2. Flow Control

2.1 Introduction to Flow Control

The Java language provides a number of constructs that enable the developer to control the sequence of execution of Java statements. Chapter Two provides examples of how these constructs are used to control the flow of execution through a block of code that is typically contained in the body of a method.

2.2 Sequential Flow

Sequential flow of execution of statements is the execution of Java source code in a statement-by-statement sequence in the order in which they are written, with no conditions. Most of the examples of methods that are discussed in previous chapters exhibit sequential flow. In general terms, such a method is written as follows.

```
public void someMethod( ) {

    // first statement to execute
    // second statement to execute
    // third statement to execute
    // and so on, to the final statement
    // final statement to execute

} // end of method definition
```

A number of the **main** methods, presented in previous chapters, are structured in this sequential way in order to satisfy straightforward testing criteria.

2.3 Conditional Flow

While sequential flow is useful, it is likely to be highly restrictive in terms of its logic. Executing statements *conditionally* gives the developer a mechanism to control the flow of execution in order to repeat the execution of one or more statements or change the normal, sequential flow of control. Constructs for conditional flow control in Java are very similar to those provided by other programming languages. Table 2.1 on the next page identifies the flow control constructs provided by the Java language.

Statement Type	Key words
Decision	if … then
Decision	if … else
Decision	switch … case
Loop	for
Loop	while
Loop	do … while
Branching	break: labelled and unlabelled form
Branching	continue: labelled and unlabelled form
Exception handling (see Chapter Four)	try … catch

Table 2.1 Flow control constructs

The sub-sections that follow show, by example, how these constructs are used.

2.4 Making Decisions

Using a decision-making construct allows the developer to execute a block of code *only* if a condition is true. The sub-sections that follow illustrate how decision-making constructs are used.

2.4.1 The if ... then Construct

The **if ... then** construct is the most basic of the decision-making constructs provided by the Java language. If a condition is true, the block of code is executed: otherwise, control skips to the first statement after the **if** block. The following code snippet illustrates a simple use of the **if ... then** construct.

```
// assume that the value of age has been entered via the keyboard
if ( age >= 18 ) // the if condition is placed between ( and )
{ // start of if block
        System.out.println( “You can drink legally.” ); // the then clause
} // end of if block
// execute the next statement
System.out.println( “The rest of the programme is next.” );
```

When the code snippet is run (in a **main** method), the output when **age = 20** is:

```
You can drink legally.
The rest of the programme is next.
```

and when **age = 17**, the output is:

```
The rest of the programme is next.
```

In some programming languages, the word ‘then’ is included in the **then** clause. As the code snippet above shows, this is not the case in Java.

An example taken from the themed application shows an **if ... then** construct in action in one of the methods of the **Member** class. The method adds a member to the array of members only if there is room in the array of (arbitrary) size 6.

```
/**
* This method adds a member if there is room in the array of members called members.
* @param fName  The member's first name.
* @param lName  The member's last name.
* @param uName  The member's user name.
* @param pWord  The member's password.
*/
public void addMember( String fName, String lName, String uName, String  pWord ) {

        if( noOfMembers < 6 )
        {
                members[ noOfMembers ] = new Member( fName,
                        lName, uName, pWord );
```

```
            System.out.println( "The member has been added." );
            // Increment the number of members.
            noOfMembers++;
        }
        System.out.println( "No room for another member." );

    } // End of addMember.
```

If there is no room in the array because **noOfMembers** is equal to or greater than 6, control skips to the print statement that outputs the message “No room for another member.”

2.4.2 The if … else Construct

The **if ... else** construct (sometimes known as the **if ... then ... else** construct) provides an alternative path of execution if the **if** condition evaluates to false. Figure 2.1 illustrates, diagrammatically, the logic of the **if ... else** construct.

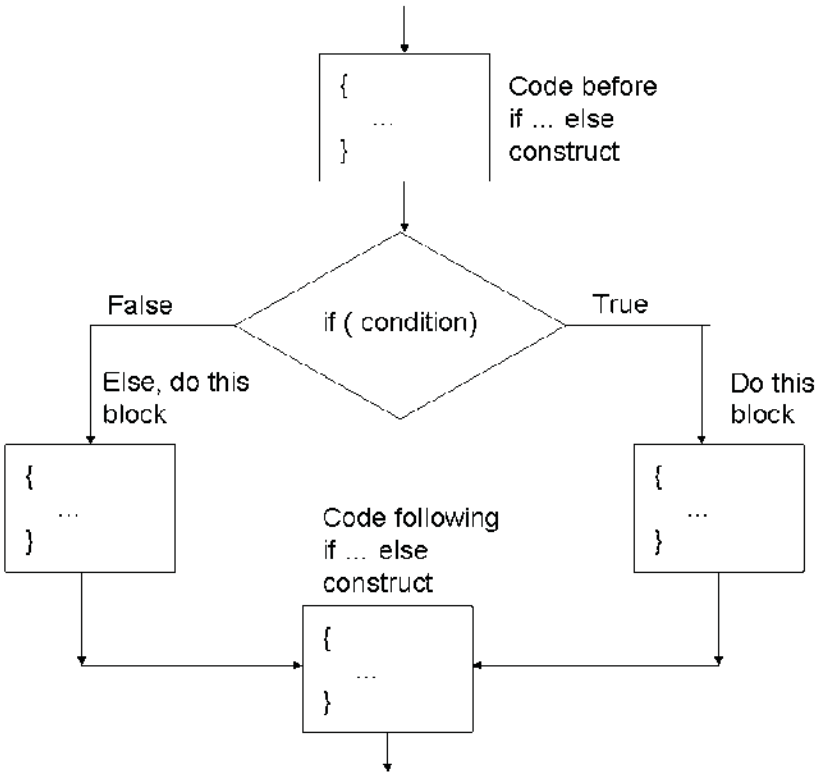

Figure 2.1 The logic of the **if … else** construct

Flow of control enters the **if** clause and the **if** condition is tested. The result of evaluating the **if** condition returns either **true** or **false** and one or other of the paths of execution are followed depending on this value. The **else** block is executed if the **if** condition is **false**.

The next code snippet illustrates a simple use of the **if ... else** construct by modifying the first code snippet in Section 2.4.1.

```
if ( age >= 18 ) // the if condition
{ // start of if block
        System.out.println( "You can drink legally." ); // the then clause
} // end of if block
else
{ // start of else block
        System.out.println( "You are too young to drink alcohol!" );
} // end of else block
// execute the next statement
System.out.println( "The rest of the programme is next." );
```

When the code snippet is run (in a `main` method), the output when `age = 20` is:

```
You can drink legally.
The rest of the programme is next.
```

and when `age = 17`, the output is:

```
You are too young to drink alcohol!
The rest of the programme is next.
```

Another example taken from the themed application shows an **if ... else** construct in action in another of the methods of the **Member** class. The **setCard** method is used to associate a member of the Media Store with a virtual membership card. Each member may have up to two cards, so the method checks whether another card can be allocated to a member.

```
/**
* This method gives a card to a member by adding it to the member's array of (two) cards.
* The array has the identifier cards.
* @return result  A boolean value to state whether the addition of a card is possible.
* @param card  A parameter of the MembershipCard type.
*/
public boolean setCard( MembershipCard card ) {

        // declare a local variable
        boolean result = true;
        // noOfCards is the number of cards allocated to the member
        if ( noOfCards < cards.length ) {
                cards[ noOfCards ] = card;
                noOfCards++;
        }
        else {
                System.out.println( "No more cards allowed for this member." );
                result = false;
        }
        return result;

} // End of setCard.
```

The **if ... else** construct in the method is used to return either **true** or **false**, depending upon the result of evaluating the **if** condition that determined whether or not the member has fewer than two cards.

2.4.3 Compound if ... else Constructs

There is another form of the **else** part of the **if .. else** construct: **else ... if.** This form of compound or cascading construct executes a code block depending on the evaluation of an **if** condition immediately after the initial **if** condition. The compound **if ... else** construct is illustrated diagrammatically in Figure 2.2 below.

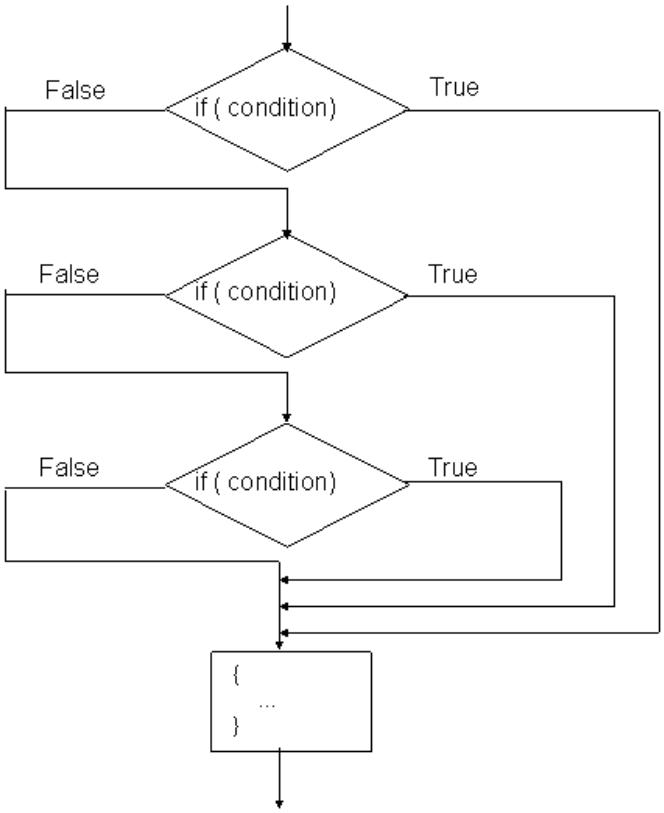

Figure 2.2 The logic of the compound **if … else** construct

The figure shows that any number of **else ... if** statements can follow the initial **if** statement.

The example on the next page illustrates how the **if .. else** construct is used to identify the classification for degrees awarded by universities in the United Kingdom, based on the average mark achieved in the final year.

```
// declare two local variables
int average  = 0;
String result = null;
if( average >= 70 )
{
        result = "First Class";
}
else if( average >= 60 )
{
        result = "Upper Second";
}
else if( average >= 50 )
{
        result = "Lower Second";
}
else if( average  >= 40 )
{
        result = "Pass";
}
else
{
        result = "You are going to have to tell your mother about this!";
}
System.out.println( "Your result is: " + result);
```

Running the code with an average of 30 % produces the following output:

Your result is: You are going to have to tell your mother about this!

and with an average of 65 %, the output is as follows:

Your result is: Upper Second

When the value of average is equal to 65, this satisfies more than one of the **else ... if** statements in the code above. However, the output confirms that the first time that a condition is met – when average >= 60 – control passes out of the initial **if** statement without evaluating the remaining conditions. When a condition is met in the code above, the output shows that control skips to the first statement after the initial **if** statement, i.e. to the statement

System.out.println("Your result is: " + result);

It is worthwhile alerting learners to the use of braces in compound **else ... if** constructs. Care must be taken when coding compound **else .. if** constructs due to the number of pairs of brackets involved: a common error is to omit one or more of these brackets. In cases where there is only one statement in an **if** block, it is good practice to include braces – as shown in the example above – in anticipation of **if** blocks that include more than one statement.

The final example in this sub-section shows a compound **else ... if** construct in action in the **Member** class of the themed application. The method scans the array of (virtual) cards held by a member and outputs some information that is stored against each card. (**for** loops are discussed in a later section of this chapter.)

```
/** This method scans the array of cards in a for loop.  */
public void getDetialsOfCards( ) {

    // Declare a local variable.
    MembershipCard card = null;
    // note the use of the instanceof operator
    for ( int i = 0; i < noOfCards; i++ )
    {
        if ( cards[ i ] instanceof DvdMembershipCard )
    {
            card = cards[ i ];
            System.out.println( "This is a DVD card with " + getNoOnLoan( )
                + " DVDs currently on loan." );
    } else if ( cards[ i ] instanceof GameMembershipCard )
    {
            card = cards[ i ];
            System.out.println( "This is a games card with " +
                getNoOnLoan( ) + " CDs currently on loan" );
    } else
    {
```

```
                    System.out.println( "Neither type of card." );
            }
            } // End of for loop.

    } // End of getDetailsOfCards.
```

2.4.4 Nested if Statements

As an alternative to compound **if** statements, as described in sub-section 2.4.3, **if** statements can be nested if the method demands this kind of logic. The simple example of nesting **if** statements shown next is a variant of the first example in sub-section 2.4.3.

```
int average = 65;
String result = null;
String course = "Java";
if ( course == "Java" )
{ // start of outer if
        if( average >= 70 ) // start of inner if
        {
                result = "First Class";
        }
        else if( average >= 60 )
        {
                result = "Upper Second";
        }
        else if( average >= 50 )
        {
                result = "Lower Second";
        }
        else if( average >= 40 )
        {
                result = "Pass";
        }
        else
        {
                result = "You are going to have to tell your mother about this!";
        }
        System.out.println( "Your result is: " + result);
} // end of outer if
System.out.println( "No more results are available." );
```

When the code is run, the output is

```
Your result is: Upper Second
No more results are available.
```

When the next version is run, the output is

No more results are available.

```
int average = 65;
String result = null;
String course = "C++";
if ( course == "Java" )
{ // start of outer if
        if( average >= 70 ) // start of inner if
        {
                result = "First Class";
        }
        else if( average >= 60 )
        {
                result = "Upper Second";
        }
        else if( average >= 50 )
        {
                result = "Lower Second";
        }
        else if( average >= 40 )
        {
                result = "Pass";
        }
        else
        {
                result = "You are going to have to tell your mother about this!";
        }
        System.out.println( "Your result is: " + result);
} // end of outer if
System.out.println( "No more results are available." );
```

Care should be taken when using any of the variants of the **if** statement to ensure and test that the required logic is implemented in the construct. It is often helpful to draw a diagram of the logic required, in order to help decide which variant to use to meet specific requirements.

2.4.5 The Conditional Operator

Java provides a ternary, conditional operator **? :** that is a compact version of an **if ... else** statement. The operator takes three operands: the first is a boolean condition; the second is the result if the condition is **true**; the third is the result if the condition is **false**.

Let us recall a previous example from this chapter.

```
if ( age >= 18 ) // the if condition
{ // start of if block
        System.out.println( “You can drink legally.” ); // the then clause
} // end of if block
else
{ // start of else block
        System.out.println( “You are too young to drink alcohol!” );
} // end of else block
// execute the next statement
System.out.println( “The rest of the programme is next.” );
```

The logic of the **if .. else** statement can be re-written as follows, using Java’s ternary operator.

```
int age = 21;
System.out.println( age >= 18 ? "Old enough to drink" : "Too young to drink" );
System.out.println( "The rest of the programme is next." );
```

and produces the following output when run:

```
Old enough to drink
The rest of the programme is next.
```

The next and final sub-section in the category of decision-making constructs describes the **switch ... case** construct.

2.4.6 The switch ... case Construct

The **switch ... case** construct is an alternative to compound **if** statements if the condition is an evaluation of an integer expression. As such, it is often easier to code than compound **if** statements in that is it less prone to errors such as the omission of brackets. The **switch ... case** construct is illustrated diagrammatically in Figure 2.3 on the next page.

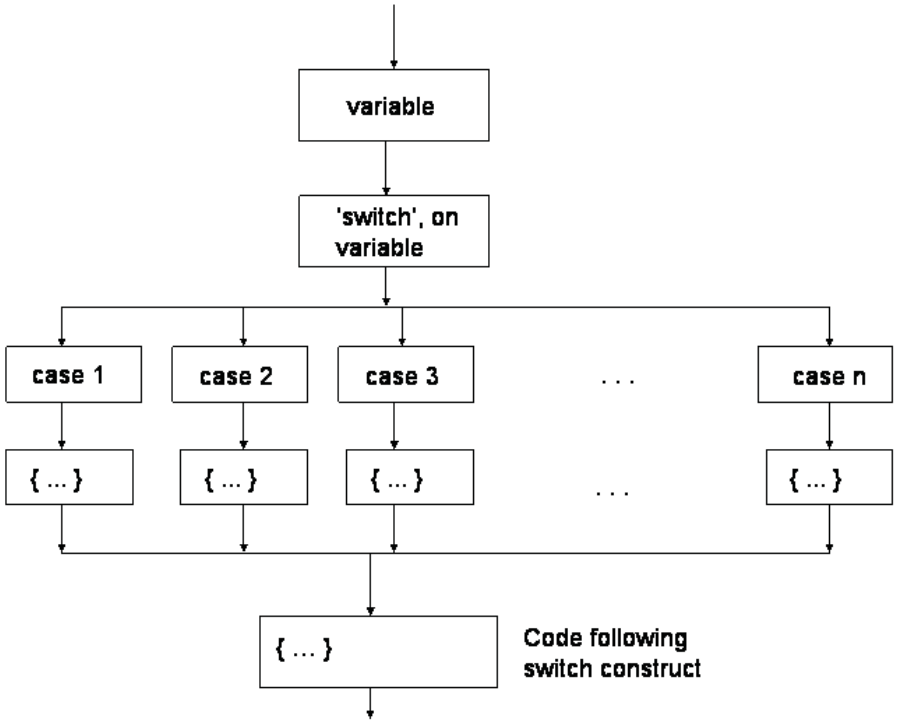

Figure 2.3 The logic of the **switch ... case** construct

The logic of the **switch ... case** construct is such that the integer condition is tested against each case in order from left to right. When this logic is translated in Java source code, the generalised syntax is as follows.

```
int someIntegerValue;
switch ( someIntegerValue ) { // start of switch block
        case 1: // do something; break;
        case 2: // do something else; break;
        case 3: // do something else ; break;
        case 3: // do something else; break;
        case 5: // do something else; break;
} // end of switch block
// first statement after the switch block
```

The **break** statement after each **case** statement is necessary to exit the enclosing **switch** block when the **switch** condition has been satisfied. When a **break** statement is executed, control passes out of the enclosing **switch** block to the first statement after the end of the **switch** block. On the face of it, it would seem logical to omit the final **break** statement. However, it is advisable to include it in case additional **case** statements are added to an existing **switch** block.

In the days before the ubiquitous use of icon-driven applications, old-fashioned text-based user interfaces for green screen types of applications were often menu-driven. **Case** statements were typically used to construct this kind of interface. While it is generally true that menu-driven applications have largely disappeared, **case** statements are useful for testing the conditional flow through application logic.

The following example illustrates testing a **switch ... case** construct with a value entered via the keyboard. (It isn't necessary to show the code used to capture a number via the keyboard for the purposes of the example.)

```
// a number entered via the keyboard is stored in a variable with the identifier month
int days;
switch( month )
{
        case 9:
        case 4:
        case 6:
        case 11: days = 30; break;
        case 2:  days = 28; break;
        default: days = 31; break;
}
System.out.println( "The number of days is: + days );
```

When this code is run in a **main** method, the output is as follows:

```
Enter the number of the month: 1
The number of days is: 31
```

```
Enter the number of the month: 2
The number of days is: 28

Enter the number of the month: 3
The number of days is: 31

Enter the number of the month: 4
The number of days is: 30

Enter the number of the month: 5
The number of days is: 31

Enter the number of the month: 6
The number of days is: 30

Enter the number of the month: 7
The number of days is: 31

Enter the number of the month: 8
The number of days is: 31

Enter the number of the month: 9
The number of days is: 30

Enter the number of the month: 10
The number of days is: 31

Enter the number of the month: 11
The number of days is: 30

Enter the number of the month: 12
The number of days is: 31
```

The output shows that the default statement is used to detect all values that aren't detected by any of the **case** statements. The output also shows that when the value of **month** is **9**, **4** or **6**, the first three **case** statements are said to 'fall through' so that the value of **days** is **30** when the value of **month** is **9**, **4**, **6** or **11**.

2.5 Controlling the Repetition of Blocks of Code

Controlling the repetition of a block of code can be achieved in one of two ways: by using a counter to repeat a block of code a known number of times; by using the evaluation of a boolean expression to decide when to stop repeating the block. The general requirement to repeat a block of code is illustrated in Figure 2.4 on the next page.

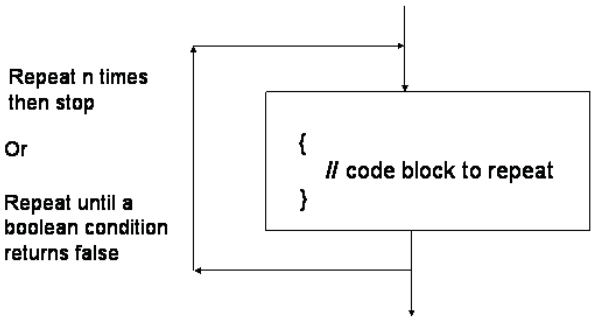

Figure 2.4 The requirement to repeat a block of code

2.5.1 Counter-Controlled Repetition

The logic of counter-controlled repetition is visualised in Figure 2.5.

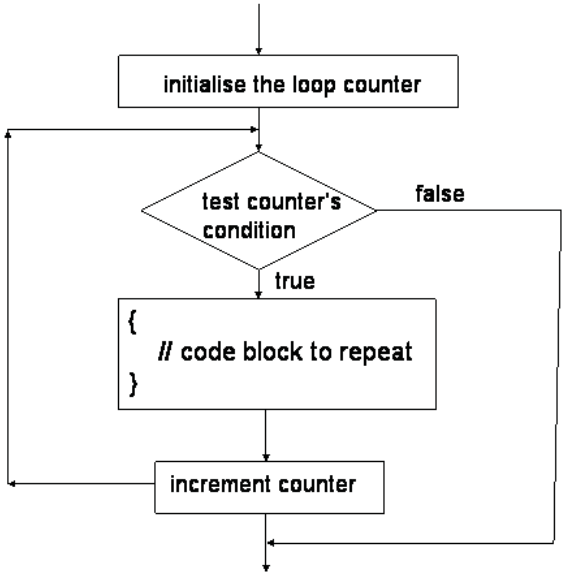

Figure 2.5 Counter-controlled repetition of a block of code

Working with the counter is implemented by what is known as a **for loop**. The general syntax of a **for loop** is as follows:

```
for ( declare and initialise the counter;
      final value condition of the counter;
      update counter )
{
        // block to repeat
}
```

The general syntax implies that the value of the final condition of the counter is a *known* value. For example, consider the following code snippet:

```
for ( int i = 0; i <= someMaxValue; i ++ )
{
        // code block to repeat
}
```

If the value of **someMaxValue** is **10**, the code block will execute 11 times. If the code snipped is modified to read as follows with the same value of **someMaxValue**

```
for ( int i = 0; i < someMaxValue; i ++ )
{
        // code block to repeat
}
```

the code block will execute 10 times. The purpose of including the two code snippets above is to show that care must be taken when initialising and setting the final value of the counter's condition when controlling the number of times that a **for loop** is executed.

The example, shown on the next page, shows a **for loop** in action in one of the methods of the **Member** class in the themed application,.

```
/**
* This method outputs the details of the cards held by the member by scanning across the
* member's array of cards with the identifier cards.
*/
public void getDetialsOfCards( ) {

        // Declare a local variable.
        MembershipCard card = null;

        // Note the use of the instanceof operator.
        for ( int i = 0; i < noOfCards; i++ ) // the value of noOfCards is known
        {
                if ( cards[ i ] instanceof DvdMembershipCard )
                {
                        card = cards[ i ];
                        System.out.println( "This is a DVD card with " +
                                card.getNoOnLoan( ) + " DVDs currently on loan." );
                } else if ( cards[ i ] instanceof GameMembershipCard )
                        {
                                card = cards[ i ];
                                System.out.println( "This is a games card with " +
                                        card.getNoOnLoan( ) + " CDs currently on
                                        loan" );
                        } else System.out.println( "Neither type of card." );
        } // End of for loop.

} // End of getDetailsOfCards.
```

The purpose of the **for loop** in the method is to scan the array of a member's (virtual) cards in order to output some information stored on each card. In the example, the number of cards held by a member is known: therefore, a **for loop** is the clear choice of construct to use to repeat the required block of code.

2.5.2 Boolean-Controlled Repetition

Java provides two boolean-controlled constructs to control the repetition of a block of code: the **while loop** and the **do ... while loop**.

The while loop

The logic of the **while** loop is visualised in Figure 2.6 below.

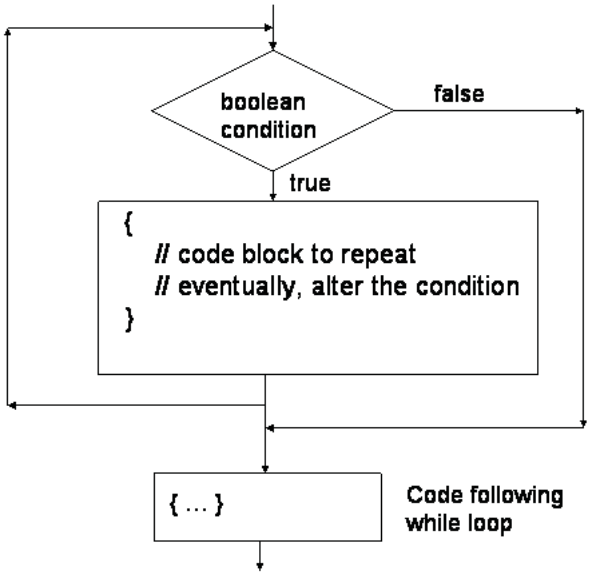

Figure 2.6 The logic of a **while** loop

Figure 2.6 implies that if the boolean condition returns an initial value of 'false', the loop will not execute at all. If, on the other hand, the boolean condition returns an initial value of 'true', the loop will repeat its execution until such time as the condition eventually returns 'false'. The general syntax of a **while** loop is as follows.

```
// initialise the variable used in the boolean condition
while( condition is true )
{
        {
                // statements to repeat
        }
        // update the condition: finally, exit the loop
}
```

The first example from Section 2.5.1 can be modified to use a **while** loop as shown on the next page.

```
int someMaxValue = 10;
int counter = 0;
while( counter < someMaxValue )
{
        {
                System.out.println( "Hello world!" );
        }
counter ++ ;
}
```

The **String** "Hello World!" is output ten times.

The do …while loop

The logic of the **do … while** loop is visualised in Figure 2.7 below.

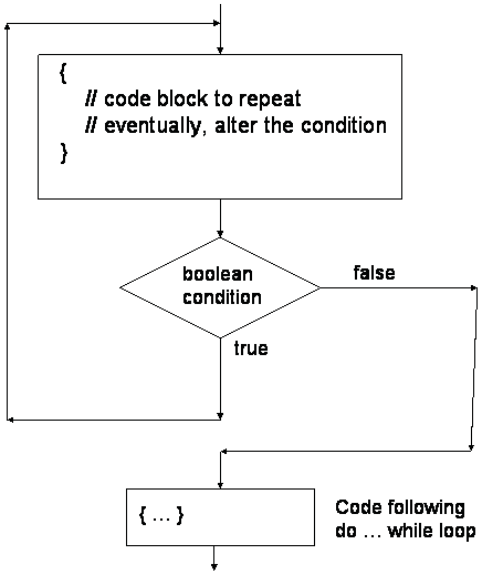

Figure 2.7 The logic of a **do … while** loop

Figure 2.7 implies that if the boolean condition returns a value of 'false', the loop will execute once. If, on the other hand, the boolean condition returns a value of 'true', the loop will repeat its execution until such time as the condition eventually returns 'false'. The general syntax of a **do … while** loop is shown on the next page.

```
// initialise the variable used in the boolean condition
do
{
        {
                // statements to repeat
        }
        // update the condition: finally, exit the loop
} while( condition is true );
```

The same example from Section 2.5.1 can be modified to use a **do ... while** loop as shown next.

```
int someMaxValue = 10;
int counter = 0;
do
{
        {
                System.out.println( "Hello world!" );
        }
counter ++ ;
} while( counter < someMaxValue );
```

In this case, the **String** "Hello World!" is output ten times.

The next example is taken from one of the classes of the graphical user interface (GUI) used to run the themed application; the example shows a **while** loop in action. The purpose of the loop is to scan the array of members of the Media Store when a member logs in via the GUI. When the member is found in the array, the loop is exited to avoid searching the remainder of the array.

```
// This method finds out if a member has registered with the Media Store and has been added
// to the array of members.  The method responds to the 'Login' button when a member
// attempts to login via the GUI.  The main purpose of the method is to find the member in the
// array of members.
// Read in the array of members. mediaStore is the reference to the MediaStore object created
// elsewhere in the application.
mediaStore.readMembers( );
// Capture the member's user name and password from the GUI.  Store these values in local
// variables username and password.
// Concatenate the user's user name and password.
String searchString = userName + password;
// Search the array of members for the combined user name and password.  First, get the array
// of members.
Member[ ] existingMembers = mediaStore.getMembers( );
// Scan the array of existing members and compare the search string with each member's
// combined user name and password.
boolean flag = true;
while( flag == true )
{
        for ( int i = 0; i < mediaStore.getNoOfMembers( ); i ++ )
        {
                existingMember = existingMembers[ i ];
                String existingUserName = existingMember.getUserName( );
                String existingPassword = existingMember.getPassword( );
                String combinedNameAndPassword =
                        existingUserName + existingPassword;
                if ( searchString.equals( combinedNameAndPassword ) )
                {
                        // Found existing member in the array of members.
                // Output a message to the GUI.
                        flag = false;
                break; // out of the for loop
                } // end if
        }  // end of for loop
break; // out of the while loop
} // end of while loop
// if there is no match, output a suitable message
```

```
if ( flag == true )
{
        // output "No such member; please try again." );
}
```

2.6 Deciding Which Construct to Use

Deciding which construct to use in any particular situation is a matter of judgement that will become easier to make when the learner gains practical experience. For example, the method implementation explained at the end of the previous sub-section is a direct consequence of the logic required to find a member in the array of members and then break out of the loop to avoid unnecessary iterations of the main loop. As can be seen from the code, the full implementation of the method is a combination of nested **if**, **while** and **for** constructs.

Before completing our examination of the contents of Table 2.1, it is worthwhile mentioning another 'loop within a loop' nested construct.

2.6.1 Nested for loops

One of my university colleagues sets a fiendish exercise to students enrolled on his Java course. The main task of the exercise is to print a calendar. While on the face of it this may seem a straightforward exercise, it actually involves a number of loops within loops, generally referred as *nested for loops*. A highly generalised skeleton solution to this exercise might read as follows.

```
for ( int months = 1; months < 13; months ++ ) // the month loop
{
        for ( int weeks = 1; weeks < 5; weeks ++ ) // the week loop
        {
                for ( int days = 1; days < 32; days ++ ) // the day loop
                {
                        // output the day and date in a calendar format
                }
        }
} // end of outer for loop
```

The outline example above suggests that nested **for** loops can be used to construct tables or two-dimensional arrays of data. The inner loop is used to output each entry in a row and the outer loop can be used to move to the next column. The next code snippet outputs the value of each cell of a table or array.

```
for ( int i = 1; i <= numberOfColumns; i++ )
{
        for ( int j = 1; j <= cellValue; j++ )
        {
                // output the known value of the cell at co-ordinates j , i, i.e. the jth row of the
                // ith column
```

```
            } // end of inner for loop
        } // end of outer for loop
```

The next section almost completes our examination of Table 2.1 by considering *branching statements*.

2.7 Branching Statements

Branching statements are used to terminate a loop or a decision construct or to skip an iteration of a loop.

2.7.1 The Unlabelled break Statement

A number of the examples discussed in this chapter include **break** statements to expedite the immediate exit from a flow control construct. The **break** statement is used to terminate **switch**, **for**, **while** and **do ... while** constructs. For example, when the following method is invoked

```
public void someMethod( ) {

        for ( int i = 0; i < 11; i++ )
        {
                System.out.print( i + ", " );
                if ( i == 5 )
                break; // out of the enclosing for loop using an unlabelled break
        }
        System.out.println( "for loop terminated." );

} // End of someMethod
```

the output is:

0, 1, 2, 3, 4, 5, for loop terminated.

The output shows that the **break** statement causes the enclosing **for** loop to terminate and control passes to the first statement after the **for** loop.

2.7.2 The Unlabelled continue Statement

We have not encountered the **continue** statement thus far in this guide. The **continue** statement is used to skip an iteration of **for**, **while** and **do ... while** loops.

For example, when the method shown on the next page is invoked, it produces the output displayed after the body of the method.

```
public void someMethod( ) {

        for ( int i = 0; i < 11; i++ )
        {
                if ( i == 5 )
                continue; // skip this iteration of the loop using an unlabelled continue
                        // statement
                System.out.print( i + ", " );
        }
        System.out.println( "The iteration when i is 5 is skipped." );

} // End of someMethod.
```

The output is:

0, 1, 2, 3, 4, 6, 7, 8, 9, 10, The iteration when i is 5 is skipped.

The output shows that the **continue** statement causes the enclosing **for** loop to skip the iteration when **i** is **5** and control passes to the next iteration of the enclosing loop.

The examples in sub-sections 2.7.1 and 2.7.2 illustrate the use of the *unlabelled* form of the **break** and **continue** statements. When loops are nested, the *labelled* form of the **break** and **continue** statements identify which of the outer loops are involved in a branch. The reader is referred to the relevant section of Sun's on-line Java tutorial for examples that explain the labelled form of the branching statements.

http://java.sun.com/docs/books/tutorial/java/nutsandbolts/branch.html

2.8 Handling Exception Objects

The final row of Table 2.1 mentions a special kind of decision, namely the one that uses the **try ... catch** construct to detect error conditions in Java programmes. Chapter Four explains how errors are detected by objects of the **Exception** class. For the purposes of this chapter, and to complete the consideration of Table 2.1, one way of looking at the **try ... catch** block is to consider the generalised code snippet that is shown on the next page.

```
// one or more of the methods invoked in the next code block are known to produce errors, i.e.
// they are said to 'throw' errors that must be detected or 'caught' in a separate code block
try
{
        // method invocations that throw errors are coded here
}
catch ( Exception e )
{
        // do something about the error
}
```

The structure of the **try ... catch** construct implies that a decision is made depending upon whether an error is detected or not. If an error is thrown by one of the method invocations in the **try** block, the remaining statements of the **try** block are skipped and the statements in the **catch** block are executed. If, on the other hand, method invocations in the **try** block do not throw any errors, the statements of the **catch** block are skipped and control passes to the first statement after the end of the **catch** block. Chapter Four goes into details about how **try ... catch** blocks are used to write robust Java code.

In the next chapter, we will find out how we can extend classes by means of a very important concept known as *inheritance*.