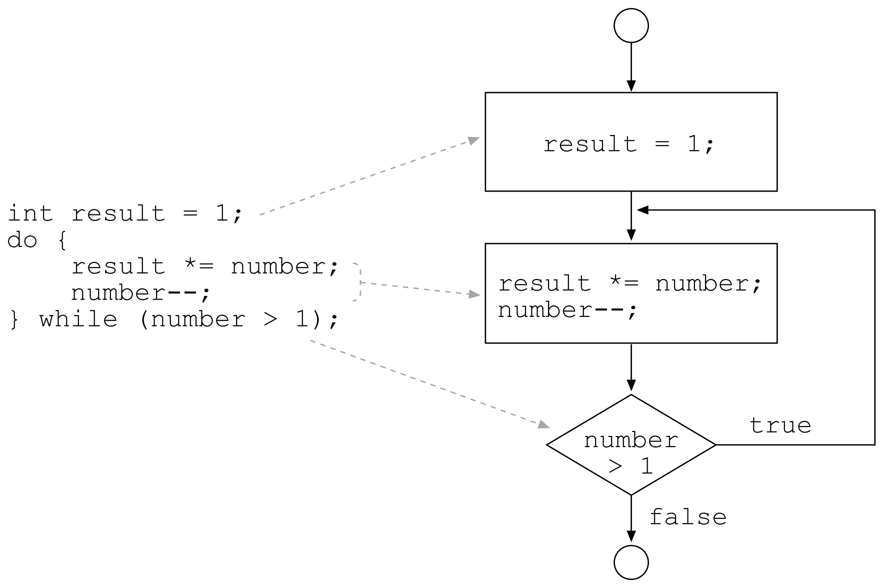


Figure 4.49 Initializing, testing and updating in a do/while loop

In the above program segment, a temporary local variable `result` is declared and is initialized to be `1`. Then, the value of the local variable `result` is updated by its value times the current value of `number` in each iteration.

For example, if the value of `number` is given as `4` via a parameter, Table 4.5 shows the changes in the variables.

Table 4.5 Sequence of operations in FactorialCalculator.java

Iteration	Values of variables before executing statement `result *= number;`		Values of variables after executing statement `number--;`		Condition `number > 1`
	`result`	`number`	`result`	`number`	
1	`1`	`4`	`4 (= 4)`	`3`	`true`
2	`4`	`3`	`12 (= 4 × 3)`	`2`	`true`
3	`12`	`2`	`24 (= 4 × 3 × 2)`	`1`	`false`

From Table 4.5, if the value passed to the method via the parameter is `4`, the condition `number > 1` becomes `false` after the third iteration and the value stored by the variable `result` is `24`. The `do/while` loop terminates, and the value stored by the variable `result` is returned. Therefore, the loop and hence the method correctly calculates the factorial of 4, and the result is 24.

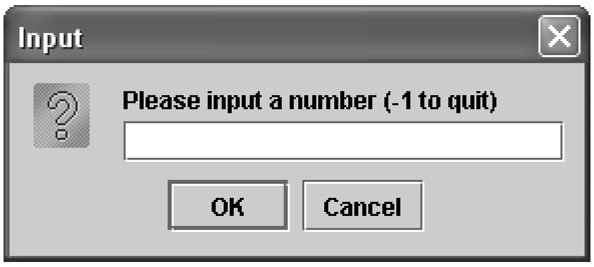
Now, you can prepare a testing program, say `FactorialTester`, to test the `FactorialCalculator` class definition. Up to now, all testing programs accept only the user's input and process and display the output once. Using a `do/while` loop in the `main()` method of the `FactorialTester` can repeat the process over and over again. Figure 4.50 is the program listing of the class `FactorialTester`.

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

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

    // 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();

        int number;
        do {
            // Get a number from the user
            String inputNumber =
                JOptionPane.showInputDialog(
                "Please input a number (-1 to quit)");
            number = Integer.parseInt(inputNumber);

            // If the number is not -1, process and output result
            if (number != -1) {
                // Calculate the factorial
                int result = calculator.factorial(number);

                // Show the result to user
                JOptionPane.showMessageDialog(null,
                    inputNumber + "! = " + result);
            }
        } while (number != -1);

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

Figure 4.50 FactorialTester.java

In the `main()` method of the `FactorialTester` class, an object of `FactorialCalculator` class is created that can be accessed via the variable `calculator`. Then, there is a `do/while` loop for handling more than one user input number.

In the `do/while` loop, a dialog is shown to the user for getting a number:



Please input a number (-1 to quit)

The `String` object that stores the user input is returned and an equivalent integer value is derived. If the number is not `–1`, it is necessary to calculate its factorial and show the result to the user. The loop condition `number != -1` is evaluated to be `true` and the entire loop body is repeated.

If the input number is `–1`, the condition of the `if` statement is `false` and the `if` statement is therefore skipped. Then, this is the end of the `do/while` loop body and the condition `number != -1` is evaluated, and the result is `false`. As a result, the `do/while` loop is terminated and hence the program terminates.

Please use the following self-test to assess your understanding of the `while` and `do/while` loop structures.

Self-test 4.5

1 Execute the `FactorialTester` program to calculate the factorial of 16 and 17. What are the results? Why?

2 What is the result of the factorial of zero (0!) obtained by the running the `FactorialTester` program? Why? Modify the `FactorialCalculator` class so that the `do/while` loop is replaced by a `while` loop. Determine whether it solves the problem.

3 Based on the class definitions for calculating the factorial or otherwise, develop a class, `PowerCalculator`, which calculates the value of `value`$^{\texttt{toPower}}$, with the method declaration as:

```
public double power(double value, int toPower) { ...... }
```

As usual, write a program, `TestPowerCalculator`, which accepts two values from the user and uses a `PowerCalculator` object to get and display the result.

4 A program segment that is intended to be

```
int n = 1;
do
    n *= 2;
while (n < 10);
```

is typed by mistake as:

```
int n = 1;
    n *= 2;
while (n < 10);
```

(The `do` statement is missing.) Will this program segment compile? What will the output be if this program segment is actually executed?

`for` loops

The `for` loop is the last looping structure that the Java programming language supports. Compared with `while` and `do/while` loop structures, it is a little more complicated but is preferable in some situations.

The following reading shows different formats, uses and suggestions for using `for` loop structures. Additional illustrative material follows the reading.

Reading

King, Section 5.2, pp 188–94. The answers to the review questions on p. 194 can be found on p. 220.

From the reading, you know that the general format of a `for` loop structure is:

```
for (initialization; test; update)
        statement
```

Like other loop structures, it is recommended you use a pair of curly brackets even if there is only one statement. The format becomes:

```
for (initialization; test; update) {
        statement(s)
}
```

As mentioned in the reading, Figure 4.51 shows the equivalence between the `for` loop and `while` loop:

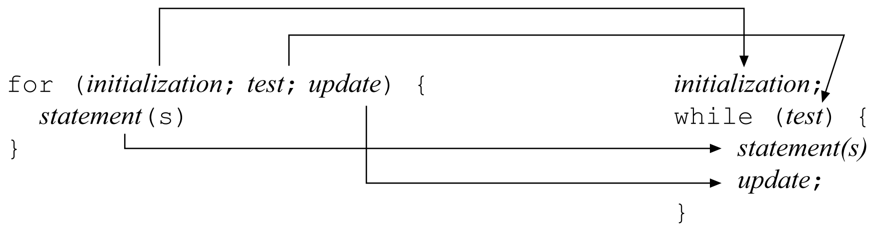


Figure 4.51 Equivalence between the for and while loop

Figure 4.52 clearly shows the flow of control in the `for` loop.

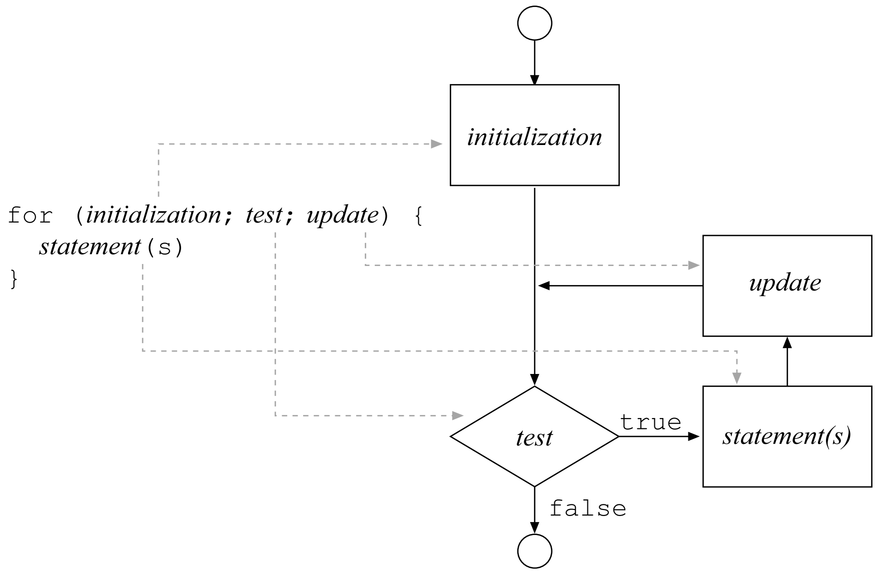


Figure 4.52 Flow of control in a for loop

Please be aware that as the `for` loop structure does not clearly show the flow of control among the four parts (*initialization*, *test*, *statement(s)*, *update*), you should use the above flowchart for clarification.

As shown in the flowchart, the *initialization* part is executed first. Then, the *test* condition is verified. If the `test` condition is evaluated to be `false`, the `for` loop will be terminated immediately without executing the loop body (*statement(s)* part) and the *update* part. If the condition is evaluated to be `true`, the *statement(s)* part and the *update* part are executed in order. Afterwards, the condition is tested again and the loop body is executed if the condition is `true`, and this process repeated.

Therefore, you should bear in mind that the loop body of a `for` loop can be completely skipped if the first test is evaluated to be `false`. That is, the condition for an iteration has to be evaluated as `true` so that the iteration is executed.

If it is necessary to print the message `"Hello World"` 10 times, it can be written as the following program segment:

```
for (i = 0; i < 10; i++) {
    System.out.println("Hello World");
}
or
for (i = 1; i <= 10; i++) {
    System.out.println("Hello World");
}
```

Although the above two program segments work in exactly the same way, there is a difference — the values of variable `i` in the `for` loop body are different. For the former `for` loop, the values of variable `i` range from `0` to `9`; in the latter one, the values of variable `i` range from `1` to `10`. For the same first iteration, the value of variable `i` in the former `for` loop is `0` whereas `i` in the latter `for` loop is `1`. As the result, the outputs as shown from the following two program segments are different:

```
for (i = 0; i < 10; i++) {  for (i = 1; i <= 10; i++)
    System.out.println(     {
      i + ":Hello World");      System.out.println(
}                                 i + ":Hello World");
                            }
0:Hello World               1:Hello World
1:Hello World               2:Hello World
2:Hello World               3:Hello World
3:Hello World               4:Hello World
4:Hello World               5:Hello World
5:Hello World               6:Hello World
6:Hello World               7:Hello World
7:Hello World               8:Hello World
8:Hello World               9:Hello World
9:Hello World               10:Hello World
```

A `for` loop usually involves a control variable, or an index, such as variable `i` in the above `for` loop structures. You usually need such a control variable to be increased or decreased by one. The increment operator `++` and the decrement operator `--` are usually used. However, the *update* part of the `for` loop can be more flexible. For example, you can use the following program segment to display the odd numbers in the range of `1` to `10`:

```
for (i = 1; i < 10; i+= 2) {
    System.out.println(i);
}
```

The output will be:

```
1
3
5
7
9
```

Before the first iteration, variable `i` is initialized as `1` and the test `i < 10` is evaluated. The test result is `true` and the value of variable `i`, `1`, is displayed. The update part `i += 2` of the `for` loop is executed and the value of variable `i` is updated to be `3`. The test is re-evaluated and the result is still `true`. The loop body is executed again and the value of `3` is displayed. The process continues until the value of variable `i` becomes `9` and is displayed. Then, the test result becomes `false` and the `for` loop is terminated.

The control variables or indexes used in `for` loops are usually used for counting the iterations. They are therefore considered to be throwaway variables and it is acceptable to use short variable names, such as `i`, `j` and `k`. Furthermore, as they are only used in `for` loops but not anywhere else in the program, you can declare them and initialize them immediately in the `for` loop structures. For example:

```
for (int i = 0; i < 10; i++) {
    statement(s)
}
```

Here, variable `i` is defined in the `for` loop structure and can be accessed only within this `for` loop. The region of the program segment in which a variable can be accessed is known as the *scope* of the variable. The scope of variable `i` in the above program is the `for` loop structure; you cannot access it afterwards. For example, the following program segment can cause a compile-time error:

```
for (int i = 0; i < 10; i++) {
    ......
}
System.out.println(i);
```

After the `for` loop structure, variable `i` is no longer accessible, and a compile time error occurs.

If your program segment or a method needs two or more `for` loop structures and each needs a control variable `i`, you can either declare a variable `i` preceding all `for` loops, or each `for` loop structure declares its own variable `i`; that is:

```
......
int i;
for (i = 0; i < 10; i++) {
    ......
}
......
for (i = 0; i < 10; i++) {
    ......
}
......
or
......
for (int i = 0; i < 10; i++) {
    ......
}
......
```

```
for (int i = 0; i < 10; i++) {
    ......
}
......
```

Section 5.2 in King mentions that the *three* parts (initialization, update and test) of a `for` loop are optional even though it is not good programming practice to omit them. Figure 4.52 shows the related parts so that you can imagine what the flow of control would be if some parts of a loop were omitted. If any part in the parentheses is omitted, the corresponding process box or decision box is bypassed.

If the *test* part is omitted (see Figure 4.53), the decision box is bypassed and the alternative path is chosen. Therefore, you can consider the test expression is by default `true` if it is omitted and an infinite loop is formed.

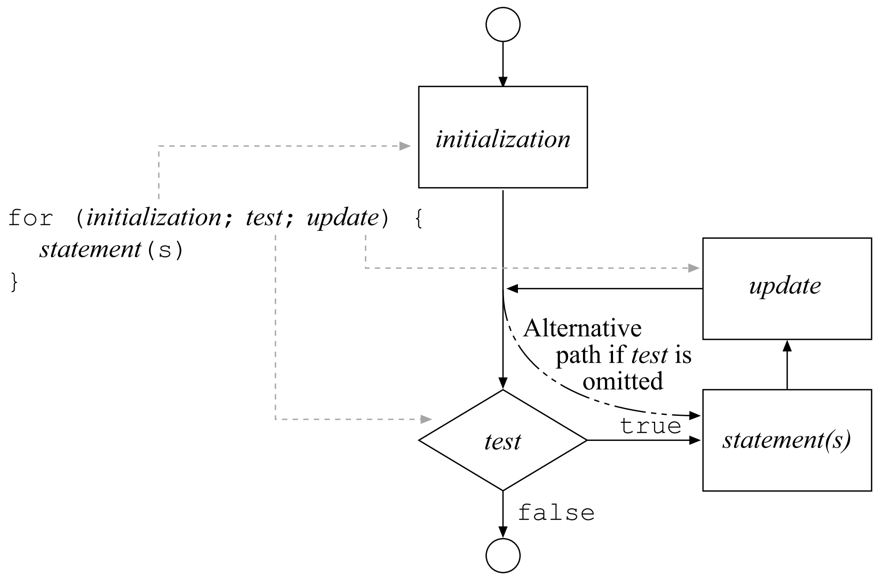

initialization
for (initialization; test; update) {
statement(s)
}
update
Alternative path if test is omitted
test
true
statement(s)
false

Figure 4.53 Omitting the test part in a for loop

The extreme case is that all three parts are omitted, that is:

```
for (;;) {
    statement(s)
}
```

You can imagine the equivalent flowchart to be like Figure 4.54.

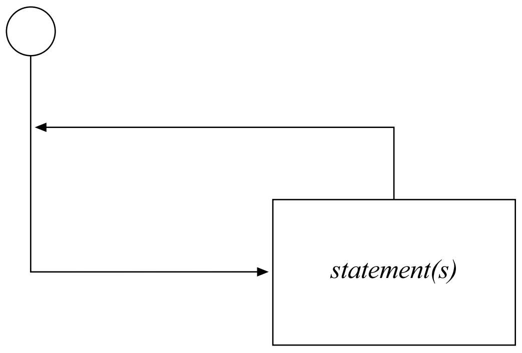


Figure 4.54 Creation of an infinite loop

You can see that there is no way out of the loop, which is effectively an infinite loop. The use of such a control structure is relatively rare, and you need some advanced features of control structure to handle it. We revisit this issue in *Unit 6*.

Example: finding the sum of numbers from 1 to 100

The changing in the value of the control variable is a handy way to process numbers, especially those with a regular pattern. For example, to find the sum of consecutive numbers from 1 to 100, you can use a simple `for` loop as follows:

```
int total = 0;
for (int i = 1; i <= 100; i++) {
    total += i;
}
```

In the above program segment, a local variable `total` is declared and initialized as `0`. The values of the control variable `i` for the first iteration and the last iteration are `1` and `100` respectively. Its value is increased by `1` after each iteration. Therefore, the variable `i` will take values from `1`, `2`, `3`, … `99`, `100` and its values are accumulated with the use of the local variable `total`. As a result, the value of `total` when the `for` loop quits is equal to the sum of numbers from `1` to `100`.

To make the above more flexible, the condition is modified so that it is compared with a variable instead of a constant value, and the above loop can be used to calculate the sum of numbers from 1 to a given limit. Such a method is implemented as the `findSumOfAll()` method of the `NumberAdder` class in Figure 4.55.

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

    // The method for calculating the sum of number from
    // 1 to the given limit
    public int findSumOfAll(int limit) {
        // Declare a local variable for storing the intermediate
        // value during iteration, and its value is initialized to
        // be 0
        int total = 0;

        // The loop for adding numbers from 1 to the given limit
        for (int i=1; i <= limit; i++) {
            total += i;
        }

        // Return the sum of all
        return total;
    }
}
```

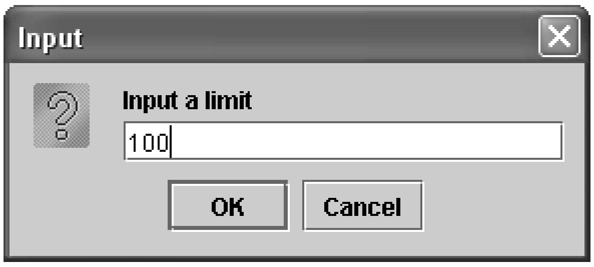
Figure 4.55 NumberAdder.java

You can use the following `TestNumberAdder` class in Figure 4.56 to test the `NumberAdder` class.

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

// The class definition of TestNumberAdder for setting up the
// environment and test the NumberAdder object
public class TestNumberAdder {

    // The main executive method
    public static void main(String args[]) {
        // Create a NumberAdder object and use variable adder
        // to refer to it
        NumberAdder adder = new NumberAdder();

        // Get the limit from the user
        String inputLimit =
            JOptionPane.showInputDialog("Input a limit");

        // Derive the number in numeric format from the input String
        int limit = Integer.parseInt(inputLimit);

        // Request the NumberAdder object to calculate the sum of
        // all numbers from 1 to the limit
        int sum = adder.findSumOfAll(limit);

        // Show the sum to the user
        JOptionPane.showMessageDialog(null, "The sum is = " + sum);

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

Figure 4.56 TestNumberAdder.java

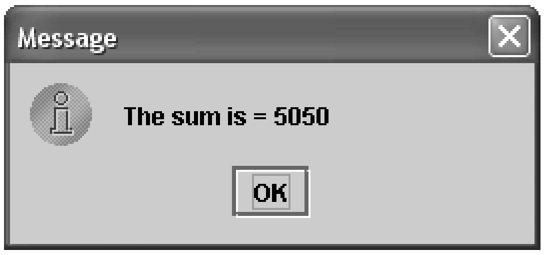
Execute the program with `"100"` as the input entry:



Input a limit

100

Complete the dialog. The sum is shown on the screen:



The sum is = 5050

Please use the following self-test to make sure that you have a solid understanding of the `for` loop control structure.

Self-test 4.6

1 Add two methods, `findSumOfAllOdd()` and `findSumOfAllEven()` to the `NumberAdder` class definition to calculate the sum of all odd numbers and the sum of all even numbers that are less than the given limit passed via the parameter. The method declarations are:

```
public int findSumOfAllOdd(int limit) { ...... }
public int findSumOfAllEven(int limit) { ...... }
```

2 Write a `MarkSixChooser` class that has a `showNumbers()` method for displaying six random numbers in the range 1 to 45 to the Command Prompt. Please write a class, `TestMarkSix`, to test the `MarkSixChooser` class.

Branching and looping control structures usually cooperate to perform some even more complicated tasks. For example, how can you determine the number of multiples of a given number that are less than a particular limit?

A simple and elegant way to do so is to check all numbers that are less than or equal to the limit one-by-one to see which of them are multiples of the given number.

You know that you can iterate the numbers in a range, but how can you determine whether a number is a multiple of another number? The solution is to use the remainder operator `%`. For values stored in variables `a` and `b`, if

```
a % b == 0
```

the value stored by variable `a` is a multiple of the value stored by `b`. As a result, you need an `if` statement like:

```
    if (i % number == 0) {
        ......
    }
```

For this `if` statement, you need a variable, say `count`, to count the multiples. With the above components in mind, you can construct a program segment as:

```
int count = 0;
for (int i = number + 1; i <= limit; i++) {
    if (i % number == 0) {
        count++;
    }
}
```

Now, let's see how the above program segment fits into a class definition, `MultipleCounter`, in Figure 4.57.

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

    // The method for counting the number of multiples
    // of a given number that are less than or equal to
    // the limit
    public int countMultiple(int number, int limit) {
        // Declare a local variable for storing the intermediate
        // count during iteration, and its value is initialized to
        // be 0
        int count = 0;

        // The loop for counting multiples from (number + 1) to
        // the limit
        for (int i = number + 1; i <= limit; i++) {

            // Check if the current control variable is a multiple
            // of the given number. If so, increase the count
            if (i % number == 0) {
                count++;
            }
        }

        // Return the count
        return count;
    }
}
```

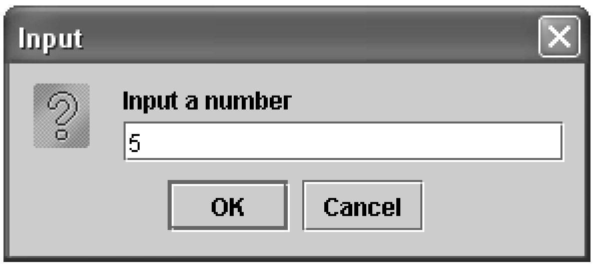
Figure 4.57 MultipleCounter.java

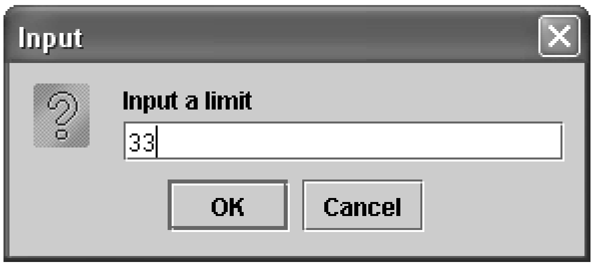
The following class `TestMultipleCounter` is written to test the `MultipleCounter` class. It accepts two inputs from the user, the `number` and the `limit` respectively. The program listing is shown in Figure 4.58.

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

// The class definition of TestMultipleCounter for setting up the
// environment and test the MultipleCounter object
public class TestMultipleCounter {

    // The main executive method
    public static void main(String args[]) {
        // Create a NumberAdder object and use variable counter
        // to refer to it
        MultipleCounter counter = new MultipleCounter();

        // Get the number and limit from the user
        String inputNumber =
            JOptionPane.showInputDialog("Input a number");
        String inputLimit =
            JOptionPane.showInputDialog("Input a limit");

        // Derive the number and limit in numeric format from
        // the input String
        int number = Integer.parseInt(inputNumber);
        int limit = Integer.parseInt(inputLimit);

        // Request the NumberAdder object to calculate the sum of
        // all numbers from 1 to the limit
        int count = counter.countMultiple(number, limit);

        // Show the sum to the user
        JOptionPane.showMessageDialog(null,
            "There are " + count + " multiple(s)");

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

Figure 4.58 TestMultipleCounter.java

For example, the multiples of 5 that are less than or equal to 33 are 10, 15, 20, 25, 30. By running the program

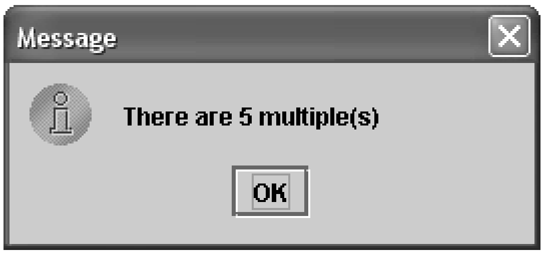




Input a limit

33

the result is:



There are 5 multiple(s)

Please use the following self-test to assess your understanding of the `for` loop control structure.

Self-test 4.7

1 Write a class named `GreetingSpeaker` with a method named `sayGreeting()`. The method declaration is:

```
public void sayGreeting(int total) { ...... }
```

By providing a number, the method displays the greetings `"Good morning"`, `"Good afternoon"` and `"Good evening"` repeatedly in the Command Prompt. The number of greeting messages displayed is determined by the parameter `total`. For example, if the parameter is given as `5`, five greeting messages will be displayed:

```
Good morning
Good afternoon
Good evening
Good morning
Good afternoon
```

Write a driver program named `TestGreetingSpeaker` that accepts a number from the user and calls the `sayGreeting()` method of a `GreetingSpeaker` object.

(Hints: You need a `for` loop structure, a remainder operator (`%`) and a `switch/case` control structure.)

Designing looping structures

The baseline of a looping structure is a *loop body* to be repeatedly executed and a loop *condition* to indicate whether the loop should continue. The condition usually involves at least one variable (such as the control variable in a `for` loop), or it is probably an infinite loop.

There must be a statement (or statements) in the loop body that will update the variable. Otherwise, the loop is either an infinite loop (if the condition result is `true`) or will quit immediately (if the condition result is `false`) when the condition is checked. Usually, there is at least one variable in the condition and the statement(s) not in the condition but that may update the variable(s) can be treated as the *update* part of the loop. Strictly speaking, the loop body includes the update part; here we separate it from the loop body to make the concept clearer.

For most loops in programs, it is necessary to initialize some variables before the loop starts. Such statements are the *initialization* part of a loop.

As a result, a looping structure can usually be characterized by the execution order of these four parts (some of which may be omitted in some cases) — initialization, condition, loop body (or statements), and update. The different arrangements of these four parts are supported by the three looping structures that the language supports.

The following section provides you with an explanation of these structures so that you can choose the most suitable one while programming.

Comparisons among different looping structures

With respect to the loop structures, the most critical feature is whether the loop condition is verified before or after the loop body. Among the three loop structures, `for` loops and `while` loops test the condition before executing the loop body. The `do/while` loop verifies the condition after the loop body is executed to see whether the loop has to continue. The consequence is that the `for` loop and `while` loop can skip the entire loop body completely, whereas the `do/while` loop executes the loop at least once.

Since both `for` loop and `while` loop structures verify the condition before executing the loop body, they are usually interchangeable. The transformations between the two loops are shown in Figure 4.59.

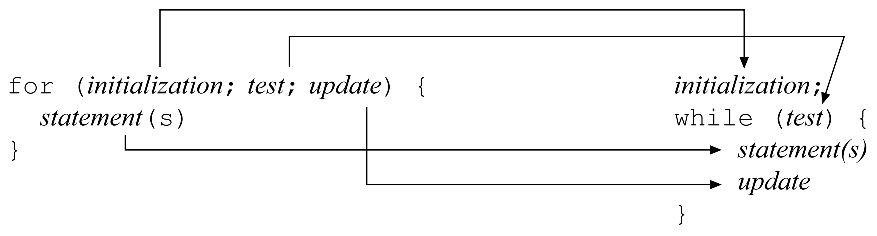


Figure 4.59 Transformations between a for and a while loop

Therefore, you can choose either a `for` loop or a `while` loop if the loop body can be skipped. However, the usual practice is that if the loop must be executed for a predetermined number of times, it is preferable to use a `for` loop. Otherwise, a `while` loop should be used. The underlying reason for this is that all factors that determine the number of iterations are enclosed in the parentheses that follow the keyword `for` (provided that the control variable is not modified in the loop body) in the `for` loop, and it is therefore easier to write (and read).

Figure 4.60 can help determine which loop structure you should use.

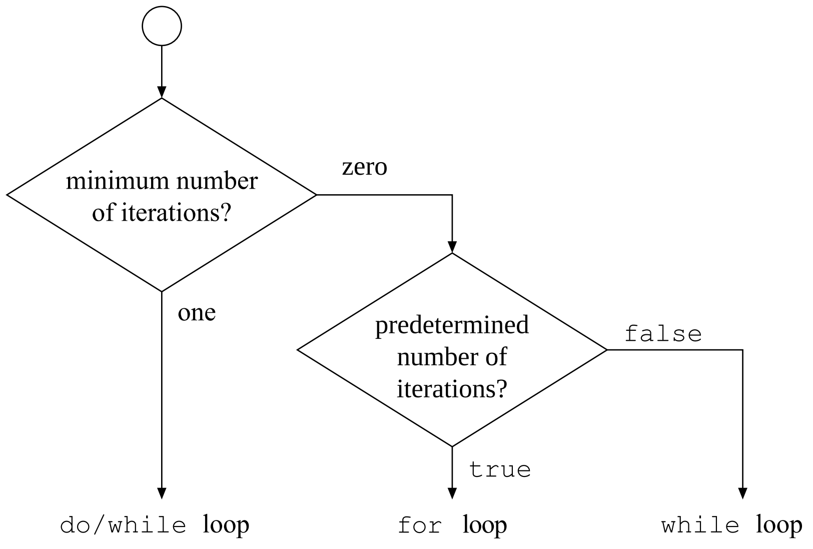

minimum number of iterations?
zero
one
predetermined number of iterations?
false
true
do/while loop
for loop
while loop

Figure 4.60 Comparison among loop types

You've already come across the discussions on infinite loops. Such loops are usually undesirable, but we sometimes need it, for example in a server type application.

Server applications are specially written programs that provide services to clients. They have to keep on executing and wait for service requests from the clients. Therefore, their core parts usually constitute an infinite loop, with a loop body like:

1 Wait for and accept a client request.
2 Handle the request.
3 Report the result to the client.
4 Go back to step 1 for the next request.

You can see that the statements for steps 1 to 3 are the loop body of an infinite loop.

All loop structures can be used to develop infinite loops, just like:

`while` loop	`Do/while` loop	`for` loop

`while (true) {`
statement(s)
`}`
	`do {`
	statement(s)
	`} while (true);`
		`for (;;) {`
		statement(s)
		`}`

Among the three loop structures, which one do you think is the best choice?

The usual practice for writing the infinite loop is to use a `while` loop. The `do/while` is not preferable because it is odd that when a reader studies the program listing, he or she only knows that it is an infinite loop at the end of the loop body. You can learn if it is an infinite loop at the beginning of a `while` loop or a `for` loop structure. However, the double semi-colon in the `for` loop may be strange to some readers, especially those who are not familiar with the programming language. They either wonder whether the program can compile or whether the loop will never execute or will execute forever. As a result, the `while` loop is the best choice.

Prevention of infinite loops

If software is properly designed and coded, it should not be executing an infinite loop. The syndromes of an infinite loop are that the software is busy executing something but there is no output, or the output repeats again and again without a stop. If there is no output, an interactive software package fails to respond.

There are two reasons why a program fails to respond. The first one is the program is running an infinite loop. The other reason is that it is waiting for the occurrence of some event(s), such as the completion of downloading a file from the Internet. The former reason is the result of a bad software design or implementation, but the latter one is reasonable.

Figure 4.61 shows some software designed to show the user that it is working properly but is just waiting for some events to occur, or it still needs substantial time to complete the operation. For example, all Web browsers show an animated icon whenever they are getting and showing the Web pages.

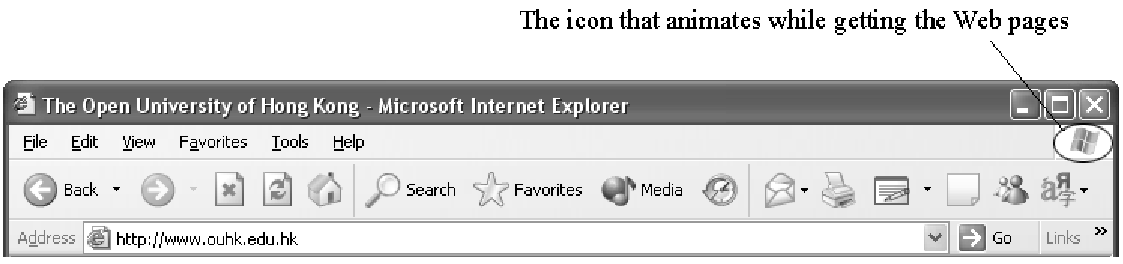

The icon that animates while getting the Web pages
The Open University of Hong Kong - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
Media
Address http://www.ouhk.edu.hk
Go
Links

Figure 4.61 'Animated' icon

Let's have a look at the following program segment. Is there any problem with this?

```
for (short i = 0; i < 100000; i++) {
  ......
}
```

The program segment is apparently a loop that will iterate 100,000 times. However, it turns out to be executing forever. That is, it is actually an infinite loop. But why?

To answer the question, you should first understand that a loop becomes an unintentional infinite loop because its loop condition is evaluated to be `true` forever. Therefore, when you are writing a loop structure, you should make sure the condition would eventually be evaluated as `false`.

However, you might have written an unintentional infinite loop and you are not aware of it during programming. If your program executes and fails to respond or shows unexpected outputs repeatedly, it is probably executing an infinite loop. The following steps can help you to locate and solve the infinite loop:

1 If your program is composed of many loops and you are not sure which one has become an infinite loop, you should insert statements in the loop bodies so that it can display debug messages when a loop iteration starts or ends. Then, by displaying different debug messages, you can easily locate which loop has become an infinite loop.

2 You should now examine the loop condition of the infinite loop. You must find that the condition involves at least one variable.

3 For each involved variable, you should subsequently examine the statements that affect its value directly or indirectly.

4 Then, you should examine whether those statements can update the variables so that the loop condition will eventually become `false`.

5 If you still are not sure how the variables change and why the condition cannot eventually be made `false`, you should add statements to display debug messages in the loop body showing the contents of the variables.

For the above suspected program segment, the loop condition is `i < 100000` and the variable involved is just variable `i`. Then, the statement that will update the variable `i` is the increment statement `i++`. You should now ask yourself if the increment statement `i++` will update the variable `i` so that the condition `i < 100000` will eventually become `false`.

If you are not sure, you can add statements to the loop body displaying the change in the variable `i` in each iteration:

```
for (short i = 0; i < 100000; i++) {
  System.out.println("DEBUG:" + i);
  ......
}
```

Compile and execute the program. You will find that consecutive numbers starting from 0 are printed in the Command Prompt. However, you might find something strange in the number list:

```
DEBUG:32765
DEBUG:32766
DEBUG:32767
DEBUG:-32768
DEBUG:-32767
DEBUG:-32766
```

What has happened? The next number of `32767` after the statement `i++` is `–32768`!

This is explained by the fact that variable `i` is type `short` and its possible range is from `-32768` to `32767`. Therefore, no matter how the statements update the variable, its value will never be greater than or equal to `100000`. Therefore, the condition is determined to be `true` forever, and hence the infinite loop.

To make the above `for` loop work as expected, the modification required is to change the variable type from `short` to `int` because the range of `int` is much larger and includes `100000`. That is,

```
for (int i = 0; i < 100000; i++) {
  ......
}
```

The reason why a loop becomes an infinite loop varies; each can be a new challenge for you. It is a learning process for programmers, and you only gain more programming experience as you write more programs and debug more.

Maintaining proper entry and exit

In *Unit 1*, we discussed low-level programming languages and high-level programming languages. Low-level programming languages and some early high-level programming languages enable the flow of control in the programs to branch to and from any location. Therefore, the following sequence of executions in Figure 4.62 is possible.

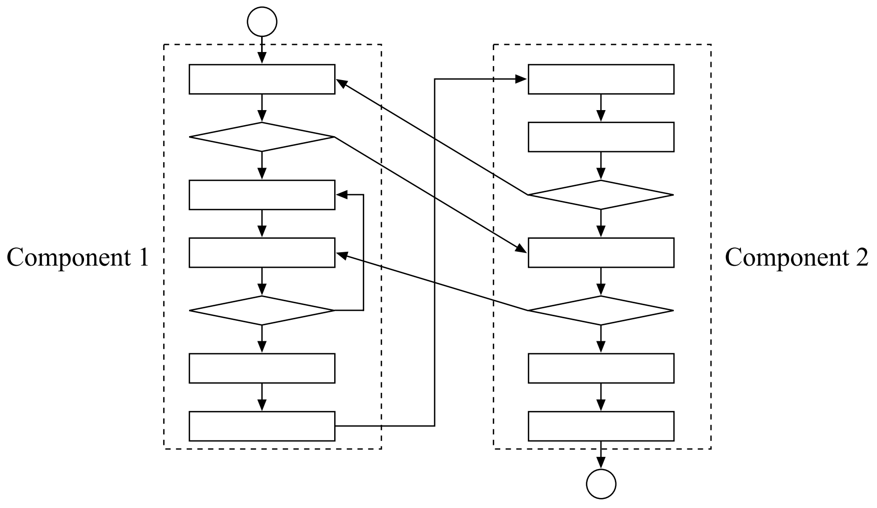


Figure 4.62 An example of bad flow of control

In Figure 4.62, component 1 and component 2 are two program segments for specific purposes, such as two methods in a class definition. You can see that it is hard to determine its actual behaviours based on the flowchart.

Such ambiguity is due to the fact that both components enable the flow of control to start and end at any point in the program segments, which was the scenario for programs written in the style of multiple entries and multiple exits. Such programs are a nightmare for the programmers who need to debug or maintain them.

More recently, programmers have been educated to provide each module or program segment with a single entry and a single exit. It makes writing, understanding, debugging and maintaining programs easier.

Modern programming languages, such as the Java programming language, are designed to enforce the single entry and single exit principle as much as possible. With respect to the Java programming language, there must be a single entry to a method or loop body, because it is not possible to branch the flow of control to the middle of a method body or loop body.

The only concern that may violate the single entry/single exit principle is that the language still enables multiple exits from a statement block with the use of the keyword `break` and `return`.

Consider the following method declaration:

```
public int factorial(int number) {
    if (number < 0) {
        return -1;
    }

    int result = 1;
    for (int i = number; i > 1; i--) {
        result *= i;
    }
    return result;
}
```

There are two `return` statements in the method. A reader must realize that if the value of `number` is less than `0`, the `for` loop and the following `return` statement are skipped. Furthermore, the underlying logic is not very clear.

Compare the above method with the following one. What do you think?

```
public int factorial(int number) {
    int result;

    if (number < 0) {
        result = -1;
    }
    else {
        result = 1;
        for (int i = number; i > 1; i--) {
            result *= i;
        }
    }

    return result;
}
```

You can see that the logic behind the method is much clearer. If the value of the variable `number` is less than `0`, the factorial of the negative integer is undefined and `-1` is assigned to the local variable `result`. Otherwise, the `else` part of the `if/else` statement is executed and the factorial is determined by the `for` loop and assigned to the variable `result`. After all that, the `result` is returned.

Another cause of multiple exits is the use of the `break` keyword. Please study the following program segment:

```
public String assignGrade(int mark) {
    String grade = "";
    switch (mark / 10) {
    ......
    case 9:
        if (mark % 10 < 5) {
            grade = "A-";
            break;
        }
        grade = "A";
        break;
    case 10:
        grade = "A+";
        break;
    }
    return grade;
}
```

For `case 9`, there is an `if` statement that assigns the grade `"A-"` for marks ranging from 90 to 94 and `"A+"` for marks ranging from 95 to 99. You should avoid such structures and use an `if/else` statement instead, such as:

```
public String assignGrade(int mark) {
    String grade = "";
    switch (mark / 10) {
    ......
    case 9:
        if (mark % 10 < 5) {
            grade = "A-";
        }
        else {
            grade = "A";
        }
        break;
    case 10:
        grade = "A+";
        break;
    }
    return grade;
}
```

This is clearer than the former one. However, such nested `switch/case` and `if/else` statements should be avoided unless it is really necessary to use them.

In the Java programming language, three types of statement — `break`, `continue` and `return` — might cause multiple exits. We have not discussed the use of the `continue` statement. The advanced features of control structures are introduced in *Unit 6*. At that time, we will revisit this issue and remind you about the single-entry single-exit principle again.

Debugging your programs

In this section, we discuss why a program can be successfully compiled but the output is not what we expected. Furthermore, we discuss some techniques to determine where and why the program goes wrong, so that you can carry out the remedial modifications. Such imperfections in programs are nicknamed ‘bugs’ in the programming community. Therefore, the process to remove the bugs in programs is called *debugging*.

The `while` loop in the `HelloSpeaker` class is quite understandable and easy to trace. However, this is not always so. For example, the following program segment can list the digits of a number one-by-one, starting from the least significant digit.

```
int n = number;
while (n > 0) {
    int digit = n % 10;
    System.out.println(digit);
    n /= 10;
}
```

Wait a moment! Can you figure out how it works just by reading the program segment? If you still cannot do so, don’t be nervous; the above program segment is discussed in detail soon.

First of all, the above program segment is put in a method called `listDigits()` of a `NumberLister` class as in Figure 4.63.

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

    // The method for listing the digits of the given number
    // starting from the least significant digit
    public void listDigits(int number) {
        //System.out.println("DEBUG:listDigits(" + number + ") started");

        // Declare a local variable for manipulation and
        // is assigned initially to be the value of parameter
        // number
        int remainDigits = number;

        // While the number with the remaining digits is
        // greater than zero
        while (remainDigits > 0) {
            //System.out.println("DEBUG:Loop body begin:");
            //System.out.println("DEBUG:remainDigits=" + remainDigits);

            // Get the least significant digit
            int digit = remainDigits % 10;
            // Print the digit
            System.out.println(digit);
            // Derive the number with remaining digits
            remainDigits /= 10;

            //System.out.println("DEBUG:Loop body end:");
            //System.out.println("DEBUG:remainDigits=" + remainDigits);
        }

        //System.out.println("DEBUG:listDigits() ended");
    }
}
```

Figure 4.63 NumberLister.java

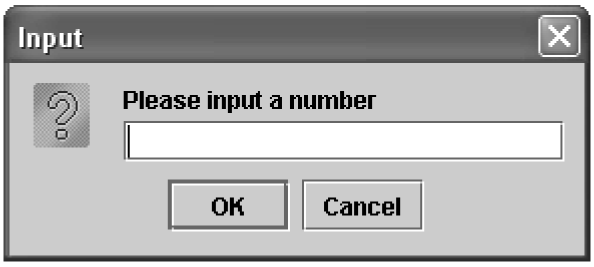
In the `NumberLister` class definition, you should find some strange comments in which the contents are like Java statements. They are to be used for debugging later.

The class definition to use a `NumberLister` object is written as in Figure 4.64.

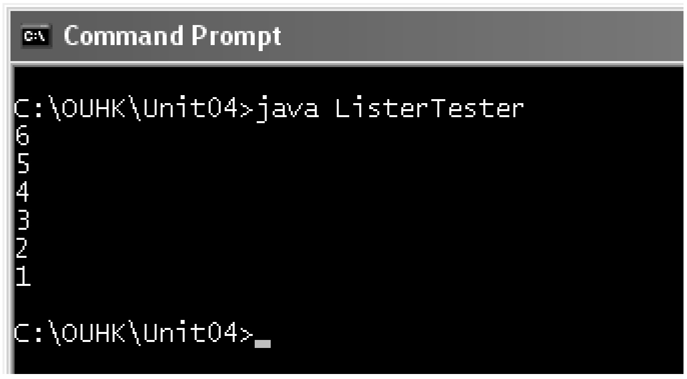
```
// Import statement for resolving the class ListerTester
import javax.swing.JOptionPane;

// The class definition of TestNumberLister for setting up the
// environment and test the NumberLister object
public class TestNumberLister {

    // The main executive method
    public static void main(String args[]) {
        // Create a NumberLister object and use variable lister
        // to refer to it
        NumberLister lister = new NumberLister();

        // Get a number from the user
        String inputNumber =
            JOptionPane.showInputDialog("Please input a number");

        // Derive the mark in numeric format from the input String
        int number = Integer.parseInt(inputNumber);

        // Request the NUmberLister object to list the digits
        // starting from the least significant digit
        lister.listDigits(number);

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

Figure 4.64 TestNumberLister.java

Compile the source code and execute the `TestNumberLister`. The following dialog appears:



If you enter an entry `"123456"` in the text field and complete the dialog by pressing <Enter> or clicking <OK>, the following message is shown at the Command Prompt:

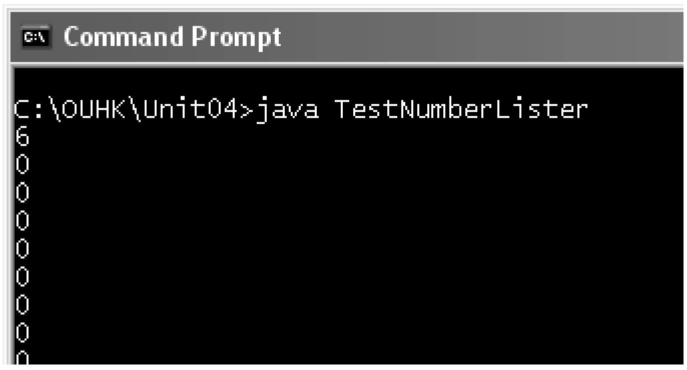


```
C:\OUHK\Unit04>java ListerTester
6
5
4
3
2
1

C:\OUHK\Unit04>_
```

If you type the `NumberLister` class definition in Notepad yourself but the method `listDigits()` is wrongly typed as

```
while (remainDigits > 0) {
  // Get the least significant digit
  int digit = remainDigits % 10;
  // Print the digit
  System.out.println(digit);
  // Derive the number with remaining digits
  remainDigits = 10;
}
```

you can still compile the `NumberLister` source code, as there is no syntax error. However, when you execute the `TestNumberLister` program and enter `"123456"` as the input, you get the following message from the Command Prompt:



```
C:\OUHK\Unit04>java TestNumberLister
6
0
0
0
0
0
0
0
0
```

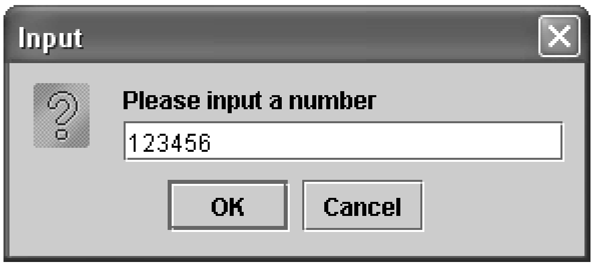
The first digit appears correctly as `6` but the program then displays the digit `0` repeatedly forever. This is the symptom that the program is executing an infinite loop (or is 'trapped' in an infinite loop).

If you cannot pinpoint the problem just by reading the program segment, you need a systematic way to locate the statement(s) that might have caused the problem.

Without other tools to help you, the handy way to determine what is going on with the program is to display particular messages whenever the execution encounters some milestones, such as the start and the end of a method execution. In the class definition of `NumberLister`, some statements for this purpose are included as comments. If you remove the double forward-slash symbol (`//`) of those comments, those statements will be executed as well. The program segment becomes:

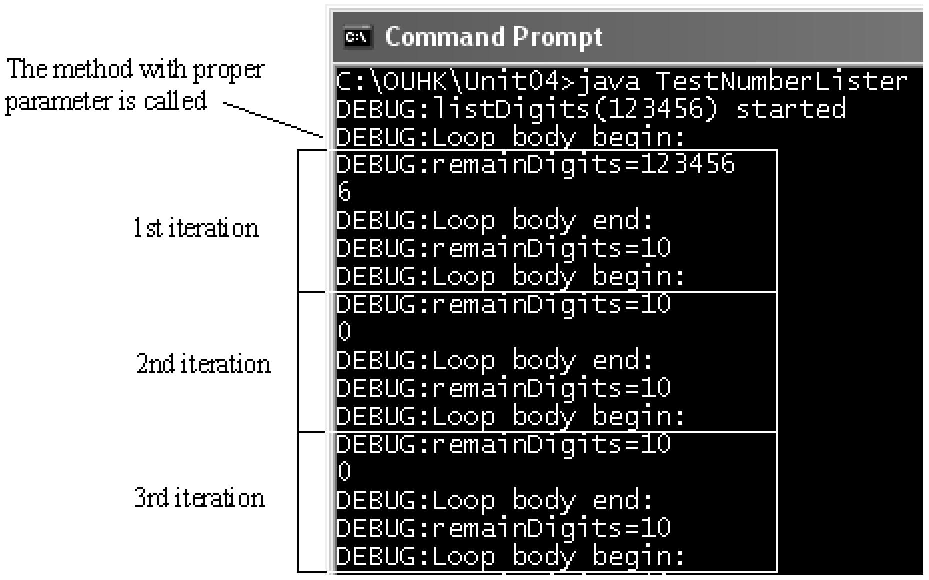
```
// The class definition of NumberLister
public class NumberLister {

    // The method for listing the digits of the given number
    // starting from the least significant digit
    public void listDigits(int number) {
        System.out.println("DEBUG:listDigits(" + number + ") started");

        // Declare a local variable for manipulation and
        // is assigned initially to be the value of parameter
        // number
        int remainDigits = number;

        // While the number with the remaining digits is
        // greater than zero
        while (remainDigits > 0) {
            System.out.println("DEBUG:Loop body begin:");
            System.out.println("DEBUG:remainDigits=" + remainDigits);

            // Get the least significant digit
            int digit = remainDigits % 10;
            // Print the digit
            System.out.println(digit);
            // Derive the number with remaining digits
            remainDigits = 10;

            System.out.println("DEBUG:Loop body end:");
            System.out.println("DEBUG:remainDigits=" + remainDigits);
        }

        System.out.println("DEBUG:listDigits() ended");
    }
}
```

Now, re-compile the source code and execute the `TestNumberLister` program again. The same dialog appears. Re-enter `"123456"` to fulfil the dialog function.



Then, the following messages are shown in the Command Prompt repeatedly and the message scrolls up rapidly. To stop the program, press <Ctrl-C>. If you can stop the program fast enough so that the Command Prompt can still keep those displayed messages, you can then use the vertical scroll bar of the Command Prompt to scroll through the messages displayed when the program was started. (If you are using Windows

95/98, its MS-DOS Prompt unfortunately does not provide the scroll bars.)



```
C:\OUHK\Unit04>java TestNumberLister
DEBUG:listDigits(123456) started
DEBUG:Loop body begin:
DEBUG:remainDigits=123456
6
DEBUG:Loop body end:
DEBUG:remainDigits=10
DEBUG:Loop body begin:
DEBUG:remainDigits=10
0
DEBUG:Loop body end:
DEBUG:remainDigits=10
DEBUG:Loop body begin:
DEBUG:remainDigits=10
0
DEBUG:Loop body end:
DEBUG:remainDigits=10
DEBUG:Loop body begin:
```

Now, you can study what was going on with the program using the messages shown on the screen.

In the displayed messages, those starting with the prefix `"DEBUG:"` are shown for debugging purposes. Otherwise, the messages are the actual output of the program. Based on the messages, it is clear that the method `listDigits()` with parameter `123456` is called and the changes of the variable `remainDigits` before and after each iteration are shown.

For the first iteration, the debug message suggests that value of the variable `remainDigits` was `123456` before the actual loop body was executed. Then, a digit `6` was derived from the expression

```
int digit = remainDigits % 10;
```

and is displayed on the screen by the following statement:

```
// Print the digit
System.out.println(digit);
```

Therefore, a line with a digit `6` is printed.

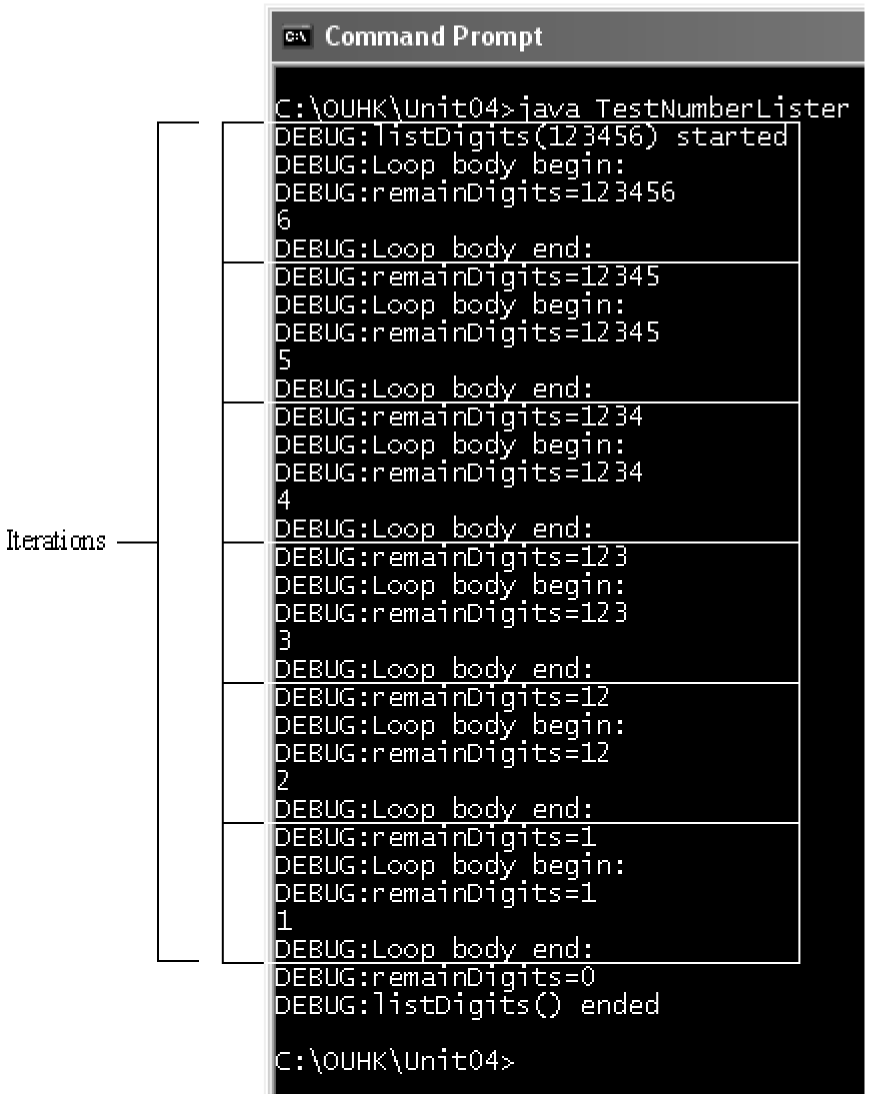
The subsequent debug message for the first iteration shows that the value of the variable `remainDigits` is `10` after the actual loop body is executed. Furthermore, its value is kept unchanged for the remaining iterations. Therefore, something must be wrong in updating the variable `remainDigits`, and you can then inspect the statement(s) that will update the variable. In the program segment, there is only one such statement:

```
// Derive the number with remaining digits
remainDigits = 10;
```

Therefore, you can be sure that the above statement must be wrong. By comparing the program listing presented in the unit and your program listing in the editor, you can determine that the above statement should be amended to:

```
    remainDigits /= 10;
```

Now, you can re-compile and execute the program again. By providing the same input "123456", you will have the following messages shown at the Command Prompt:



```
C:\OUHK\Unit04>java TestNumberLister
DEBUG:listDigits(123456) started
DEBUG:Loop body begin:
DEBUG:remainDigits=123456
6
DEBUG:Loop body end:
DEBUG:remainDigits=12345
DEBUG:Loop body begin:
DEBUG:remainDigits=12345
5
DEBUG:Loop body end:
DEBUG:remainDigits=1234
DEBUG:Loop body begin:
DEBUG:remainDigits=1234
4
DEBUG:Loop body end:
DEBUG:remainDigits=123
DEBUG:Loop body begin:
DEBUG:remainDigits=123
3
DEBUG:Loop body end:
DEBUG:remainDigits=12
DEBUG:Loop body begin:
DEBUG:remainDigits=12
2
DEBUG:Loop body end:
DEBUG:remainDigits=1
DEBUG:Loop body begin:
DEBUG:remainDigits=1
1
DEBUG:Loop body end:
DEBUG:remainDigits=0
DEBUG:listDigits() ended

C:\OUHK\Unit04>
```

You can easily understand the actual operations from the debug message. Table 4.6 summarizes the operations for your reference.

Table 4.6 Summary of actual operations while debugging the program

Iteration	Value of `remainDigits` before loop body	The digit obtained by `remainDigit % 10`	Value of `remainDigits` after the loop body (`remainDigit /= 10`)
1	`123456`	`6`	`12345`
2	`12345`	`5`	`1234`
3	`1234`	`4`	`123`
4	`123`	`3`	`12`
5	`12`	`2`	`1`
6	`1`	`1`	`0`

After the loop body is executed for the sixth iteration, the value of the variable `remainDigits` is `0.` Therefore, when the condition `remainDigit > 0` is tested before executing the loop body for the next iteration, the result is evaluated to be `false` and the loop is terminated.

You should now understand how the `while` loop in the method `listDigits()` can be used to list the digits in reverse order.

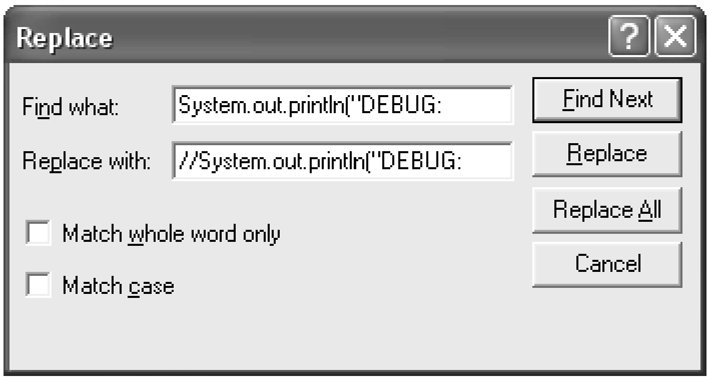
The program has been successfully debugged. You can now remove the statements for debugging purposes. You can remove the statements from the program one-by-one. If you think you will use these debug statements again, another option is to re-mark them as comments, which you can easily do using the editor's Replace function. For example, for the `NumberLister` class definition, you can use the editor to replace the text pattern

```
System.out.println("DEBUG:
```

with

```
//System.out.println("DEBUG:
```

For example:



Find what: System.out.println("DEBUG:

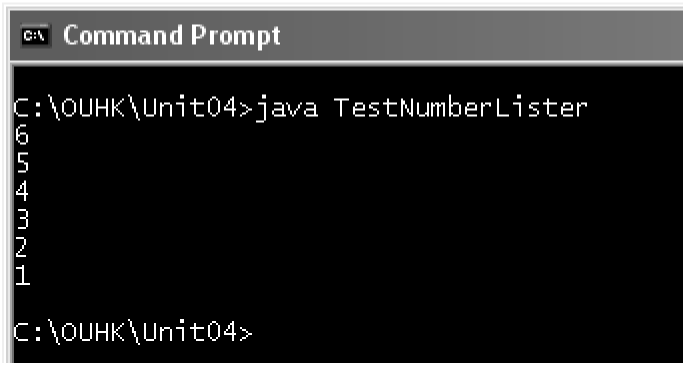
Replace with: //System.out.println("DEBUG:

☐ Match whole word only

☐ Match case

Then, you can mark all statements for debugging in a single operation and get the program listing in Figure 4.63. Similarly, you can restore the statements by doing a reverse Replace operation.

When you recompile and execute the program again, you will get the desired output messages as:



```
C:\OUHK\Unit04>java TestNumberLister
6
5
4
3
2
1

C:\OUHK\Unit04>
```

The following is a list of points that you may find useful in debugging the above program:

1 You should identify the events that will occur during the program execution so that you can trace the flow of control while the program is executing. Some typical events are, for example, the starts and ends of method executions or loop iterations.

2 Debug messages should be displayed while the program is executing. You can determine the message content format case by case. The baseline is that the debug messages provide sufficient information about the states you are interested in, and you can isolate the debug messages from the actual output of the program.

3 If possible, the statements for printing debug messages should be written so that they could be easily removed and restored.

If you add statements to display a debug message in a loop that originally had only one statement, you should make sure that parentheses are provided. For example, if the original program segment is

```
while (n < 10)
    n *= 2;
```

you might add statements for debugging such as:

```
while (n < 10)
    System.out.println("DEBUG:Loop-start");
    System.out.println("value of n = " + n);
    n *= 2;
    System.out.println("DEBUG:Loop-end");
    System.out.println("value of n = " + n);
```

Then, the loop body of the `while` loop contains the following statement only,

```
    System.out.println("DEBUG:Loop-start");
```

and the program segment is considered to be:

```
while (n < 10) {
    System.out.println("DEBUG:Loop-start");
}
System.out.println("value of n = " + n);
n *= 2;
System.out.println("DEBUG:Loop-end");
System.out.println("value of n = " + n);
```

This is the reason why we always recommend you use the pair of curly brackets for the loop body even it has only one statement.

Summary

This has been a rather long unit in which two basic building blocks in writing programs were discussed — branching and looping.

A program is a sequence of statements (or instructions) that are executed by the computer. We used the term 'flow of control' to denote the execution order of the statements. The program might need to alter the flow of control so that it can do some non-trivial tasks. Therefore, we need relational operators, such as greater than (`>`), less than (`<`) and so on. They can be used to construct the conditions so that the flow of control can be altered.

The Java programming language supports two types of branching statement — `if/else` statements and `switch/case` statements. The `if/else` statement enables two-way branching based on a condition result of either `true` or `false`. The `switch/case` structure implements multiple branching by checking if a variable equals one of the values specified. The `if/else` statement is good for condition flexibility, whereas the `switch/case` statement is an elegant structure for multiple equality checking.

Looping structures enable a statement or a block of statements to be executed repeatedly. A looping structure consists of a *loop body* to be executed and a loop *condition* to determine whether the loop has to continue. It usually has an *initialization* part that sets up the variables and an *update* part that might affect the loop condition. The three looping structures provided by the Java programming language support different combinations and arrangements of the above four parts (*initialization*, *condition*, *loop body*, and *update*).

To determine which looping structure is suitable for your current tasks, you should first determine the minimum number of times the loop body is executed (or iterations). The minimum number of iterations is zero for `while` loops and `for` loops, whereas the loop body of a `do/while` loop must be executed at least once. If you are wondering whether to use a `while` loop or a `for` loop, as a usual programming practice you can choose a `for` loop for a predetermined number of iterations and a `while` loop for the rest.

Programming and debugging are two processes that will live together forever. While you are writing a program, you need to debug it as well. Compile-time errors are relatively easy to resolve and the compiler will detect them for you. However, runtime errors are usually a result of imperfect designs or logic and can be much more difficult to detect and resolve. This unit discussed an approach to handle them by adding statements that display the current values of some variables to the program segment, which then should identify what is causing the problem. By studying such debug messages, the problem is pinpointed and solved.

We discussed some elementary uses of the control structures in the language. In *Unit 6*, we will revisit control structures. You’ll learn their advanced features then.

Java API reference

`java.lang.Double`

Fields

Name	`MIN_VALUE`
Declaration	`public static double MIN_VALUE`
Usages	The minimum possible value support by type `double`
Example	`double minValue = Double.MIN_VALUE;`

Name	`MAX_VALUE`
Declaration	`public static double MAX_VALUE`
Usages	The maximum possible value support by type `double`
Example	`double maxValue = Double.MAX_VALUE;`

Methods

Name	`parseDouble(String)`
Declaration	`public static double parseDouble(String s)`
Usages	Converts the given `String` object into the corresponding value of type `double`.
Example	`double d = Double.parseDouble("1.2");`

`java.lang.Integer`

Fields

Name	`MIN_VALUE`
Declaration	`public static int MIN_VALUE`
Usages	The minimum possible value support by type `int`
Example	`int minValue = Integer.MIN_VALUE;`

Name	`MAX_VALUE`
Declaration	`public static int parseInt(String s)`
Usages	The maximum possible value support by type `int`
Example	`int maxValue = Integer.MAX_VALUE;`

Methods

Name	`ParseInt(String)`
Declaration	`public static int parseInt(String s)`
Usages	Converts the given `String` object into the corresponding value of type `int`.
Example	`int i = Integer.parseInt("123");`

`java.lang.Math`

Methods

Name	`random()`
Declaration	`public static double random()`
Usages	Return a random number (y) of type `double` and its range is $0 \leq y < 1$.
Example	`double d = Math.random();`

Name	`sqrt(double n)`
Declaration	`public static double sqrt(double n)`
Usages	Return the square root of the given number.
Example	`double sr = Math.sqrt(2.0);`

`javax.swing.JOptionPane`

Methods

Name	`showMessageDialog(Component c, String message)`
Declaration	`public static void showMessageDialog(Component c, String message)`
Usages	Show a graphical dialog to the user with the message specified by the second parameter. If the dialog is used with a master window, the reference of the window is presented to the method via the first parameter. If the dialog is used by itself, the first parameter is set to `null`.
Example	`JOptionPane.showMessageDialog(null, "This is the message");`

Name	`showInputDialog(String message)`
Declaration	`public static String showInputDialog(String message)`
Usages	Show a dialog with a text field to the user so that the user can enter a text in the text field. The input text is returned as a `String` object if the user presses <Enter> or clicks <OK>. If the user clicks <Cancel>, it returns a value of `null`.
Example	`String input = JOptionPane.showInputDialog( "Please input something");`

Suggested answers to self-test questions

Self-test 4.1

1 a
```
5 + 4 / 3 * 2
⇒ 5 + 1 * 2
⇒ 5 + 2
⇒ 7
```

b
```
(3 + 2) * 4
⇒ 5 * 4
⇒ 20
```

c
```
33 * 3 < 99
⇒ 99 < 99
⇒ false
```

d
```
 (4 + 3) * 2 != 12
⇒ 7 * 2 != 12
⇒ 14 != 12
⇒ true
```

2 a `(x1 – x2 ) / (y1 – y2)`

b `m * c * c`

c `u * t + a * t * t / 2`

3 a The expression `n %= 3` is interpreted as `n = n % 3`. As the value of variable `n` is `10`, the result of the expression on the right-hand side of the assignment operator (=) is `1`, and that value is assigned to variable `n`. Therefore, the value of variable `n` after the program segment is `1`.

b The expression `n *= n - 5` is interpreted as `n = n * (n - 5)`. If the value of variable `n` is `15`, the result of the expression on the right-hand side is:

```
  15 * (15 – 5)
⇒ 15 * 10
⇒ 150
```

Therefore, the value of `150` is assigned to variable `n`. As a result, the value of variable `n` after executing the program segment is `150`.

Self-test 4.2

1 The definition of the class `TicketCounter` is modified as (the modified lines are in **boldface**):

```
// The class definition of a ticket counter
public class TicketCounter {
    private int reading;              // The ticket counter reading

    // To increase the reading by one
    public void increase() {
        reading = reading + 1;
        if (reading > 9999) {         // verify whether reading > 9999
            reading = 0;              // if true, reset reading to 0
        }
    }

    // To increase the reading specified by the parameter, amount
    public void increaseByAmount(int amount) {
        reading = reading + amount;
        if (reading > 9999) {         // verify whether reading > 9999
            reading = reading % 10000; // if true, rectify reading
        }
    }

    // To get the ticket counter reading
    public int getReading() {
        return reading;
    }

    // To set a reading to the ticket counter reading
    public void setReading(int theReading) {
        // Verify the parameter to be non-negative before updating
        // the attribute reading
        if (theReading >= 0) {
            reading = theReading % 10000;
        }
    }
}
```

2 The program can be compiled successfully because the program is considered to be:

```
// To increase the reading by one
void increase() {
    reading = reading + 1;
    if (reading > 999)
        ;
    {
        reading = 0;
    }
}
```

In the Java programming language, a single semi-colon `;` is considered an empty statement. In the above program segment, the single semi-colon is therefore the statement to be executed if the condition `reading >`

999 is true and is skipped if the condition is false, which means nothing is executed regardless of whether the condition is true or false. Afterwards, the statement block

```
    {
        reading = 0;
    }
```

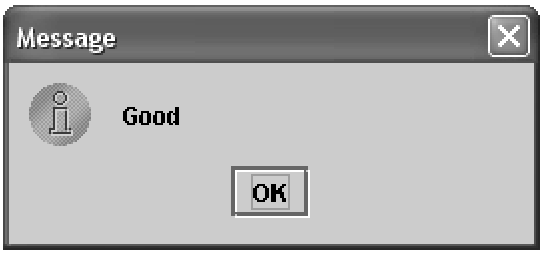
is executed anyway. As a result, the value of the reading attribute of the object is updated to 0 if the method is executed.

Therefore, the following output is displayed at the Command Prompt:

```
The initial reading is: 124
The reading after 'increase' is: 0
The reading is set to be 997
The reading after 'increaseByAmount' is: 2
```

Self-test 4.3

The output of the program is:



Good

According to the method assignGreeting():

```
    // Determine the greeting based on the given hour
    public String assignGreeting(int hour) {
        // A local variable for storing the result temporarily
        String greeting = " ";

        // Check whether the hour is in the morning
        if (hour >= 0) {
            greeting = "morning";
        }

        // Check whether the hour is in the afternoon
        if (hour >= 12) {
            greeting = "afternoon";
        }

        // Check whether the hour is in the evening
        if (hour >= 18) {
            greeting = "evening";
        }

        // Return the result to the caller
        return greeting;
    }
```

As the given hour is `-1`, all condition results are `false` and the local variable `greeting` is not updated to `"morning"`, `"afternoon"` or `"evening"`. Therefore, the local variable `greeting` keeps referring to a `String` object with empty content. As a result, this `String` object is appended to the string `"Good"`. This yields the truncated output

`"Good "`, which is finally shown in the output dialog.

Self-test 4.4

1 The `switch/case` structure in the `CalendarExpert3` has a `default` part, whereas those in `CalendarExpert1` and `CalendarExpert2` do not. Therefore, for `CalendarExpert3` if the parameter `month` does not equal `2`, `4`, `6`, `9` or `11`, the local variable `days` will be updated to `31`. Therefore, if the value of the parameter `month` is set to `-1`, the local variable `days` is also updated to `31`. As a result, the method returns `31` even if the parameter is given as `-1`.

For the `switch/case` structures in `CalendarExpert1` and `CalendarExpert2`, if the value of parameter `month` does not fall within the range from `1` to `12` inclusive, no statement is executed at all, the local variable `days` is kept unchanged and the return value of the method is `0`.

2 **NumberTranslator.java**

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

    // The method for getting the word from a number
    public String assignWordForNumber(int number) {
        // Declare a local variable for storing the word for
        // the number
        String word = " ";

        // Determine the word based on the number
        switch (number) {
        case 1: word = "one"; break;
        case 2: word = "two"; break;
        case 3: word = "three"; break;
        case 4: word = "four"; break;
        case 5: word = "five"; break;
        case 6: word = "six"; break;
        case 7: word = "seven"; break;
        case 8: word = "eight"; break;
        case 9: word = "nine"; break;
        case 10: word = "ten"; break;
        }

        // Return the word determined
        return word;
    }
}
```

TestTranslator.java

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

// The class definition of TestTranslator for setting up the
// environment and test the NumberTranslator object
public class TestTranslator {

    // The main executive method
    public static void main(String args[]) {
        // Create a NumberTranslator object and use variable translator
        // to refer to it
        NumberTranslator translator= new NumberTranslator ();

        // Get the number from the user
        String inputNumber =
            JOptionPane.showInputDialog("Input the number");

        // Derive the number in numeric format from the input String
        int number = Integer.parseInt(inputNumber);

        // Get the grade using the NumberTranslator object
        String word = translator.assignWordForNumber(number);

        // Show the number of days in the month to user
        JOptionPane.showMessageDialog(null,
            "The word for " + inputNumber + " is " + word);

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

Self-test 4.5

1 Running the program `FactorialTester` with input `"16"` and `"17"` gives results `2004189184` and `-288522240` respectively. The formula of factorial suggests that a factorial must be positive but the value of 17! is negative because the value of 17! is out of the range of type `int`; hence the strange result.

Therefore, you should be alert to the fact that the variables you use in your program must support the range of data being manipulated.

2 Based on the class definition of the `FactorialCalculator` given in Figure 4.48, the output for the input number 0 is 0, which is incorrect because the value of 0!, by definition, is 1.

The reason for such a problem is the loop design. First of all, let’s revisit the program segment in the method `factorial()`.

```
// The method for calculating the factorial of a given
// number via the parameter
public int factorial(int number) {
    // Declare a local variable for storing the intermediate
    // value during iteration, and its value is initialized to
    // be 1
    int result = 1;
    do {
        // The intermediate result is multiplied by the
        // the current value of variable number
        result *= number;

        // Decrease the value of number by one
        number--;

    // While the number is greater than one,
    // repeat the loop body
    } while (number > 1);

    // Return the finalized result
    return result;
}
```

The local variable `result` is initialized to `1` and the given the parameter `number` is `0`. The `do/while` loop is executed at least once. Therefore, the local variable `result` is updated to `0` by `result *= number`. Then, the condition is checked to be `false`, and the loop is terminated. The value of the local variable `result` is returned.

Let's try to change the method from using a `do/while` loop to a `while` loop. The method becomes:

```
// The method for calculating the factorial of a given
// number via the parameter
public int factorial(int number) {
    // Declare a local variable for storing the intermediate
    // value during iteration, and its value is initialized to
    // be 1
    int result = 1;

    // While the number is greater than one,
    // repeat the loop body
    while (number > 1) {
        // The intermediate result is multiplied by the
        // the current value of variable number
        result *= number;

        // Decrease the value of number by one
        number--;

    // While the number is greater than one,
    // repeat the loop body
    }

    // Return the finalized result
    return result;
}
```

If the value of the parameter `number` is `0`, the `while` loop is skipped completely and the value of the local variable `result`, `1`, is returned.

3 **PowerCalculator.java**

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

    // The method for calculating the power of a given
    // number via the parameter
    public double power(double value, int toPower) {
        // Declare a local variable for storing the intermediate
        // value during iteration, and its value is initialized to
        // be 1
        double result = 1.0;

        // While the toPower is greater than 1
        while (toPower >= 1) {
            // The intermediate result is multiplied by the value
            result *= value;

            // Decrease the value of toPower by one
            toPower--;

        }

        // Return the finalized result
        return result;
    }
}
```

TestPowerCalculator.java

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

// The class definition of TestPowerCalculator for setting up the
// environment and test the PowerCalculator object
public class TestPowerCalculator {

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

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

        // Get a number from the user
        String inputNumber =
            JOptionPane.showInputDialog("Input the number");
        double number = Double.parseDouble(inputNumber);

        String inputPower =
            JOptionPane.showInputDialog("Input the power");
        int power = Integer.parseInt(inputPower);

        // Calculate the power of the number
        double result = calculator.power(number, power);

        // Show the result to user
        JOptionPane.showMessageDialog(null,
            "The result is = " + result);

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

4 If the program is typed by mistake as

```
int n = 1;
    n *= 2;
while (n < 10);
```

it is actually considered to be:

```
int n = 1;
n *= 2;
while (n < 10)
    ;
```

As a result, the program segment can still be successfully compiled, because the semi-colon is considered an empty statement.

The interpretation is that the first statement declares a variable `n` and initializes it to be `1`. This variable is further updated to be `2` by `n *= 2`. Finally, the while loop repeatedly checks the condition `n < 10` which is `true` but there is no statement to update the value of variable `n`. Therefore, the loop will execute forever and hence is an infinite loop.

Self-test 4.6

1 **NumberAdder.java**

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

    // The method for calculating the sum of number from
    // 1 to the given limit
    public int findSumOfAll(int limit) {
        // Declare a local variable for storing the intermediate
        // value during iteration, and its value is initialized to
        // be 0
        int total = 0;

        // The loop for adding numbers from 1 to the given limit
        for (int i=1; i <= limit; i++) {
            total += i;
        }

        // Return the sum of all
        return total;
    }

    // The method for calculating the sum of odd number from
    // 1 to the given limit
    public int findSumOfAllOdd(int limit) {
        // Declare a local variable for storing the intermediate
        // value during iteration, and its value is initialized to
        // be 0
        int total = 0;

        // The loop for adding odd numbers from 1 to the given limit
        for (int i=1; i <= limit; i += 2) {
            total += i;
        }

        // Return the sum of all
        return total;
    }

    // The method for calculating the sum of even number from
    // 1 to the given limit
    public int findSumOfAllEven(int limit) {
        // Declare a local variable for storing the intermediate
        // value during iteration, and its value is initialized to
        // be 0
        int total = 0;

        // The loop for adding even numbers from 1 to the given limit
        for (int i=2; i <= limit; i += 2) {
            total += i;
        }

        // Return the sum of all
        return total;
    }
}
```

TestNumberAdder.java

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

// The class definition of TestNumberAdder for setting up the
// environment and test the NumberAdder object
public class TestNumberAdder {

    // The main executive method
    public static void main(String args[]) {
        // Create a NumberAdder object and use variable adder
        // to refer to it
        NumberAdder adder = new NumberAdder();

        // Get the limit from the user
        String inputLimit =
            JOptionPane.showInputDialog("Input a limit");

        // Derive the number in numeric format from the input String
        int limit = Integer.parseInt(inputLimit);

        // Request the NumberAdder object to calculate the sums of
        // all numbers from 1 to the limit
        int sum = adder.findSumOfAll(limit);
        int sumOdd = adder.findSumOfAllOdd(limit);
        int sumEven = adder.findSumOfAllEven(limit);

        // Show the sum to the user
        JOptionPane.showMessageDialog(null,
            "The sum of all is = " + sum +
            "\nThe sum of all odd is = " + sumOdd +
            "\nThe sum of all even is = " + sumEven);

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

2 **MarkSixChooser.java**

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

    // The method generates six random number from 1 to 45
    // and show them
    public void showNumbers() {

        // The loop for adding numbers from 1 to the given limit
        for (int i=0; i < 6; i++) {
            // Get a random number from 1 to 45
            int number = (int) (Math.random() * 45) + 1;
            // Display the number
            System.out.println(number);
        }

    }
}
```

TestMarkSix.java

```
// The class definition of TestMarkSix setting up the
// environment and test the MarkSixChooser object
public class TestMarkSix {

    // The main executive method
    public static void main(String args[]) {
        // Create a MarkSixChooser object and use variable chooser
        // to refer to it
        MarkSixChooser chooser = new MarkSixChooser();

        // Show the numbers
        chooser.showNumbers();
    }
}
```

When you execute the program, six random numbers are drawn and displayed. There may be duplicated numbers because all six numbers are independently drawn. We discuss how to handle this issue in *Unit 5*.

Self-test 4.7

1 **GreetingSpeaker.java**

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

    // The method display greetings to the number of times given
    // via the parameter
    public void sayGreetings(int total) {

        // The loop for showing the greetings
        for (int i=0; i < total; i++) {
            // The index variable i is used to determine which
            // greeting to be displayed by remainder operator
            switch (i % 3) {
            case 0:
                System.out.println("Good morning");
                break;
            case 1:
                System.out.println("Good afternoon");
                break;
            case 2:
                System.out.println("Good evening");
            }
        }

    }
}
```

TestGreetingSpeaker.java

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

// The class definition of TestGreetingSpeaker for setting up the
// environment and test the GreetingSpeaker object
public class TestGreetingSpeaker {

    // The main executive method
    public static void main(String args[]) {
        // Create a GreetingSpeaker object and use variable speaker
        // to refer to it
        GreetingSpeaker speaker = new GreetingSpeaker ();

        // Get the total from the user
        String inputTotal =
            JOptionPane.showInputDialog("Input number of greetings");

        // Derive the number in numeric format from the input String
        int total = Integer.parseInt(inputTotal);

        // Request the GreetingSpeaker object to show the greetings
        speaker.sayGreetings(total);


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