MODULE-1

CHAPTER 1

Welcome to C#

Beginning programming with the Visual Studio 2013 environment:

Visual Studio 2013 is a tool-rich programming environment containing the functionality that you need to create large or small C# projects running on Windows 7, Windows 8, and Windows 8.1. You can even construct projects that seamlessly combine modules written in different programming languages such as C++, Visual Basic, and F#.

Note: A console application is an application that runs in a command prompt window rather than providing a graphical user interface (GUI).

Files associated with the new project in Visual Studio:

Solution File:

This is the top-level solution file. Each application contains a single solution file. A solution can contain one or more projects, and Visual Studio 2013 creates the solution file to help organize these projects.

Project file:

Each project file references one or more files containing the source code and other artifacts for the project, such as graphics images. You must write all the source code in a single project in the same programming language.

Properties

This folder contains a file called ***AssemblyInfo.cs***. *AssemblyInfo.cs* is a special file that you can use to add attributes to a program, such as the name of the author, the date the program was written, and so on. You can specify additional attributes to modify the way in which the program runs.

References

This folder contains references to libraries of compiled code that your application can use. When your C# code is compiled, it is converted into a library and given a unique name. In the

Microsoft .NET Framework, these libraries are called ***assemblies***. Developers use assemblies to package useful functionality that they have written so that they can distribute it to other developers who might want to use these features in their own applications.

App.config

This is the application configuration file. It is optional, and it might not always be present. You can specify settings that your application can use at run time to modify its behaviour, such as the version of the .NET Framework to use to run the application.

Program.cs

This is a C# source file, and it is displayed in the Code and Text Editor window when the project is first created. You will write your code for the console application in this file. It also contains some code that Visual Studio 2013 provides automatically.

Using namespaces

The example you have seen so far is a very small program. However, small programs can soon grow into much bigger programs. As a program grows, two issues arise. First, it is harder to understand and maintain big programs than it is to understand and maintain smaller ones. Second, more code usually means more classes, with more methods, requiring you to keep track of more names. As the number of names increases, so does the likelihood of the project build failing because two or more names clash; for example, you might try and create two classes with the same name. The situation becomes more complicated when a program references assemblies written by other developers who have also used a variety of names.

In the past, programmers tried to solve the name-clashing problem by prefixing names with some sort of qualifier (or set of qualifiers). This is not a good solution because it’s not scalable; names become longer, and you spend less time writing software and more time typing (there is a difference), and reading and rereading incomprehensibly long names. Namespaces help solve this problem by creating a container for items such as classes. Two classes with the same name will not be confused with each other if they live in different namespaces. You can create a class named *Greeting* inside the namespace named *TestHello* by using the *namespace* keyword like this:

```
namespace TestHello
{
        class Greeting
        {
                ...
        }
}
```

You can then refer to the *Greeting* class as *TestHello.Greeting* in your programs. If another developer also creates a *Greeting* class in a different namespace, such as *NewNamespace*, and you install the assembly that contains this class on your computer, your programs will still work as expected because they are using the *TestHello.Greeting* class. If you want to refer to the other developer's *Greeting* class, you must specify it as *NewNamespace.Greeting*. It is good practice to define all your classes in namespaces, and the Visual Studio 2013 environment follows this recommendation by using the name of your project as the top-level namespace.

The .NET Framework class library also adheres to this recommendation; every class in the .NET Framework lives within a namespace. For example, the *Console* class lives within the *System* namespace. This means that its full name is actually *System.Console*. Of course, if you had to write the full name of a class every time you used it, the situation would be no better than prefixing qualifiers or even just naming the class with some globally unique name such *SystemConsole*. Fortunately, you can solve this problem with a *using* directive in your programs. If you return to the *TestHello* program in Visual Studio 2013 and look at the file *Program.cs* in the Code and Text Editor window, you will notice the following lines at the top of the file:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

These lines are *using* directives. A *using* directive brings a namespace into scope. In subsequent code in the same file, you no longer need to explicitly qualify objects with the namespace to which they belong. The five namespaces shown contain classes that are used so often that Visual Studio 2013 automatically adds these *using* statements every time you create a new project. You can add further *using* directives to the top of a source file if you need to reference other namespaces.

CHAPTER 2

Working with variables, operators, and expressions

Understanding statements

- A ***statement*** is a command that performs an action, such as calculating a value and storing the result, or displaying a message to a user.
- Statements in C# follow a well-defined set of rules describing their format and construction. These rules are collectively known as ***syntax***.
- In contrast, the specification of what statements *do* is collectively known as ***semantics***.
- One of the simplest and most important C# syntax rules states that you must terminate all statements with a semicolon.

Console.WriteLine("Hello, World!");

Using identifiers

- ***Identifiers*** are the names that you use to identify the elements in your programs, such as namespaces, classes, methods, and variables.

Syntax rules when choosing identifiers:

  - You can use only letters (uppercase and lowercase), digits, and underscore characters.
  - An identifier must start with a letter or an underscore.

- For example, *result*, *_score*, *footballTeam*, and *plan9* are all valid identifiers, whereas *result%*, *footballTeam$*, and *9plan* are not.

Important: C# is a case-sensitive language: *footballTeam* and *FootballTeam* are two different identifiers.

Identifying keywords

- The C# language reserves 77 identifiers for its own use, and you cannot reuse these identifiers for your own purposes. These identifiers are called ***keywords***, and each has a particular meaning.
- Examples of keywords are ***class***, ***namespace***, and ***using***.

The following is the list of keywords:

abstract	do	in	protected	true
as	double	int	public	try
base	else	interface	readonly	typeof
bool	enum	internal	ref	uint
break	event	is	return	ulong
byte	explicit	lock	sbyte	unchecked
case	extern	long	sealed	unsafe

- C# also uses the identifiers that follow. These identifiers are not reserved by C#, which means that you can use these names as identifiers for your own methods, variables, and classes, but you should avoid doing so if at all possible.

add	get	remove
alias	global	select
ascending	group	set
async	into	value
await	join	var
descending	let	where
dynamic	orderby	yield
from	partial	

Using variables

- A ***variable*** is a storage location that holds a value. You can think of a variable as a box in the computer's memory that holds temporary information.
- You must give each variable in a program an unambiguous name that uniquely identifies it in the context in which it is used.
- You use a variable's name to refer to the value it holds. For example, if you want to store the value of the cost of an item in a store, you might create a variable simply called *cost* and store the item's cost in this variable.

Naming variables

- You should adopt a naming convention for variables that helps you to avoid confusion concerning the variables you have defined.
  - Don't start an identifier with an underscore.
  - Don't create identifiers that differ only by case. For example, do not create one variable named *myVariable* and another named *MyVariable* for use at the same time, because it is too easy to get them confused.
  - Start the name with a lowercase letter.
  - In a multiword identifier, start the second and each subsequent word with an uppercase letter. (This is called *camelCase notation*.)
  - Don't use Hungarian notation.

For example, *score*, *footballTeam*, *_score*, and *FootballTeam* are all valid variable names, but only the first two are recommended.

Declaring variables

- Variables hold values. C# has many different types of values that it can store and process—integers, floating-point numbers, and strings of characters, to name three.
- When you declare a variable, you must specify the type of data it will hold.
- You declare the type and name of a variable in a declaration statement. For example, the statement that follows declares that the variable named *age* holds *int* (integer) values. As always, you must terminate the statement with a semicolon.

```
int age;
```

The variable type ***int*** is the name of one of the *primitive* C# types, *integer*, which is a whole number.

Note: If you are a Visual Basic programmer, you should note that C# does not allow implicit variable declarations. You must explicitly declare all variables before you use them.

After you've declared your variable, you can assign it a value. The statement that follows assigns *age* the value 42. Again, note that the semicolon is required.

```
age = 42;
```

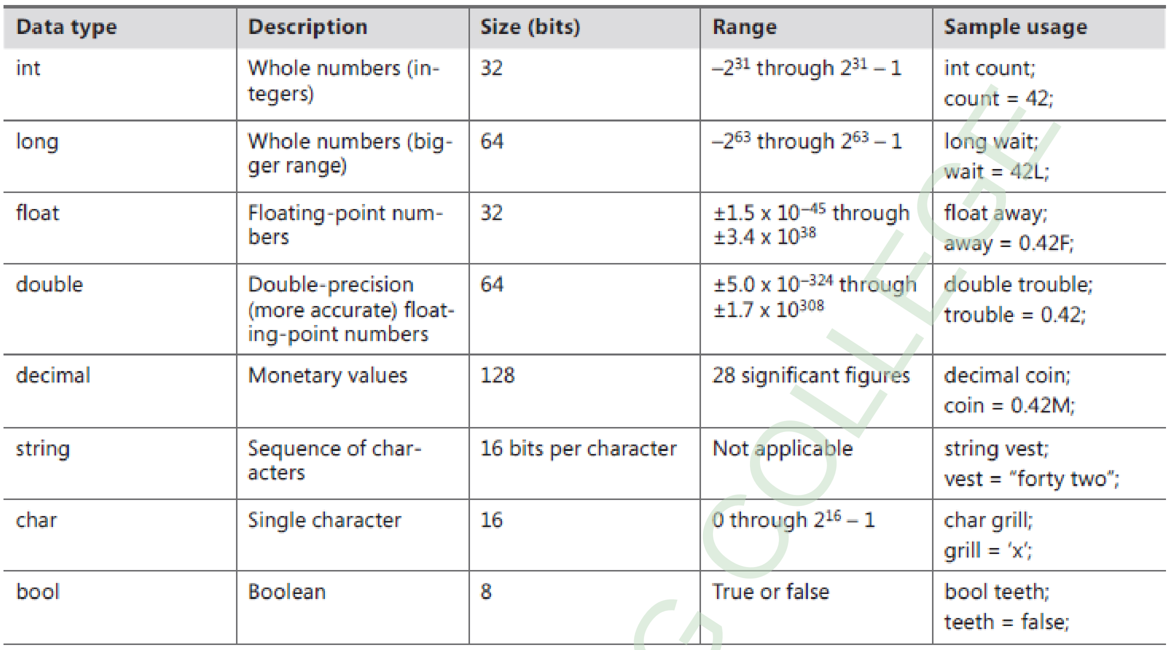
The equal sign (=) is the *assignment* operator, which assigns the value on its right to the variable on its left. After this assignment, you can use the *age* variable in your code to refer to the value it holds.

The next statement writes the value of the *age* variable (42) to the console:

```
Console.WriteLine(age);
```

Working with primitive data types

C# has a number of built-in types called *primitive data types*. The following table lists the most commonly used primitive data types in C# and the range of values that you can store in each.



Data type	Description	Size (bits)	Range	Sample usage
int	Whole numbers (integers)	32	-2^{31} through $2^{31}-1$	int count; count = 42;
long	Whole numbers (bigger range)	64	-2^{63} through $2^{63}-1$	long wait; wait = 42L;
float	Floating-point numbers	32	$\pm1.5 \times 10^{-45}$ through $\pm3.4 \times 10^{38}$	float away; away = 0.42F;
double	Double-precision (more accurate) floating-point numbers	64	$\pm5.0 \times 10^{-324}$ through $\pm1.7 \times 10^{308}$	double trouble; trouble = 0.42;
decimal	Monetary values	128	28 significant figures	decimal coin; coin = 0.42M;
string	Sequence of characters	16 bits per character	Not applicable	string vest; vest = "forty two";
char	Single character	16	0 through $2^{16}-1$	char grill; grill = 'x';
bool	Boolean	8	True or false	bool teeth; teeth = false;

Unassigned local variables

- When you declare a variable, it contains a random value until you assign a value to it.
- This behaviour was a rich source of bugs in C and C++ programs that created a variable and accidentally used it as a source of information before giving it a value.
- C# does not allow you to use an unassigned variable.
- You must assign a value to a variable before you can use it; otherwise, your program will not compile. This requirement is called the ***definite assignment rule***.
- For example, the following statements generate the compile-time error message "Use of unassigned local variable 'age'" because the *Console. WriteLine* statement attempts to display the value of an uninitialized variable:

```
int age;
Console.WriteLine(age); // compile-time error
```

Using arithmetic operators

- C# supports the regular arithmetic operations you learned in your childhood: the plus sign (+) for addition, the minus sign (–) for subtraction, the asterisk (*) for multiplication, and the forward slash (/) for division.

- The symbols +, –, *, and / are called *operators* because they "operate" on values to create new values.
  In the following example, the variable *moneyPaidToConsultant* ends up holding the product of 750 (the daily rate) and 20 (the number of days the consultant was employed):

```
long moneyPaidToConsultant;
moneyPaidToConsultant = 750 * 20;
```

Operators and types

- Not all operators are applicable to all data types.
- The operators that you can use on a value depend on the value's type.
  For example, you can use all the arithmetic operators on values of type *char, int, long, float, double*, or *decimal.*
- However, with the exception of the plus operator, +, you can't use the arithmetic operators on values of type *string*, and you cannot use any of them with values of type *bool*.
  So, the following statement is not allowed, because the *string* type does not support the minus operator (subtracting one string from another is meaningless):

```
// compile-time error
Console.WriteLine("Gillingham" - "Forest Green Rovers");
```

- However, you can use the + operator to concatenate string values. You need to be careful because this can have unexpected results. For example, the following statement writes "431" (not "44") to the console:

```
Console.WriteLine("43" + "1");
```

- You should also be aware that the type of the result of an arithmetic operation depends on the type of the operands used.
  For example, the value of the expression 5.0/2.0 is 2.5; the type of both operands is *double*, so the type of the result is also *double*.

- (In C#, literal numbers with decimal points are always *double*, not *float*, to maintain as much accuracy as possible.) However, the value of the expression 5/2 is 2. In this case, the type of both operands is *int*, so the type of the result is also *int*.
- C# always rounds toward zero in circumstances like this. The situation gets a little more complicated if you mix the types of the operands. For example, the expression 5/2.0 consists of an *int* and a *double*.
- The C# compiler detects the mismatch and generates code that converts the *int* into a *double* before performing the operation.
- The result of the operation is therefore a *double* (2.5). However, although this works, it is considered poor practice to mix types in this way.
- C# also supports one less-familiar arithmetic operator: the *remainder,* or *modulus,* operator, which is represented by the percent sign (%).
- The result of *x % y* is the remainder after dividing the value *x* by the value *y*. So, for example, 9 % 2 is 1 because 9 divided by 2 is 4, remainder 1.

Note: If you are familiar with C or C++, you know that you can't use the remainder operator on *float* or *double* values in these languages. However, C# relaxes this rule. The remainder operator is valid with all numeric types, and the result is not necessarily an integer. For example, the result of the expression 7.0 % 2.4 is 2.2.

Numeric types and infinite values

- There are one or two other features of numbers in C# about which you should be aware.
- For example, the result of dividing any number by zero is infinity, which is outside the range of the *int, long*, and *decimal* types; consequently, evaluating an expression such as 5/0 results in an error.
- However, the *double* and *float* types actually have a special value that can represent infinity, and the value of the expression 5.0/0.0 is ***Infinity***.
- The one exception to this rule is the value of the expression 0.0/0.0.
- Usually, if you divide zero by anything, the result is zero, but if you divide anything by zero the result is infinity.

- The expression 0.0/0.0 results in a paradox—the value must be zero and infinity at the same time. C# has another special value for this situation called ***NaN***, which stands for "**not a number**."
- So if you evaluate 0.0/0.0, the result is *NaN*.
- ***NaN*** and ***Infinity*** propagate through expressions.
    - 10 + *NaN*= *NaN*
    - 10 + *Infinity*= *Infinity*.
- The one exception to this rule is the case when you multiply *Infinity* by 0.
- The value of the expression
    - *Infinity* * 0=0
    - *NaN* * 0 is *NaN*.

Controlling precedence

Precedence governs the order in which an expression's operators are evaluated. Consider the following expression, which uses the + and * operators:

*2 + 3 * 4*

This expression is potentially ambiguous: do you perform the addition first or the multiplication?

The order of the operations matters because it changes the result:

- If you perform the addition first, followed by the multiplication, the result of the addition (2 + 3) forms the left operand of the * operator, and the result of the whole expression is 5 * 4, which is 20.
- If you perform the multiplication first, followed by the addition, the result of the multiplication
  (3 * 4) forms the right operand of the + operator, and the result of the whole expression is
  2 + 12, which is 14.

In C#, the multiplicative operators (*, /, and %) have precedence over the additive operators (+ and –), so in expressions such as 2 + 3 * 4, the multiplication is performed first, followed by the addition. The answer to 2 + 3 * 4 is therefore 14.

You can use parentheses to override precedence and force operands to bind to operators in a different way. For example, in the following expression, the parentheses force the 2 and the 3 to bind to the + operator (making 5), and the result of this addition forms the left operand of the * operator to produce the value 20:

*(2 + 3) * 4*

Using associativity to evaluate expressions

Operator precedence is only half the story. What happens when an expression contains different operators that have the same precedence? This is where *associativity* becomes important. Associativity is the direction (left or right) in which the operands of an operator are evaluated. Consider the following expression that uses the / and * operators:

4 / 2 * 6

At first glance, this expression is potentially ambiguous. Do you perform the division first or the multiplication? The precedence of both operators is the same (they are both multiplicative), but the order in which the operators in the expression are applied is important because you can get two different results:

- If you perform the division first, the result of the division (4/2) forms the left operand of the * operator, and the result of the whole expression is (4/2) * 6, or 12.
- If you perform the multiplication first, the result of the multiplication (2 * 6) forms the right operand of the / operator, and the result of the whole expression is 4/(2 * 6), or 4/12.

In this case, the associativity of the operators determines how the expression is evaluated. The * and / operators are both left-associative, which means that the operands are evaluated from left to right. In this case, 4/2 will be evaluated before multiplying by 6, giving the result 12.

Associativity and the assignment operator

In C#, the equal sign (=) is an operator. All operators return a value based on their operands. The assignment operator = is no different. It takes two operands: the operand on the right side is evaluated and then stored in the operand on the left side. The value of the assignment operator is the value that was assigned to the left operand. For example, in the following assignment statement, the value returned by the assignment operator is 10, which is also the value assigned to the variable *myInt*:

```
int myInt;
myInt = 10; // value of assignment expression is 10
```

At this point, you might be thinking that this is all very nice and esoteric, but so what? Well, because the assignment operator returns a value, you can use this same value with another occurrence of the assignment statement, like this:

```
int myInt;
int myInt2;
myInt2 = myInt = 10;
```

The value assigned to the variable *myInt2* is the value that was assigned to *myInt*. The assignment statement assigns the same value to both variables. This technique is useful if you want to initialize several variables to the same value. It makes it very clear to anyone reading your code that all the variables must have the same value:

```
myInt5 = myInt4 = myInt3 = myInt2 = myInt = 10;
```

From this discussion, you can probably deduce that the assignment operator associates from right to left. The rightmost assignment occurs first, and the value assigned propagates through the variables from right to left. If any of the variables previously had a value, it is overwritten by the value being assigned.

You should treat this construct with caution, however. One frequent mistake that new C# programmers make is to try to combine this use of the assignment operator with variable declarations. For example, you might expect the following code to create and initialize three variables with the same value (10):

```
int myInt, myInt2, myInt3 = 10;
```

This is legal C# code (because it compiles). What it does is declare the variables *myInt*, *myInt2*, and *myInt3*, and initialize *myInt3* with the value 10. However, it does not initialize *myInt* or *myInt2*. If you try to use *myInt* or *myInt2* in an expression such as

```
myInt3 = myInt / myInt2;
```

the compiler generates the following errors:

```
Use of unassigned local variable 'myInt'
Use of unassigned local variable 'myInt2'
```

Incrementing and decrementing variables

If you want to add 1 to a variable, you can use the + operator, as demonstrated here:

```
count = count + 1;
```

However, adding 1 to a variable is so common that C# provides its own operator just for this purpose: the ++ operator. To increment the variable *count* by 1, you can write the following statement:

```
count++;
```

Similarly, C# provides the -- operator that you can use to subtract 1 from a variable, like this:

```
count--;
```

The ++ and -- operators are *unary* operators, meaning that they take only a single operand. They share the same precedence and are both left-associative.

Prefix and postfix

The increment (++) and decrement (--) operators are unusual in that you can place them either before or after the variable. Placing the operator symbol before the variable is called the *prefix form* of the operator, and using the operator symbol after the variable is called the *postfix form*. Here are examples:

```
count++; // postfix increment
++count; // prefix increment
count--; // postfix decrement
--count; // prefix decrement
```

Whether you use the prefix or postfix form of the ++ or -- operator makes no difference to the variable being incremented or decremented. For example, if you write *count++*, the value of *count* increases by 1, and if you write *++count*, the value of *count* also increases by 1. Knowing this, you're probably wondering why there are two ways to write the same thing. To understand the answer, you must remember that ++ and -- are operators and that all operators are used to evaluate an expression that has a value. The value returned by *count++* is the value of *count* before the increment takes place, whereas the value returned by *++count* is the value of *count* after the increment takes place.
Here is an example:

```
int x;
x = 42;
Console.WriteLine(x++); // x is now 43, 42 written out
x = 42;
Console.WriteLine(++x); // x is now 43, 43 written out
```

The way to remember which operand does what is to look at the order of the elements (the operand and the operator) in a prefix or postfix expression. In the expression *x*++, the variable *x* occurs first, so its value is used as the value of the expression before *x* is incremented. In the expression ++*x*, the operator occurs first, so its operation is performed before the value of *x* is evaluated as the result.

Declaring implicitly typed local variables

Earlier in this chapter, you saw that you declare a variable by specifying a data type and an identifier, like this:

```
int myInt;
```

It was also mentioned that you should assign a value to a variable before you attempt to use it.

You can declare and initialize a variable in the same statement, such as illustrated in the following:

```
int myInt = 99;
```

Or, you can even do it like this, assuming that *myOtherInt* is an initialized integer variable:

```
int myInt = myOtherInt * 99;
```

Now, remember that the value you assign to a variable must be of the same type as the variable.

For example, you can assign an *int* value only to an *int* variable. The C# compiler can quickly work out the type of an expression used to initialize a variable and indicate if it does not match the type of the variable. You can also ask the C# compiler to infer the type of a variable from an expression and use this type when declaring the variable by using the *var* keyword in place of the type, as demonstrated here:

```
var myVariable = 99;
var myOtherVariable = "Hello";
```

The variables *myVariable* and *myOtherVariable* are referred to as *implicitly typed* variables. The *var* keyword causes the compiler to deduce the type of the variables from the types of the expressions used to initialize them. In these examples, *myVariable* is an *int*, and *myOtherVariable* is a *string*. However, it is important for you to understand that this is a convenience for declaring variables only, and that after a variable has been declared you can assign only values of the inferred type to it—you cannot assign *float*, *double*, or *string* values to *myVariable* at a later point in your program, for example.

You should also understand that you can use the *var* keyword only when you supply an expression to initialize a variable. The following declaration is illegal and causes a compilation error:

```
var yetAnotherVariable; // Error - compiler cannot infer type
```

Quick Reference

To	Do this
Declare a variable	Write the name of the data type, followed by the name of the variable, followed by a semicolon. For example: `int outcome;`
Declare a variable and give it an initial value	Write the name of the data type, followed by the name of the variable, followed by the assignment operator and the initial value. Finish with a semicolon. For example: `int outcome = 99;`
Change the value of a variable	Write the name of the variable on the left, followed by the assignment operator, followed by the expression calculating the new value, followed by a semicolon. For example: `outcome = 42;`

Generate a string representation of the value in a variable	Call the ToString method of the variable. For example:
	`int intVar = 42;`
	`string stringVar = intVar.ToString();`

Convert a string to an int	Call the System.Int32.Parse method. For example:
	`string stringVar = "42";`
	`int intVar = System.Int32.Parse(stringVar);`

Override the precedence of an operator	Use parentheses in the expression to force the order of evaluation. For example: `(3 + 4) * 5`
Assign the same value to several variables	Use an assignment statement that lists all the variables. For example: `myInt4 = myInt3 = myInt2 = myInt = 10;`
Increment or decrement a variable	Use the ++ or -- operator. For example: `count++;`

CHAPTER-3

Writing methods and applying scope

- Declare and call methods.
- Pass information to a method.
- Return information from a method.
- Define local and class scope.
- Use the integrated debugger to step into and out of methods as they run.

Creating methods

A method is a named sequence of statements. A method has a name and a body. Method name should be a meaningful identifier that indicates the overall purpose of the method (eg: calculateIncomeTax). The method body contains the actual statements to be run when the method is called. Methods can be given some data for processing, can return information (result of the processing). Methods are a fundamental and powerful mechanism.

Declaring a method

Syntax for declaring a C# method:

```
returnType methodName (parameterList)
{
        // method body statements
}
```

returnType is the name of a type and specifies the kind of information the **method** returns as a result of its processing. Can be any type (int, string, etc). If a method does not return a value, use the keyword **void** as the return type.

The ***methodName*** is the name used to call the method. Eg: Valid method name: *addValues*

Invalid method name: *add$Values*

The ***parameterList*** is optional and describes the types and names of the information that is passed into the method for it to process.

Parameters are written between opening and closing parentheses, ( ), as though variables are declared, with the datatype followed by the parameterName. If the method has two or more parameters, separate them with commas.

The ***method body*** statements are the lines of code that are run when the method is called. Method body is enclosed between opening and closing braces, { }.

```
public class add2
{
        public static void Main()
        {
                System.Console.WriteLine(15+17);
        }
}
public class divide3
{
        public static void Main()
        {
                System.Console.WriteLine(36/6);
        }
}
```

Returning data from a method

A **return** statement is included at the end of the processing in the method body. A return statement consists of the keyword **return** followed by an **expression** that specifies the returned value, and a **semicolon**. The type of the expression must be the same as the type specified by the method declaration. Eg:, if a method returns an **int**, the return statement must return an **int**; otherwise, your program will not compile.

```
returntype methodName(datatype parameterName1, datatype parameterName2,…)
{
    // ...
    return expression;
}
int addValues(int leftHandSide, int rightHandSide)
```

```
{
    // ...
    return leftHandSide + rightHandSide;
}
```

The **return** statement is usually positioned at the end of the method because it causes the method to finish, and control returns to the statement that called the method. Any statements that occur after the return statement are not executed.

If the method doesn't return information (ie: return type is ***void)***, **use return;**

Eg:
```
void methodName( datatype  parameterName)
{
        // display the answer ...
        return;
}
```

If the method does not return anything, the ***return statement***, can also be omitted.

Eg:
```
void showResult(int answer)
{
        // display the answer ...
        return;
}
```

Calling methods

A **method** is called to perform a specific task. If **method** requires information (**parameters**), supply requested information. If the **method** returns information (as specified by its **return type**), capture the information.

Specifying the method call syntax

Syntax of C# method call:

```
result = methodName ( argumentList );
```

methodName must exactly match the name of the method being called.

result = clause is optional. **result** contains the value returned by the method. If the method **returntype** is **void** (that is, it does not return a value), you must omit the **result** = clause. If the **result** = clause is not specified and the method returns a value, the method runs but the return value is discarded.

argumentList supplies the information that the method accepts. Supply an argument for each parameter, and the value of each argument must be compatible with the type of its corresponding parameter. If the method being called has two or more parameters, separate them with commas.

Eg:

```
int addValues(int leftHandSide, int rightHandSide)
{
        // ...
}
```

addValues method has two int parameters, so it is called with two comma-separated **int** arguments.

addValues(39, 3); // okay literals as parameters

OR

int arg1 = 99;
int arg2 = 1;
addValues(arg1, arg2); // int variables as parameters

addValues; // compile-time error, no parentheses
addValues(); // compile-time error, not enough arguments
addValues(39); // compile-time error, not enough arguments
addValues("39", "3"); // compile-time error, wrong types for arguments

int result = addValues(39, 3); // on right-hand side of an assignment
showResult(addValues(39, 3)); // as argument to another method call

Applying scope

Variables are created to hold values. Can also create variables at various points in C# applications.

```
private void calculateClick(object sender, RoutedEventArgs e)
{
        int calculatedValue = 0;
        ...
}
```

Variable comes into existence at the point where it is defined and subsequent statements in the **calculateClick** method can then use this variable. It cannot be used elsewhere. A variable can be used only after it has been created. When the method has finished, this variable disappears. When a variable can be accessed at a particular location in a program, the variable is said to be in **scope at that location.**

The **calculatedValue** variable has **method scope**; it can be accessed throughout the **calculateClick** method but not outside of that method. Variables with different scope can also be defined.

Eg: define a variable outside of a method but within a class. This variable can be accessed by any method within that class. Such a variable is said to have **class scope**.

Scope of a Variable:

It is the region of the program in which a variable is usable. Scope applies to methods as well as variables. The scope of an identifier (a variable or method) is linked to the location of the declaration that introduces the identifier in the program.

Defining local scope

The opening and closing braces that form the body of a method define the scope of the method. Variables declared inside the body of a method are scoped to that method. They disappear when the method ends and can be accessed only by code running in that method. These variables are called ***local variables*** because they are local to the method in which they are declared.

```
class Example
{
```

```
            void firstMethod()
            {
                    int myVar;
                    ...
            }
            void anotherMethod()
            {
                    myVar = 42; // error - variable not in scope
                    ...
            }
    }
```

Note: This code fails to compile because ***anotherMethod*** is trying to use the variable ***myVar***, which is not in scope.

Defining class scope

The opening and closing braces that form the body of a class define the scope of that class. Any variables declared within the body of a class (but not within a method) are scoped to that class. C# term for a variable defined by a class is ***field***. Fields can be used to share information between methods.

```
class Example
{
    void firstMethod()
    {
            myField = 42; // ok
        ...
    }
    void anotherMethod()
    {
            myField++; // ok
        ...
    }
    int myField = 0;
}
    OR
```

```
class Example
{
    int myField = 0;
    void firstMethod()
    {
myField = 42; // ok
...
}
void anotherMethod()
{
myField++; // ok
...
}
}
```

The variable ***myField*** is defined in the class but outside the methods ***firstMethod*** and ***anotherMethod***. Therefore, ***myField*** has class scope and is available for use by all methods in that class.

In a method, you must declare a variable before you can use it. Fields are a little different. A method can use a field before the statement that defines the field.

Overloading methods

Overloading is primarily useful when there is need to perform the same operation on different data types or varying groups of information. Can overload a method when the different implementations have different sets of parameters—i.e: when they have the same name but a different number of parameters, or when the types of the parameters differ.

When a method is called, a comma-separated list of arguments is also supplied, and the number and type of the arguments are used by the compiler to select one of the overloaded methods. Can't overload the return type of a method. i.e can't declare two methods with the same name that differ only in their return type.

Using optional parameters and named arguments

A key feature of C# and other languages designed for the .NET Framework is the ability to interoperate with applications and components written using other technologies. One of the principal technologies that underpins many Microsoft Windows applications and services running outside of the .NET Framework is the Component Object Model (COM).

The Common Language Runtime (CLR) used by the .NET Framework is also heavily dependent on COM, as is the Windows Runtime of Windows 8 and Windows 8.1. COM does not support overloaded methods; instead, it uses methods that can take optional parameters. To make it easier to incorporate COM libraries and components into a C# solution, C# also supports **optional parameters**.

Optional parameters are also useful in other situations. They provide a compact and simple solution when it is not possible to use overloading because the types of the parameters do not vary sufficiently to enable the compiler to distinguish between implementations.

```
public void doWorkWithData(int intData1, float floatData, int intData2)
{
    ...
}
```

To provide an implementation of doWorkWithData that took only two parameters: intData and floatData, overload the method:

```
public void doWorkWithData(int intData, float floatData)
{
        ...
}
int arg1 = 99;
float arg2 = 100.0F;
int arg3 = 101;
doWorkWithData(arg1, arg2, arg3); // Call overload with three parameters
doWorkWithData(arg1, arg2); // Call overload with two parameters
```

Suppose that you want to implement two further versions of **DoWorkWithData** that take only the first parameter and the third parameter.

```
public void doWorkWithData(int intData)
{
        ...
}
public void doWorkWithData(int moreIntData)
{
        ...
}
```

To the compiler, these two overloads appear identical. The code fails to compile and will instead generate the error "Type 'typename' already defines a member called '**doWorkWithData'** with the same parameter types."

```
int arg1 = 99;
int arg3 = 101;
doWorkWithData(arg1);
doWorkWithData(arg3);
```

Which overload or overloads would the calls to **doWorkWithData** invoke? Optional parameters and named arguments can help to solve this problem.

Defining optional parameters

Specify that a parameter is optional when defining a method by providing a default value for the parameter and indicate a default value by using the assignment operator. Eg: In the optMethod method, the first parameter is mandatory because it does not specify a default value, but the second and third parameters are optional:

```
void optMethod(int first, double second = 0.0, string third = "Hello")
{
    ...
}
```

Must always specify all mandatory parameters before any optional parameters.

optMethod(99, 123.45, "World"); // Arguments provided for all three parameters
optMethod(100, 54.321); // Arguments provided for first two parameters only

Passing Named Arguments

By default, C# uses the position of each argument in a method call to determine to which parameters they apply. Hence, the second example method shown in the previous section passes the two arguments to the first and second parameters in the **optMethod** method because this is the order in which they occur in the method declaration. With C#, parameters can also be specified by name. This feature lets the programmer pass the arguments in a different sequence. To pass an argument as a named parameter, specify the parameter name, followed by a colon and the value to use.

optMethod(first : 99, second : 123.45, third : "World");
optMethod(first : 100, second : 54.321);

Named arguments gives the programmer the ability to pass arguments in any order.

optMethod(third : "World", second : 123.45, first : 99);
optMethod(second : 54.321, first : 100);

This feature also makes it possible for the programmer to omit arguments. Using this feature the programmer can call the **optMethod** method and specify values for the first and third parameters only and use the default value for the second parameter:

optMethod(first : 99, third : "World");

Positional and named arguments can be mixed, but all the positional arguments must be specified before the first named arguments.

optMethod(99, third : "World"); *// First argument is positional*

Resolving ambiguities with optional parameters and named arguments

Using optional parameters and named arguments can result in some possible ambiguities in the code. The programmer needs to understand how the compiler resolves these ambiguities; otherwise, he might find his applications behaving in unexpected ways. Suppose the optMethod method is defined as an overloaded method, as shown:

```
void optMethod(int first, double second = 0.0, string third = "Hello")
{
        ...
}
void optMethod(int first, double second = 1.0, string third = "Goodbye", int fourth = 100 )
{
        ...
}
```

This is perfectly legal C# code that follows the rules for overloaded methods. The compiler can distinguish between the methods because they have different parameter lists. However, problem arises if the programmer attempts to call the **optMethod** method and omit some of the arguments corresponding to one or more of the optional parameters:

optMethod(1, 2.5, "World"); // first method is called

Again, this is perfectly legal code, but which version of the **optMethod** method does it run? The answer is that it runs the version that most closely matches the method call, so it invokes the method that takes three parameters and not the version that takes four. Consider the next example:

optMethod(1, fourth : 101); // second method is called

In this code, the call to **optMethod** omits arguments for the second and third parameters, but it specifies the fourth parameter by name. Only one version of **optMethod** matches this call, so this is not a problem. This next one will get you thinking, though:

optMethod(1, 2.5); // ambiguous; does not compile

This time, neither version of the **optMethod** method exactly matches the list of arguments provided. Both versions of the **optMethod** method have optional parameters for the second, third, and fourth arguments. So, does this statement call the version of **optMethod** that takes three parameters and use the default value for the third parameter, or does it call the version of **optMethod** that takes four parameters and use the default value for the third and fourth parameters? The answer is that it does neither. This is an unresolvable ambiguity, and the compiler does not let the application compile. The same situation arises with the same result if you try to call the **optMethod** method as shown in any of the following statements:

optMethod(1, third : "World"); *// ambiguous; does not compile*

optMethod(1); *// ambiguous; does not compile*

optMethod(second : 2.5, first : 1); *// ambiguous; does not compile*

CHAPTER-4

Using Decision Statements

Declaring Boolean variables

A Boolean expression always evaluates to true or false. Visual C# provides a data type called bool. A bool variable can hold one of two values: true or false.

```
bool areYouReady;
areYouReady = true;
Console.WriteLine(areYouReady); // writes True to console
```

Using Boolean operators

A Boolean operator is an operator that performs a calculation whose result is either **true** or **false**. C# has several very useful Boolean operators, **NOT** operator, represented by **!**. The **!** operator negates a Boolean value, yielding the opposite of that value. Eg: if the value of the variable **areYouReady** is **true**, the value of the expression **!areYouReady** is **false**.

Understanding Equality and Relational operators

Two Boolean operators that you will frequently use are **equality** (==*)* and **inequality** (**!=**). They are binary operators which determine whether one value is same as another value of the same type, yielding a Boolean result.

Equality Operators

```
int marks = 20;
```

Operator	Meaning	Example	Outcome if marks is 20
==	Equal to	marks== 100	false
!=	Not equal to	marks != 0	true

Relational Operators

Operator	Meaning	Example	Outcome if marks is 20
<	Less than	marks< 21	false
<=	Less than or equal to	marks< 18	false
>	Greater than	marks< 16	true
>=	Greater than or equal to	marks< 30	true

Understanding Conditional Logical Operators

C# also provides two other binary Boolean operators: the **logical AND operator**, represented by the **&&** symbol, and the **logical OR operator**, represented by the || symbol.

The purpose of conditional logical operators is to combine two Boolean expressions or values into a single Boolean result. Similar to the equality and relational operators in that the value of the expressions in which they appear is either true or false, but they differ in that the values on which they operate must also be either true or false. The outcome of the && operator is true if and only if both of the Boolean expressions it's evaluating are true.

```
bool validPercentage;
validPercentage = (percent >= 0) && (percent <= 100);
percent >= 0 && <= 100 // this statement will not compile
validPercentage = percent >= 0 && percent <= 100
```

and

```
validPercentage = (percent >= 0) && (percent <= 100)
```

The outcome of the || operator is true if either of the Boolean expressions it evaluates is true. || operator is used to determine whether any one of a combination of Boolean expressions is true.

```
bool invalidPercentage;
invalidPercentage = (percent < 0) || (percent > 100);
```

Short-circuiting

The **&&** and || operators both exhibit a feature called **short-circuiting**. It is not necessary to evaluate both operands when ascertaining the result of a conditional logical expression.

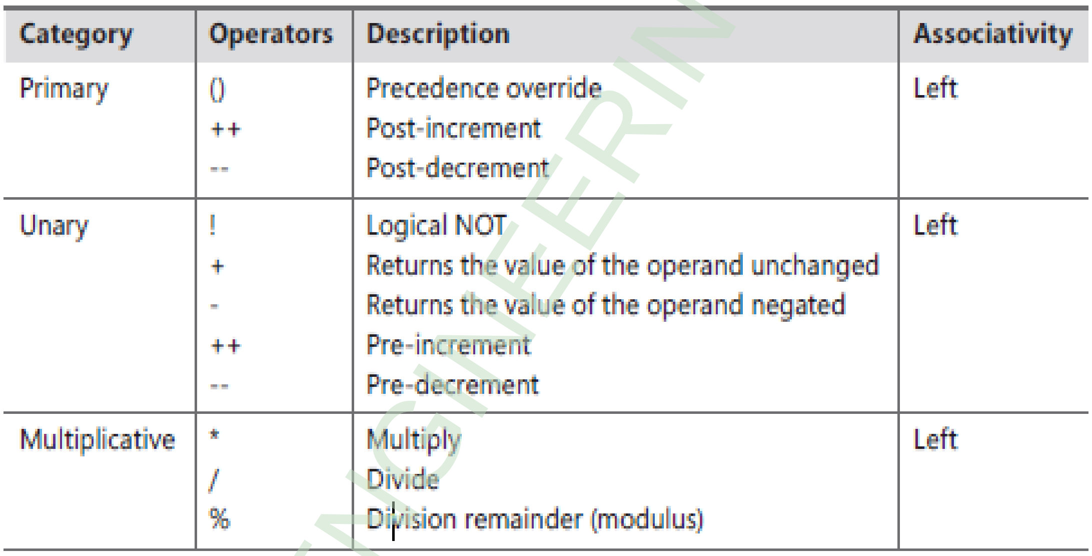
(percent >= 0) *&& (percent <= 100)*

(percent < 0) *|| (percent > 100)*

Carefully design expressions that use the conditional logical operators, you can boost the performance of your code by avoiding unnecessary work.

Place simple Boolean expressions that can be evaluated easily on the left side of a conditional logical operator, and put more complex expressions on the right side.

Summarizing Operator Precedence and Associativity



Category	Operators	Description	Associativity
Primary	() ++ --	Precedence override Post-increment Post-decrement	Left
Unary	! + - ++ --	Logical NOT Returns the value of the operand unchanged Returns the value of the operand negated Pre-increment Pre-decrement	Left
Multiplicative	* / %	Multiply Divide Division remainder (modulus)	Left

Category	Operators	Description	Associativity
Additive	+ -	Addition Subtraction	Left
Relational	< <=	Less than Less than or equal to	Left
	>	Greater than	
	>=	Greater than or equal to	
Equality	== !=	Equal to Not equal to	Left
Conditional AND	&&	Conditional AND	Left
Conditional OR	||	Conditional OR	Left
Assignment	=	Assigns the right-hand operand to the left and returns the value that was assigned	Right

Using *if* statements to make decisions

Understanding *if statement syntax*

if statement syntax (**if** and **else** are C# keywords)

if (booleanExpression)

    statement-1;

else

    statement-2;

If **booleanExpression evaluates** to **true**, **statement-1** runs; otherwise, **statement-2** runs. The **else** keyword and the subsequent **statement-2** are **optional**. If there is no else clause and the **booleanExpression** is **false**, execution **continues** with **code following** the **if** statement. Also, notice that the **Booleanexpression** must be enclosed in **parentheses**; otherwise, the code will not compile. For example, here's an if statement that increments a variable representing the second hand of a stopwatch. If the value of the seconds variable is 59, it is reset to 0; otherwise, it is incremented by using the ++ operator:

Eg:

int seconds;

...

if (seconds == 59)

```
        seconds = 0;
else
        seconds++;
```

The **expression** in an **if** statement must be enclosed in **parentheses**. The **expression** must be a **Booleanexpression** only.

```
int seconds;
...
if (seconds = 59) // compile-time error
...
if (seconds == 59) // ok
bool inWord;
...
if (inWord == true) // ok, but not commonly used
...
if (inWord) // more common and considered better style
```

Using Blocks to Group Statements

A **block** is simply a **sequence of statements grouped** between an **opening brace** and a **closing brace**. A **block** also **starts** a **new scope**.

```
int seconds = 0;
int minutes = 0;
...
if (seconds == 59)
{
    seconds = 0;
    minutes++;
}
else
{
    seconds++;
}
```

It is good practice to always define the statements for each branch of an if statement within a block, even if a block consists of only a single statement.

Variables can be defined inside a block, but they will disappear **at** the end of the block.

```
if (...)
{
        int myVar = 0; // myVar can be used here
        ...
} // myVar disappears here
else
{ // myVar cannot be used here
        ...
}// myVar cannot be used here
```

Cascading if statements

if statements can be nested inside other if statements.

```
if (day == 0)
        { dayName = "Sunday“;}
else if (day == 1)
        { dayName = "Monday";}
else if (day == 2)
        { dayName = "Tuesday";}
else if (day == 3)
        { dayName = "Wednesday";}
else if (day == 4)
        { dayName = "Thursday";}
else if (day == 5)
        { dayName = "Friday";}
else if (day == 6)
        { dayName = "Saturday";}
else
        { dayName = "unknown";}
```

Using switch statements

The following nested if statement can be

```
if (day == 0)
{ dayName = "Sunday"; }
else if (day == 1)
{ dayName = "Monday";}
else if (day == 2)
{ dayName = "Tuesday";}
else if (day == 3)
{ ...}
else
{ dayName = "Unknown";}
```

rewritten as a switch statement to make the program more efficient and more readable.

Understanding switch statement syntax

Syntax of a **switch** statement (**switch**, **case**, and **default** are keywords)

```
switch ( controllingExpression )
{
        case constantExpression : statements
                        break;
        case constantExpression : statements
                break;
        ...
        default : statements
                break;
}
```

The **controllingExpression**(or **case label**), which must be **enclosed** in **parentheses**, is **evaluated once**. Control then jumps to the block of code identified by the **constantExpression**, whose value is equal to the result of the **controllingExpression**. Execution runs as far as the break statement, at which point the switch statement finishes and the program continues at the first statement that follows the closing brace of the switch statement. If none of the **constantExpression** values is equal to the value of the **controllingExpression**, the statements below the optional default label run.

```
switch (day)
{
    case 0 : dayName = "Sunday";
                break;
    case 1 : dayName = "Monday";
                break;
    case 2 : dayName = "Tuesday";
                break;
                ...
    default : day       Name = "Unknown";
                break;
}
```

Following the switch statement rules

- Use switch only on certain data types, such as **int**, **char**, or **string**. With any other types (including **float** and **double**), you must use an **if** statement.
- The case labels must be constant expressions, such as 42 if the switch data type is an int, ‘4’ if the switch data type is a char, or “42” if the switch data type is a string.
- Case labels must be unique expressions.

```
switch (trumps)
{
        case Hearts :
        case Diamonds :     // Fall-through allowed - no code between labels
                color = "Red";    // Code executed for Hearts and Diamonds
                break;
            case Clubs :
                color = "Black";
            case Spades :       // Error - code between labels
                color = "Black";
                break;
        }
```

CHAPTER 5
USING COMPOUND ASSIGNMENT AND ITERATION STATEMENTS

Using compound assignment operators

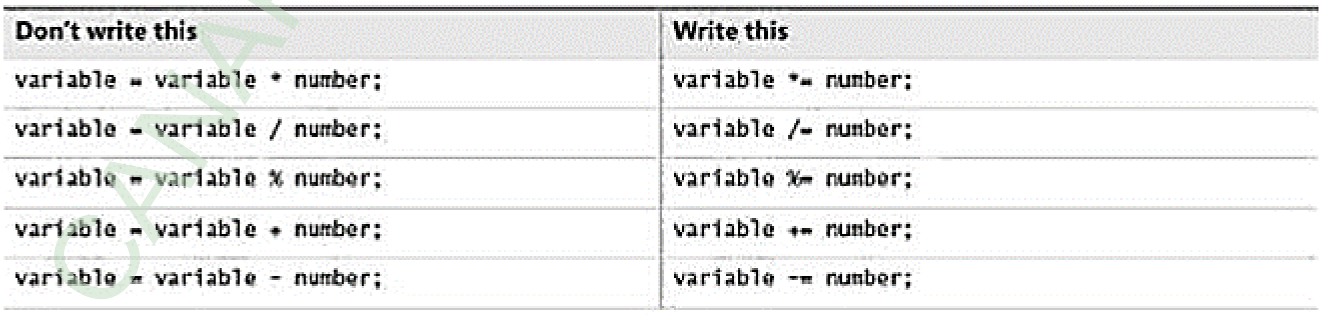
- You've already seen how to use arithmetic operators to create new values.
- For example, the following statement uses the plus operator (+) to display to the console a value that is 42 greater than the variable answer.

  Console.WriteLine(answer + 42);
- You've also seen how to use assignment statements to change the value of a variable.
- The following statement uses the assignment operator (=) to change the value of answer to 42:

  answer = 42;
- If you want to add 42 to the value of a variable, you can combine the assignment operator and the plus operator.
- For example, the following statement adds 42 to answer.
- After this statement runs, the value of answer is 42 more than it was before:

  answer = answer + 42;
- Although this statement works, you'll probably never see an experienced programmer write code like this.
- Adding a value to a variable is so common that C# provides a way for you to perform this task in a shorthand manner by using the operator +=.
- To add 42 to answer, you can write the following statement:

  answer += 42;
- You can use this notation to combine any arithmetic operator with the assignment operator, as the following table shows.
- These operators are collectively known as the compound assignment operators.



Don't write this	Write this
`variable = variable * number;`	`variable *= number;`
`variable = variable / number;`	`variable /= number;`
`variable = variable % number;`	`variable %= number;`
`variable = variable + number;`	`variable += number;`
`variable = variable - number;`	`variable -= number;`

- The += operator also works on strings; it appends one string to the end of another.
- For example, the following code displays “Hello John” on the console:

```
string name = "John";
string greeting = "Hello  ”;
greeting += name;
Console.WriteLine(greeting);
```

- You cannot use any of the other compound assignment operators on strings.

Writing while statements

- You use a while statement to run a statement repeatedly for as long as some condition is true.
- The syntax of a while statement is as follows:

```
while ( booleanExpression )
      statement
```

- The Boolean expression (which must be enclosed in parentheses) is evaluated, and if it is true, the statement runs and then the Boolean expression is evaluated again, if the expression is still true, the statement is repeated, and then the Boolean expression is evaluated yet again.
- This process continues until the Boolean expression evaluates to false, at which point the while statement exits.
- Execution then continues with the first statement that follows the while statement.
- A while statement shares the following syntactic similarities with an if statement (in fact, the syntax is identical except for the keyword):
- The expression must be a Boolean expression.
- The Boolean expression must be written within parentheses.
- If the Boolean expression evaluates to false when first evaluated, the statement does not run.
- If you want to perform two or more statements under the control of a while statement, you must use braces to group those statements in a block.
- Here’s a while statement that writes the values 0 through 9 to the console.
- Note that as soon as the variable /' reaches the value 10, the while statement finishes and the code in the statement block does not run:

```
int i = 0;
while (i < 10)
{
        Console.WriteLine(i);
        i++;
}
```

- All while statements should terminate at some point.
- A common beginner's mistake is to forget to include a statement to cause the Boolean expression eventually to evaluate to false and terminate the loop, which results in a program that runs forever.
- In the example, the statement /'++; performs this role.

Writing for statements

In C#, most while statements have the following general structure:

```
initialization
while (Boolean expression)
{
        statement
        update control variable
}
```

- The for statement in C# provides a more formal version of this kind of construct by combining the initialization, Boolean expression, and code that updates the control variable.
- You'll find the for statement useful because in a for statement, it is much harder to accidentally leave out the code that initializes or updates the control variable, so you are less likely to write code that loops forever.
- Here is the syntax of a for statement:

```
for (initialization; Boolean expression; update control variable)
        statement
```

- The statement that forms the body of the for construct can be a single line of code or a code block enclosed in braces.
- You can rephrase the while loop shown earlier that displays the integers from 0 through 9 as the following for loop:

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

- The initialization occurs just once, at the very beginning of the loop.
- Then, if the Boolean expression evaluates to true, the statement runs.
- The control variable update occurs, and then the Boolean expression is reevaluated.
- If the condition is still true, the statement is executed again, the control variable is updated, the Boolean expression is evaluated again, and so on.
- Notice that the initialization occurs only once, that the statement in the body of the loop always executes before the update occurs, and that the update occurs before the Boolean expression reevaluates.
- You can omit any of the three parts of a for statement.
- If you omit the Boolean expression, it defaults to true, so the following for statement runs forever:

```
for (int i = 0; i++)
{
        Console.WriteLine("somebody stop me!");
}
```

- If you omit the initialization and update parts, you have a strangely spelled while loop:

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

- You can also provide multiple initializations and multiple updates in a for loop.
- (You can have only one Boolean expression, though.)
- To achieve this, separate the various initializations and updates with commas, as shown in the following example:

```
for (int i = 0, j = 10; i <= j; i++, j--)
{
        …
}
```

As a final example, here is the while loop from the preceding exercise recast as a for loop:

```
for (string line = reader.ReadLine( ); line != null; line = reader.ReadLine( ))
{
        source.Text += line + '\n';
}
```

Understanding for statement scope

- You might have noticed that you can declare a variable in the initialization part of a ***for*** statement.
- That variable is scoped to the body of the ***for*** statement and disappears when the for statement finishes.
- This rule has two important consequences.
- First, you cannot use that variable after the ***for*** statement has ended because it's no longer in scope.
- Here's an example:

```
        for (int i = 0; i < 10; i++)
        {
                …
        }
        Console.WriteLine(i); // compile-time error
```

- Second, you can write two or more for statements that reuse the same variable name because each variable is in a different scope, as shown in the following code:

```
for (int i = 0; i<10; i ++)
{
        …
}
for (int i = 0; i <20; i += 2) //OKAY
{
        …
}
```

Writing do statements

- Both the while and for statements test their Boolean expression at the beginning of the loop.
- This means that if the expression evaluates to false on the first test, the body of the loop does not run—not even once.
- The do statement is different: its Boolean expression is evaluated after each iteration, so the body always executes at least once.
- The syntax of the do statement is as follows (don't forget the final semicolon):

```
do
        statement
while (booleanExpression);
```

- You must use a statement block if the body of the loop contains more than one statement (the compiler will report a syntax error if you don't).
- Here's a version of the example that writes the values 0 through 9 to the console, this time constructed by using a do statement:

```
int i = 0;
do
{
        Console.WriteLine(i);
        i++;
}
while (i < 10);
```

The break and continue statements

- The **break** statement terminates the closest enclosing loop or switch statement in which it appears. Control is passed to the statement that follows the terminated statement, if any.
- When you break out of a loop, the loop exits immediately and execution continues at the first statement that follows the loop.
- Neither the update nor the continuation condition of the loop is rerun.
- In contrast, the continue statement causes the program to perform the next iteration of the loop immediately (after re-evaluating the Boolean expression).

- Here’s another version of the example that writes the values 0 through 9 to the console, this time using break and continue statements:

```
int i = 0;
while (true)
{
    Console.WriteLine(i);
    i++;
    if (i < 10)
        continue;
    else
        break;
}
```

- This code is absolutely ghastly.
- Many programming guidelines recommend using continue cautiously or not at all because it is often associated with hard-to-understand code.
- The behavior of continue is also quite subtle.
- For example, if you execute a continue statement from within a for statement, the update part runs before performing the next iteration of the loop.

CHAPTER 6
MANAGING ERRORS AND EXCEPTIONS

Coping with errors

- In the world of computers, hard disks become corrupt, other applications running on the same computer as your program run amok and use up all the available memory, wireless network connections disappear at the most awkward moment, and even natural phenomena such as a nearby lightning strike can have an impact if it causes a power outage or network failure.
- Errors can occur at almost any stage when a program runs, and many of them might not actually be the fault of your own application, so how do you detect them and attempt to recover?
- Over the years, a number of mechanisms have evolved.
- A typical approach adopted by older systems such as UNIX involved arranging for the operating system to set a special global variable whenever a method failed.
- Then, after each call to a method, you checked the global variable to see whether the method succeeded.
- C# and most other modern object-oriented languages don't handle errors in this manner; it's just too painful.
- Instead, they use exceptions.
- If you want to write robust C# programs, you need to know about exceptions.

Trying code and catching exceptions

- Errors can happen at any time, and using traditional techniques to manually add error-detecting code around every statement is cumbersome, time consuming, and error prone in its own right.
- You can also lose sight of the main flow of an application if each statement requires contorted error-handling logic to manage each possible error that can occur at every stage.
- Fortunately, C# makes it easy to separate the error-handling code from the code that implements the primary logic of a program by using exceptions and exception handlers.
- To write exception-aware programs, you need to do two things:
- Write your code within a try block (itry is a C# keyword).

- When the code runs, it attempts to execute all the statements in the try block, and if none of the statements generates an exception, they all run, one after the other, to completion.
- However, if an error condition occurs, execution jumps out of the try block and into another piece of code designed to catch and handle the exception—a catch handler.
- Write one or more catch handlers (catch is another C# keyword) immediately after the try block to handle any possible error conditions.
- A catch handler is intended to capture and handle a specific type of exception, and you can have multiple catch handlers after a try block, each one designed to trap and process a specific exception; you can provide different handlers for the different errors that could arise in the try block.
- If any one of the statements within the try block causes an error, the runtime throws an exception.
- The runtime then examines the catch handlers after the try block and transfers control directly to the first matching handler.
- Here's an example of code in a try block that attempts to convert strings that a user has typed in some text boxes on a form to integer values, call a method to calculate a value, and write the result to another text box.
- Converting a string to an integer requires that the string contain a valid set of digits and not some arbitrary sequence of characters.
- If the string contains invalid characters, the int.Parse method throws a FormatException, and execution transfers to the corresponding catch handler.
- When the catch handler finishes, the program continues with the first statement that follows the handler.
- Note that if there is no handler that corresponds to the exception, the exception is said to be unhandled (this situation will be described shortly).

```
try
{
    int leftHandSide = int.Parse(IhsOperand.Text);
    int rightHandSide = int.Parse(rhsOperand.Text);
    int answer = doCalculation(leftHandSide, rightHandSide);
    result.Text = answer. ToStringO;
}
```

```
catch (FormatException fEx)
{
    // Handle the exception
    ...
}
```

- A catch handler employs syntax similar to that used by a method parameter to specify the exception to be caught.
- In the preceding example, when a *FormatException* is thrown, the *fEx* variable is populated with an object containing the details of the exception.
- The *FormatException* type has a number of properties that you can examine to determine the exact cause of the exception.
- Many of these properties are common to all exceptions.
- For example, the Message property contains a text description of the error that caused the exception.
- You can use this information when handling the exception, perhaps recording the details to a log file or displaying a meaningful message to the user and then asking the user to try again.

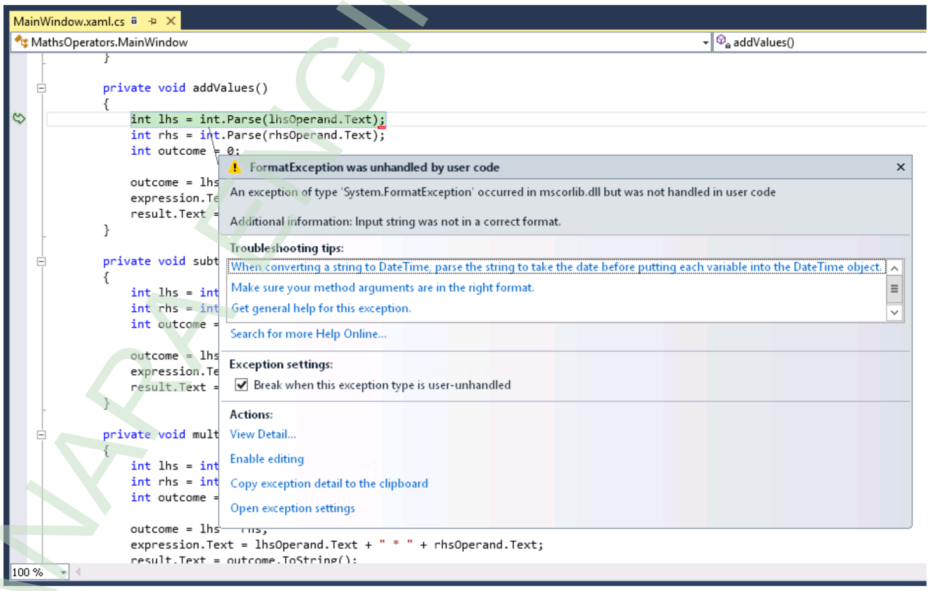
Unhandled Exceptions

- What happens if a try block throws an exception and there is no corresponding catch handler?
- In the previous example, it is possible that the *lhsOperand* text box contains the string representation of a valid integer but the integer it represents is outside the range of valid integers supported by C# (for example, "2147483648").
- In this case, the int.Parse statement throws an *OverflowException*, which will not be caught by the *FormatException* catch handler.
- If this occurs and the try block is part of a method, the method immediately exits and execution returns to the calling method.
- If the calling method uses a try block, the runtime attempts to locate a matching catch handler for this try block and execute it.

- If the calling method does not use a try block or there is no matching catch handler, the calling method immediately exits and execution returns to its caller, where the process is repeated.
- If a matching catch handler is eventually found, the handler runs and execution continues with the first statement that follows the catch handler in the catching method.

Important

- Notice that after catching an exception, execution continues in the method containing the catch block that caught the exception.
- If the exception occurred in a method other than the one containing the catch handler, control does not return to the method that caused the exception.
- If, after cascading back through the list of calling methods, the runtime is unable to find a matching catch handler, the program terminates with an unhandled exception.
- You can easily examine exceptions generated by your application.
- If you are running the application in Microsoft Visual Studio 2013 in debug mode (that is, on the Debug menu you selected Start Debugging to run the application) and an exception occurs, a dialog box similar to the one shown in the following image appears and the application pauses, helping you to determine the cause of the exception:



```
        }

        private void addValues()
        {
            int lhs = int.Parse(lhsOperand.Text);
            int rhs = int.Parse(rhsOperand.Text);
            int outcome = 0;

            outcome = lhs
            expression.Te
            result.Text =
        }

        private void subt
        {
            int lhs = int
            int rhs = int
            int outcome =

            outcome = lhs
            expression.Te
            result.Text =
        }

        private void mult
        {
            int lhs = int
            int rhs = int
            int outcome =

            outcome = lhs   rhs;
            expression.Text = lhsOperand.Text + " * " + rhsOperand.Text;
            result.Text = outcome.ToString();
```

FormatException was unhandled by user code

An exception of type 'System.FormatException' occurred in mscorlib.dll but was not handled in user code

Additional information: Input string was not in a correct format.

Troubleshooting tips:

When converting a string to DateTime, parse the string to take the date before putting each variable into the DateTime object.
Make sure your method arguments are in the right format.
Get general help for this exception.

Search for more Help Online...

Exception settings:

[✓] Break when this exception type is user-unhandled

Actions:

View Detail...
Enable editing
Copy exception detail to the clipboard
Open exception settings

- The application stops at the statement that caused the exception and you drop into the debugger.

- You can examine the values of variables, you can change the values of variables, and you can step through your code from the point at which the exception occurred by using the Debug toolbar and the various debug windows.

Using multiple catch handlers

- The previous discussion highlighted how different errors throw different kinds of exceptions to represent different kinds of failures.
- To cope with these situations, you can supply multiple catch handlers, one after the other, such as in the following:

```
try
{
        int leftHandSide = int.Parse(IhsOperand.Text);
        int rightHandSide = int.Parse(rhsOperand.Text);
        int answer = doCalculation(leftHandSide, rightHandSide);
        result.Text = answer. ToString( );
}
catch (FormatException fEx)
{
                //...
}
catch (OverflowException oEx)
{
        //...
}
```

- If the code in the try block throws a *FormatException* exception, the statements in the catch block for the *FormatException* exception run.
- If the code throws an *OverflowException* exception, the catch block for the *OverflowException* exception runs.

Catching multiple exceptions

- The exception-catching mechanism provided by C# and the Microsoft .NET Framework is quite comprehensive.
- The .NET Framework defines many types of exceptions, and any programs you write can throw most of them.

- It is highly unlikely that you will want to write catch handlers for every possible exception that your code can throw—remember that your application must be able to handle exceptions that you might not have even considered when you wrote it! So, how do you ensure that your programs catch and handle all possible exceptions?
- The answer to this question lies in the way the different exceptions are related to one another.
- Exceptions are organized into families called inheritance hierarchies.
- *FormatException* and *OverflowException* both belong to a family called *SystemException*, as do a number of other exceptions.
- *SystemException* is itself a member of a wider family simply called *Exception*, and this is the great-granddaddy of all exceptions.
- If you catch *Exception*, the handler traps every possible exception that can occur.
- The next example shows how to catch all possible exceptions:

```
try
{
    int leftHandSide = int.Parse(lhsOperand.Text);
    int rightHandSide = int.Parse(rhsOperand.Text);
    int answer = doCalculation(leftHandSide, rightHandSide);
    result.Text = answer.ToString( );
}
catch (Exception ex) // this is a general catch handler
{
        //. . .
}
```

- There is one final question you should be asking at this point: What happens if the same exception matches multiple catch handlers at the end of a try block?
- If you catch *FormatException* and *Exception* in two different handlers, which one will run? (Or will both execute?)
- When an exception occurs, the runtime uses the first handler it finds that matches the exception and the others are ignored.
- This means that if you place a handler for Exception before a handler for *FormatException*, the *FormatException* handler will never run.

- Therefore, you should place more specific catch handlers above a general catch handler after a try block.
- If none of the specific catch handlers matches the exception, the general catch handler will.

Exception filters

- Exception filters are a new feature of C# that affect the way in which exceptions are matched against catch handlers.
- An exception filter enables you to specify additional conditions under which the catch handler is used.
- These conditions take the form of a Boolean expression prefixed by the when keyword.
- The following example illustrates the syntax:

```
catch (Exception ex) when (ex.GetType( !=typeof(System.OutOfMemoryException))
        {
            // Handle all previously uncaught
            exceptions except OutOfMemoryException
        }
```

- This example catches all exceptions (the Exception type), but the filter specifies that if the type of the exception is an out-of-memory exception, then this handler should be ignored.
- (The GetType method returns the type of the variable specified as the argument.)
- This provides a neat way to handle all exceptions except out-of- memory exceptions.
- If an out-of-memory exception occurs, the runtime will continue looking for another exception handler to use, and if it fails to find one, then the exception will be treated as an unhandled exception.

Using checked and unchecked integer arithmetic

- Chapter 2 discusses how to use binary arithmetic operators such as + and * on primitive data types such as *int* and *double*.
- It also instructs that the primitive data types have a fixed size.
  For example, a C# *int* is 32 bits.
- Because *int* has a fixed size, you know exactly the range of values that it can hold: it is -2147483648 to 2147483647.
- The fixed size of the *int* type creates a problem.

- For example, what happens if you add 1 to an *int* whose value is currently 2147483647?
- The answer is that it depends on how the application is compiled.
- By default, the C# compiler generates code that allows the calculation to overflow silently and you get the wrong answer.
- (In fact, the calculation wraps around to the largest negative integer value, and the result generated is - 2147483648.)
- The reason for this behavior is performance: integer arithmetic is a common operation in almost every program, and adding the overhead of overflow checking to each integer expression could lead to very poor performance.
- In many cases, the risk is acceptable because you know (or hope!) that your int values won’t reach their limits.
- If you don’t like this approach, you can turn on overflow checking.
- Regardless of how you compile an application, you can use the checked and unchecked keywords to turn on and off integer arithmetic overflow checking in parts of an application that you think need it.
- These keywords override the compiler option specified for the project.

Writing checked statements

- A checked statement is a block preceded by the ***checked*** keyword.
- All integer arithmetic in a checked statement always throws an *OverflowException* if an integer calculation in the block overflows, as shown in this example:

```
int number = int.MaxValue;
checked
{
    int willThrow = number++;
    Console.WriteLine("this won’t be reached");
}
```

- You can also use the unchecked keyword to create an ***unchecked*** block statement.
- All integer arithmetic in an unchecked block is not checked and never throws an *OverflowException*. For example:

```
int number = int.MaxValue;
unchecked
{
        int wontThrow = number++;
        Console.WriteLine("this will be reached");
}
```

Writing checked expressions

- You can also use the checked and unchecked keywords to control overflow checking on integer expressions by preceding just the individual parenthesized expression with the checked or unchecked keyword, as shown in this example:

```
int wontThrow = unchecked(int.MaxValue + 1);
int willThrow = checked(int.MaxValue + 1);
```

- The compound operators (such as += and -=) and the increment (++) and decrement (--) operators are arithmetic operators and can be controlled by using the checked and unchecked keywords.
- Remember, x += y is the same as x = x +y

IMPORTANT:

- You cannot use the checked and unchecked keywords to control floating-point (noninteger) arithmetic.
- The checked and unchecked keywords apply only to integer arithmetic using data types such as int and long.
- Floating-point arithmetic never throws OverflowException—not even when you divide by 0.0.

Throwing exceptions

- Suppose that you are implementing a method called *monthName* that accepts a single int argument and returns the name of the corresponding month.
- For example, *monthName(l)* returns “January”, *monthName(2)* returns “February”, and so on.
- The question is, what should the method return if the integer argument is less than 1 or greater than 12?

- The best answer is that the method shouldn't return anything at all—it should throw an exception.
- The .NET Framework class libraries contain lots of exception classes specifically designed for situations such as this.
- Most of the time, you will find that one of these classes describes your exceptional condition.
- (If not, you can easily create your own exception class, but you need to know a bit more about the C# language before you can do that.)
- In this case, the existing .NET Framework *ArgumentOutOfRangeException* class is just right.
- You can throw an exception by using the ***throw*** statement, as shown in the following example:

```
public static string monthName(int month)
{
switch (month)
    {
        case 1 :
        return "January";
        case 2 :
        return "February";
        case 12 :
        return "December";
        default :
        throw new
        ArgumentOutOfRangeException("Bad month");
    }
}
```

- The *throw* statement needs an exception object to throw.
- This object contains the details of the exception, including any error messages.
- This example uses an expression that creates a new *ArgumentOutOfRangeException* object.
- The object is initialized with a string that populates its Message property by using a constructor.

Using a finally block

- It is important to remember that when an exception is thrown, it changes the flow of execution through the program.
- This means that you can't guarantee that a statement will always run when the previous statement finishes because the previous statement might throw an exception.
- Remember that in this case, after the catch handler has run, the flow of control resumes at the next statement in the block holding this handler and not at the statement immediately following the code that raised the exception.
- Look at the example that follows, which is adapted from the code in Chapter 5, "Using compound assignment and iteration statements."
- It's very easy to assume that the call to *reader.Dispose* will always occur when the while loop completes.
- After all, it's right there in the code.

```
TextReader reader = ... ;
…
string line = reader.ReadLine( );
while (line != null)
{
    line = reader.ReadLine( );
}
    reader.Dispose( );
```

- Sometimes it's not an issue if one particular statement does not run, but on many occasions it can be a big problem.
- If the statement releases a resource that was acquired in a previous statement, failing to execute this statement results in the resource being retained.
- This example is just such a case: when you open a file for reading, this operation acquires a resource (a file handle), and you must ensure that you call reader.Dispose to release the resource.
- If you don't, sooner or later you'll run out of file handles and be unable to open more files.
- If you find that file handles are too trivial, think of database connections instead.
- The way to ensure that a statement is always run, whether or not an exception has been thrown, is to write that statement inside a finally block.

- A finally block occurs immediately after a try block or immediately after the last catch handler after a try block.
- As long as the program enters the fry block associated with a finally block, the finally block will always be run, even if an exception occurs.
- If an exception is thrown and caught locally, the exception handler executes first, followed by the finally block.
- If the exception is not caught locally (that is, the runtime has to search through the list of calling methods to find a handler), the finally block runs first.
- In any case, the finally block always executes.
- The solution to the reader.Dispose problem is as follows:

```
TextReader reader = . . . ;
…
try
{
        string line = reader.ReadLine( );
        while (line != null)
        {
                line = reader.ReadLine( );
        }
}
finally
{
        if (reader != null)
        {
                reader.Dispose( );
        }
}
```

- Even if an exception occurs while reading the file, the finally block ensures that the reader.Dispose statement always executes.