### Statements

Statements are everywhere in code. Nearly every line that you write is going to be a statement. Statements are a way to take your thoughts and organize them into logical code that the compiler can follow and process. A good understanding of statements in C# is essential to being able to write good code.

##### What are Statements?

A statement in C# (or almost every other programming language going) can be thought of as equiv- alent to a complete sentence in the English language. It might seem odd to compare a programming language with a real, living language, but this happens to be the best and easiest analogy.

For example, if someone says:

I like C#.

You know exactly what they mean. However, if they said:

I like.

or

I C#.

You would realize that there’s something wrong with these sentences. They’re not complete, and they are ambiguous.

The same is true for a statement in C#. A statement in C# is a complete instruction that the com- piler understands and can process. The statement has to be valid and make sense to the compiler, and it has to follow syntax rules just as sentences in English must.

Here’s a simple statement in C#:

var1 = 3 + 4;

This is a single statement in C#. It’s logical and makes perfect sense to the compiler, which will take the two numbers, add them together, and store the result in a variable called var1.

No ambiguities. No problems.

C# statements don’t end with a period like sentences in English but instead with a semicolon (;). This is used to indicate to the compiler that the statement has ended. Just as sentences in English don’t make any sense if the period is missing and they run into one another, C# statements that don’t have the termi- nator at the end are also not valid.

Just as sentences build on one another to form paragraphs, statements build to form code blocks. In code blocks, statements are processed one by one:

{

statement1; statement2; statement3;

}

In this code block, three statements are processed one after the other, starting with statement1 and ending after statement3.

There would be nothing technically wrong with putting all the statements on a single line — the com- piler can still find the end of each statement because of the semicolon:

{

statement1;statement2;statement3;

}

The problem with this kind of layout is that it makes reading the code and future debugging an awful experience.

The following layout is looser and makes it easier to read the code:

{

statement1; statement2; statement3;

}

So far, all this seems simple enough, but as you can imagine, there are numerous specific rules governing statements, and we will be looking at these rules in the remainder of this chapter.

##### C# Statements

A number of different types of statements are possible in C#:

statement:

labeled-statement declaration-statement embedded-statement

embedded-statement: block

empty-statement expression-statement selection-statement iteration-statement jump-statement

try-statement checked-statement unchecked-statement lock-statement using-statement yield-statement

An embedded-statement is used within other statements, and these must be placed within code blocks.

This is a valid embedded-statement:

public class Test

{

public static void Main()

{

bool i = false;

if ( i)

{

int j = 7;

}

}

}

While this is invalid:

public class Test

{

public static void Main()

{

bool i = false;

if ( i) int j = 7;

}

}

End Point and Reachability

There are two other concepts that a programmer needs to be comfortable with:

* End point
* Reachability

End Point

Every valid statement has an end point. The end point of a statement is the end of the statement itself. Embedded statements within statements are called *composite statements*.

Reachability

If a statement can be reached during code execution, this statement is said to be *reachable*. If that state- ment cannot be reached, it is said to be *unreachable*.

The following code contains reachable and unreachable statements:

public class Test

{

public static void Main()

{

int x = 6;

const int y = 7; if ( x == 6)

System.Console.WriteLine(“Reachable”); if ( y == 6) System.Console.WriteLine(“Unreachable”);

}

}

What makes the unreachable statement unreachable? It’s that the value of y is defined as a constant and as such cannot change. This is detected by the compiler, and a warning is issued:

C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727>csc test.cs Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42 for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727

Copyright (C) Microsoft Corporation 2001-2005. All rights reserved. test.cs(10,9): warning CS0162: Unreachable code detected C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727>

In this example, both statements are reachable:

public class Test

{

public static void Main()

{

int x = 6;

const int y = 6; if ( x == 6)

System.Console.WriteLine(“Reachable”); if ( y == 6) System.Console.WriteLine(“Reachable”);

}

}

Take a look at this code, a slight variation of the preceding code:

public class Test

{

public static void Main()

{

int x = 7;

const int y = 7;

if ( x == 6)

System.Console.WriteLine(“Potentially reachable”);

if ( y == 6) System.Console.WriteLine(“Unreachable”);

}

}

Even though the value of x makes the if statement that refers to it currently unreachable, it is poten- tially reachable because the value of x could later be changed.

If the unreachable statement is removed (or modified) to make it reachable, the potentially reachable statement now generates a warning.

public class Test

{

public static void Main()

{

int x = 7;

const int y = 7; if ( x == 6)

System.Console.WriteLine(“Potentially reachable”); if ( y == 7) System.Console.WriteLine(“Unreachable”);

}

}

The following is always considered reachable:

* + The block of a function member
  + The block of an anonymous-method-expression

Reachability is determined by the compiler by evaluating each statement in a block. By carrying out this operation successively, the reachability of any statement can be determined.

There are two scenarios where a compile-time error is generated when the end point of a statement is reachable:

* If the end point of a function that computes a value is reachable. In this case, the return state- ment is usually missing.
* If the end point of the statement list of a switch section is reachable. This is usually the case when a break statement is missing.

##### Code Blocks

A code block (also called a block) is a way to allow multiple statements to be written in situations where only a single statement is allowed.

block:

{ statement-listopt }

A code block consists of an optional statement-list. This is enclosed in braces ({ and }). If the state- ment list is omitted, the code block is said to be empty.

A block can also contain declaration statements, and the scope of a local variable or constant declared in a code block is the block itself and no more.

A block of code is executed as follows:

* If the code block is empty, control is passed straight to the end point of the code block.
* If the block contains statements, control is transferred to the statement list, and the statements are executed. If control reaches the end point of the statement list, control is transferred to the end point of the code block.

The statement list of a code block is always reachable if the block is reachable.

Statement Lists

A statement list consists of one or more statements written and presented in a sequence. Statement lists can be found in code blocks or in switch blocks.

statement-list: statement

statement-list statement

Statement lists are executed when the control is transferred to the first statement in the list. If control reaches the end of the statement in the list, control is transferred to the end point of the statement list.

For a statement in a statement list to be reachable, the following have to be true:

* The statement is the first in the statement list, and the statement list is reachable (the first state- ment in any reachable statement list is reachable).
* The end point of the statement coming before the current statement is reachable.
* The statement is labeled, and the label is referenced by a goto statement that is itself reachable.

For the end point of a statement list to be reachable, the end point of the last statement in the list also has to be reachable.

##### Empty Statements

An empty statement does nothing. It is used when there are no operations to perform but a statement is required (such as in a while statement).

empty-statement:

;

When executed, an empty statement merely transfers control to the end point of the statement. The end point of an empty statement is always reachable.

##### Labeled Statements

A labeled statement has been prefixed by a label. This label is used to declare a unique name for the statement. These labeled statements are referenced from goto statements:

labeled-statement: identifier : statement

The scope of a label is limited to the block where the label is declared (this includes any nested blocks that the main block contains).

class Test

{

static void Main() { goto X;

X: Console.Write(“Hello, World!”);

}

}

No two labels that share the same scope can have the same name without causing a compiler error, as will happen when compiling the following example:

class Test

{

static void Main() { goto X;

X: Console.Write(“Hello, “);

X: Console.Write(“World!”);

}

}

*Note that label names don’t interfere with other identifiers in code. This means that you could have a label, a variable, and a parameter all with the same name in the same block of code.*

A labeled statement is reachable if the label is referenced by a goto statement that is itself reachable. The only exception is where the goto statement is inside a try that includes a finally block whose end point is unreachable, and the labeled statement is outside the try.

##### Declaration Statements

Declaration statements are used to declare either a local variable or a constant. Declaration statements are allowed inside code blocks, but they are not allowed inside any embedded statements:

declaration-statement:

local-variable-declaration ; local-constant-declaration ;

Local Variable Declarations

Local variable declarations are used to declare one or more local variables:

local-variable-declaration:

type local-variable-declarators

local-variable-declarators: local-variable-declarator

local-variable-declarators , local-variable-declarator

local-variable-declarator: identifier

identifier = local-variable-initializer

local-variable-initializer: expression

array-initializer

The type of declaration specifies the type of the variables brought into existence by the declaration.

The type is followed by a list of declarators, each of which specifies a new variable. A declarator consists of an identifier that names the variable and is optionally followed by an = token and an initializer that gives the initial value of the variable.

The value of a local variable is retrieved by an expression using a simple name, while the value of a local variable is modified using an assignment. A local variable has to be definitely assigned at each location where its value is retrieved.

The scope of a local variable declared in a local variable declaration is the block in which the declaration is found. Code cannot refer to a local variable in a textual position that comes before the local variable declarator of the local variable.

Also, you cannot declare another variable or constant within the scope of another variable or constant with the same name.

Here are two ways to declare and assign a variable:

class Test

}

And:

{

static void Main() { int x = 7;

}

class Test

{

static void Main() { int x;

x = 7;

}

}

The code in these two blocks is functionally equivalent.

Local Constant Declarations

Local constant declarations are used to declare one or more local constants:

local-constant-declaration:

const type constant-declarators

constant-declarators: constant-declarator

constant-declarators , constant-declarator

constant-declarator:

identifier = constant-expression

The type of declaration specifies the type of the constants brought into existence by the declaration.

The type is followed by a list of declarators, each of which specifies a new constant. A declarator consists of an identifier that names the variable and is optionally followed by an = token and an initializer that gives the initial value of the constant.

The value of a local constant is retrieved by an expression using a simple name.

The scope of a local constant declared in a local constant declaration is the block in which the declaration is found.

Also, you cannot declare a constant within the scope of another constant with the same name.

##### Expression Statements

Expression statements are used to evaluate an expression. Values that result from expressions are discarded unless they are preserved (by assigning them to variables):

expression-statement: statement-expression ;

statement-expression: invocation-expression object-creation-expression assignment

post-increment-expression post-decrement-expression pre-increment-expression pre-decrement-expression

It is important to note that some expressions are not permitted. For example, the following are used only to compute values and are not in themselves valid expressions:

x + y + z; x ==7;

Execution of an expression statement evaluates the expression and, after that is completed, transfers con- trol to the end point of the expression statement.

The end point of an expression statement is always reachable if that expression statement itself is reachable.

Selection Statements

Selection statements are used to select appropriate statements to run from a list of possible statements. The decision as to what statements to run is based on the outcome of a selection expression:

selection-statement: if-statement switch-statement

The if Statement

The if statement is used to select statements for execution based on the value of a Boolean expression:

if-statement:

if ( boolean-expression ) embedded-statement

if ( boolean-expression ) embedded-statement else embedded-statement

The if statement also allows for there to be an else clause. The else clause is associated with the lexi- cally nearest preceding if allowed by the syntax.

The following code examples show equivalent if statements:

if (x)

{

if (y)

{ A();

}

else

}

{ B();

}

And:

if (x) if (y) A(); else B();

Which style you use is a personal choice.

The steps carried out to execute an if statement are as follows:

* + The Boolean expression that the if statement depends on is first evaluated.
  + If the Boolean expression evaluates to true, control is transferred to the first embedded statement. If control reaches the end point of that statement, control is transferred to the end point of the entire if statement.
  + If the Boolean expression evaluates to false and an else clause is present, control is transferred to the second embedded statement. If control reaches the end point of that statement, control is transferred to the end point of the if statement.
  + If the Boolean expression evaluates to false and if an else clause is not specified, control is transferred to the end point of the if statement.

The first embedded statement of any if statement will be reachable if the if statement is reachable and the Boolean expression does not have the constant value false.

The second embedded statement of an if statement, if present, will be reachable if the if statement is reachable and the Boolean expression does not have the constant value true.

The end point of any if statement will be reachable if the end point of at least one of the embedded statements is reachable. The end point of an if statement with no else part will be reachable if the if statement is reachable and the Boolean expression does not have the constant value true.

The switch Statement

The switch statement selects a statement list for execution that has a switch label that corresponds to the value of the switch expression.

A switch statement is a substitute for multiple if statements — both work in the same way. It is ulti- mately a matter of style as to which of them to use. The syntax is as follows:

switch-statement:

switch ( expression ) switch-block

switch-block:

{ switch-sectionsopt }

switch-sections:

switch-section

switch-sections switch-section

switch-section:

switch-labels statement-list

switch-labels:

switch-label

switch-labels switch-label

switch-label:

case constant-expression :

default :

The switch statement consists of four parts:

* At the core of the switch statement is the keyword switch.
* Following this keyword is a parenthesized expression called the switch expression.
* This is followed by a switch block. A switch block is made up of zero or more switch sections enclosed in braces.
* Switch sections are made up of one or more switch labels followed by a statement list.

Here is an example of a switch statement. We have labeled which statement is executed with the words “executed.”

public class test

{

public static void Main()

{

test a = new test(); a.xyz(1);

}

void xyz(int i)

{

switch (i)

{

case 0:

System.Console.WriteLine(“not executed”); break;

case 1: System.Console.WriteLine(“executed”); break;

default:

System.Console.WriteLine(“not executed”); break;

}

}

}

The governing type of a switch statement is worked out by the switch expression. If the type of the

switch expression is any of the following types, that will become the governing type:

* + byte
  + sbyte
  + char
  + int
  + uint
  + long
  + ulong
  + short
  + ushort
  + string
  + an enum type

Otherwise, one (and one only) user-defined implicit conversion operator will be present that will convert from the type of the switch expression or a base type of this type to one of the following governing types:

* + byte
  + sbyte
  + char
  + int
  + uint
  + long
  + ulong
  + short
  + ushort
  + string

If no implicit conversion operator exists or if more than one such implicit conversion operator is present, a compiler error will be generated.

Switch statements are executed as follows:

* + The switch expression is evaluated and converted to the appropriate governing type.
  + If one of the constants specified in a case label in the same switch statement matches the value of the switch expression, control is transferred to the statement list that follows the matched case label.
* If none of the constants specified in case labels in the same switch statement is equal to the value of the switch expression and if a default label is present, control is then transferred to the statement list that follows the default label.
* If none of the constants specified in case labels in the same switch statement is equal to the value of the switch expression and no default label is present, control is transferred to the end point of the switch statement.

In the following code example, the statement list after the default label is run:

public class test

{

public static void Main()

{

test a = new test();

a.xyz(7);

}

void xyz(int i)

{

switch (i)

{

case 0:

System.Console.WriteLine(“not executed”); break;

case 1:

System.Console.WriteLine(“not executed”); break;

default: System.Console.WriteLine(“executed”); break;

}

}

}

Note that statement lists in a switch section usually end with one of the following statements:

* break
* goto case
* goto default

However, any statement that makes the end point of the list unreachable is valid (for example, a while

statement controlled by a Boolean expression that evaluates to true).

Multiple labels are allowed in switch sections:

public class test

{

public static void Main()

{

test a = new test();

a.xyz(2);

}

void xyz(int i)

{

switch (i)

{

case 0:

System.Console.WriteLine(“not executed”); break;

case 1:

System.Console.WriteLine(“not executed”); break;

case 2:

default: System.Console.WriteLine(“executed”); break;

}

}

}

The statement lists contained in a switch block are allowed to contain declaration statements. The scope of these local variables or constants will be the switch block in which they are declared.

The statement list of a given switch section is reachable if the switch statement is reachable and if one or more of the following are true:

* + The switch expression is a constant value that matches a case label in the switch section.
  + The switch expression is a nonconstant value.
  + The switch expression is a constant value that doesn’t match any case label, but the switch

section contains the default label.

* + A switch label of the switch section is referenced by a goto case or goto default statement that is itself reachable.

The end point of a switch statement is reachable if one or more of the following are true:

* + The switch statement contains a reachable break statement that exits the switch statement.
  + The switch statement is reachable, the switch expression is a nonconstant value, and there is no default label present.
  + The switch statement is reachable, the switch expression is a constant value that doesn’t match any case label, and no default label is present.

Iteration Statements

Iteration statements are used to execute an embedded statement repeatedly:

iteration-statement: while-statement do-statement for-statement

foreach-statement

The while Statement

The while statement is used to conditionally execute an embedded statement zero or more times:

while-statement:

while ( boolean-expression ) embedded-statement

All while statements are evaluated as follows:

* First, the Boolean expression is evaluated.
* If the Boolean expression evaluates to true, control is transferred to the embedded statement. If control reaches the end point of the embedded statement, control is transferred to the beginning of the while statement.
* If the Boolean expression evaluates to false, control is transferred to the end point of the while

statement.

The embedded statement of a while statement is reachable when the while statement is reachable and the Boolean expression is not set to have the constant value false.

The end point of a while statement will be reachable if at least one of the following is true:

* The while statement contains a reachable break statement that exits the while statement.
* The while statement is reachable, and the Boolean expression is not set to have the constant value true.

The do Statement

The do statement is used to conditionally execute an embedded statement one (not zero) or more times:

do-statement:

do embedded-statement while ( boolean-expression ) ;

All do statements are executed as follows:

* Control is initially passed to the embedded statement.
* If control reaches the end point of the embedded statement, the Boolean expression is evaluated. If that Boolean expression evaluates to true, control is transferred to the beginning of the do statement, and another iteration cycle is processed. If the Boolean expression evaluates to false, control is transferred to the end point of the do statement.

The embedded statement of a do statement is always reachable if the do statement itself is reachable. The end point of a do statement will be reachable if at least one of the following is true:

* + The do statement contains a reachable break statement that exits the do statement.
  + The end point of the embedded statement is reachable, and the Boolean expression does not have the constant value true.

The for Statement

The for statement is used to evaluate a sequence of initialization expressions. While the condition eval- uates to true, the for statement repeatedly executes the statement and each time evaluates the iteration expressions.

for-statement:

for ( for-initializeropt ; for-conditionopt ; for-iteratoropt ) embedded-statement for-initializer:

local-variable-declaration statement-expression-list

for-condition:

boolean-expression

for-iterator:

statement-expression-list

statement-expression-list:

statement-expression

statement-expression-list , statement-expression

A for statement is executed as follows:

* + If a for initializer is present, the variable initializers or statement expressions are executed in the order they are written. This step is only carried out once, no matter how many times the statement is executed.
  + If a for condition is present, it is next evaluated.
  + If the for condition is not present or if the evaluation evaluates to true, control is transferred to the embedded statement. If control reaches the end point of the embedded statement, the expressions of the for iterator, if any, are evaluated in sequence, and then another iteration is performed, starting with evaluation of the for condition from the preceding step.
  + If the for condition is present and the evaluation evaluates to false, control is then transferred to the end point of the for statement.

The embedded statement of a for statement is reachable if one of the following is true:

* + The for statement is reachable, and so no for condition is present.
  + The for statement is reachable, and a for condition is present but does not have the constant value false.

The end point of a for statement will be reachable if at least one of the following is true:

* The for statement contains a reachable break statement that exits the for statement.
* The for statement is reachable and a for condition is present but does not have the constant value true.

The foreach Statement

The foreach statement enumerates the elements of a collection, executing an embedded statement for each element of the collection:

foreach-statement:

foreach ( type identifier in expression ) embedded-statement

The type and identifier of a foreach statement declare the iteration variable of the statement. The itera- tion variable is a read-only local variable that has scope that extends over the embedded statement.

When the statement is executed, the iteration variable is used to represent the collection element for which an iteration is currently being performed.

A compiler error is generated if the embedded statement tries to modify the iteration variable in any way or if an attempt is made to pass the iteration variable as a ref or out parameter.

Jump Statements

Jump statements are used to unconditionally transfer control to another statement in the code. The loca- tion to which the jump occurs is called the target of the jump statement:

jump-statement: break-statement

continue-statement goto-statement return-statement throw-statement

Jump statements can transfer control from a block of code but not into a block of code.

The break Statement

The break statement is used to exit an enclosing do, for, foreach, switch, or while statements (in fact, break statements have to be enclosed by one of these statements or a compiler error will occur):

break-statement: break ;

In the event that a break statement is enclosed in a nested set of statements, the break statement applies only to the innermost statement.

All break statements are processed as follows:

* If the break statement is used to exit one or more try blocks that have associated finally blocks, control is first transferred to the finally block of the innermost try statement. If con- trol reaches the end point of a finally block, control is then transferred to the finally block

of the next enclosing try statement. This process is repeated until the finally blocks of all try

statements have been executed.

* + Control is then transferred to the target of the break statement. The end point of a break statement is never reachable.

The continue Statement

The continue statement is used to begin a new iteration cycle of the enclosing do, for, foreach, and

while statements:

continue-statement:

continue ;

When there are multiple enclosing do, for, foreach, and while statements, the continue statement only applies to the innermost enclosing statement.

The end point of the continue statement is never reachable. A continue statement is processed as follows:

* + If the continue statement is used to exit one or more try blocks with associated finally blocks, control is first passed to the finally block of the innermost try statement. If control reaches the end point of a finally block, control is then passed to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all try statements have been executed.
  + Control is transferred to the target of the continue statement.

The goto Statement

The goto statement is used to transfer control to a statement that has been marked using a label:

goto-statement:

goto identifier ;

goto case constant-expression ; goto default ;

The target of any goto identifier statement is a statement marked by a label. If a label with the given name does not exist in the current function member, or if the goto statement is not within the scope of the label, a compiler error is generated.

A goto statement is executed as follows:

* + If the goto statement is used to exit one or more try blocks with associated finally blocks, control is first passed to the finally block of the innermost try statement. If control reaches the end point of a finally block, control is then transferred to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all try statements have been executed.
  + Control is transferred to the target of the goto statement.

The end point of a goto statement is always unreachable.

The return Statement

The return statement is used to return control to the caller of the function member:

return-statement:

return expressionopt ;

A return statement is executed as follows:

* If the return statement is used to specify an expression, the expression is evaluated and the resulting value is converted to the return type of the containing function member using an implicit conversion. The result of the conversion is then set as the value returned to the caller.
* If the return statement is enclosed by one (or more) try blocks that have finally blocks, con- trol is first passed to the finally block of the innermost try statement. If control reaches the end point of a finally block, control is then transferred to the finally block of the next enclosing try statement. This process is repeated until all the finally blocks of all enclosing try statements have been executed.
* Control is returned to the caller of the containing function member. The end point of a return statement is always unreachable.

The throw Statement

The throw statement is used to throw exceptions:

throw-statement:

throw expressionopt ;

A throw statement with an expression is used to throw the value produced by evaluating the expres- sion. The expression will indicate a value of the class type System.Exception or a class type derived from System.Exception. If, on evaluation, the expression results in a null, a System.NullReferenceException will be thrown instead.

The throw statement can be used with expressions that have a type given by a type parameter only where that type parameter has System.Exception or a subclass of System.Exception as the effective base class.

A throw statement with no expression can only be used in catch blocks. Here the statement will rethrow the exception currently being handled by that catch block.

The end point of a throw statement is always unreachable.

The using Statement

The using statement is used to obtain one or more resources, execute a statement, and finally dispose of the resources:

using-statement:

using ( resource-acquisition ) embedded-statement

resource-acquisition:

local-variable-declaration expression

A resource is a class or struct that implements the System.IDisposable interface.

The using statement is only useful for objects with a lifetime that does not extend beyond the method in which the objects are constructed.

A using statement is translated into three parts:

* + Acquisition
  + Usage
  + Disposal

Usage of the resource will be implicitly enclosed in a try statement that includes a finally clause. This

finally clause is used to dispose of the resource when it is finished.

Instantiated objects must implement the System.IDisposable interface. Note that the following code snippets are equivalent in function:

using (ResourceType resource = expression) embedded-statement

And:

{

ResourceType resource = expression; try

{

embedded-statement

}

finally

{

}

}

The yield Statement

The yield statement is used inside iterator blocks to yield a value to the enumerator object. It is also used to indicate the end of the iteration.

yield-statement:

yield return expression ; yield break ;

*Note that in order to maintain compatibility,* yield *is not a keyword. Instead, it has special meaning only when it is used before a* return *or* break *keyword. In all other contexts,* yield *is used as an identifier.*

There are a number of restrictions on the location where a yield statement can appear.

* A yield statement cannot appear outside any of the following: accessor-body, method-body, or operator-body.
* A yield statement cannot appear anywhere in a try statement that contains catch clauses.
* A yield statement cannot appear in the finally clause of a try statement.
* A yield statement cannot appear inside an anonymous method. A yield return statement is executed as follows:
* The expression that appears in the statement is evaluated and implicitly converted to the yield

type. This is assigned to the Current property of the enumerator object.

* Execution of the iterator block is halted. If the yield return statement is within one or more

try blocks, the associated finally blocks are not yet executed.

* The MoveNext method of the enumerator object returns true to the caller. This indicates that the enumerator object has moved on to the next item.

A yield break statement is executed as follows:

* If the yield break statement is enclosed by one or more try blocks that have finally blocks, control is first transferred to the finally block of the innermost try statement. If control reaches the end point of a finally block, control is then passed to the finally block of the next enclosing try statement. This process is looped until the finally blocks of all enclosing try statements have been executed.
* Control is then returned to the caller of the iterator block. This is either the MoveNext method or

Dispose method of the enumerator object.

The end point of a yield break statement is always unreachable.

##### Summary

In this chapter you looked at C# statements. The chapter started off by taking a broad look at statements and how they work, before taking at look at specific statements present in C#.

In Chapter 11, you look at namespaces and how they are used in C#.