
Module 2: Understanding C# Language Fundamentals

Contents

Overview	1
Lesson: Understanding the Fundamentals of a C# Program	2
Lesson: Using C# Predefined Types	7
Lesson: Writing Expressions	23
Lesson: Creating Conditional Statements	33
Lesson: Creating Iteration Statements	42
Review	50
Lab 2.1: Writing a Savings Account Calculator	52



Information in this document, including URL and other Internet Web site references, is subject to change without notice. Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2002 Microsoft Corporation. All rights reserved.

Microsoft, MS-DOS, Windows, Windows NT, ActiveX, BizTalk, FrontPage, IntelliSense, JScript, Microsoft Press, MSDN, PowerPoint, Visual Basic, Visual C++, Visual C#, Visual Studio, Win32, Windows Media are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Overview

- Understanding the Fundamentals of a C# Program
- Using C# Predefined Types
- Writing Expressions
- Creating Conditional Statements
- Creating Iteration Statements

Introduction

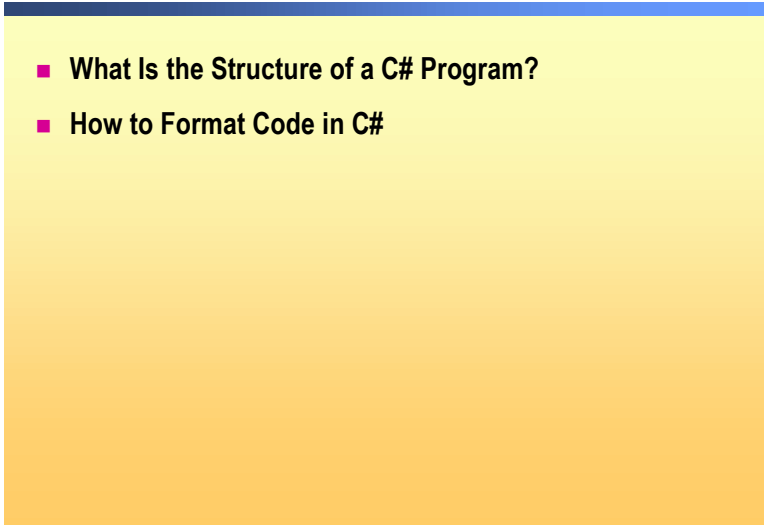
This module introduces you to the basic syntax and structure of the C# language. It describes C# data types, including variables and constants, describes the Microsoft® .NET common type system, introduces conditional and iterative statements, and explains how to create user-defined enumeration types. Understanding the syntax of the language is fundamental to writing code in C#.

Objectives

After completing this module, you will be able to:

- Understand the fundamentals of a C# program.
- Use C# predefined types.
- Write expressions.
- Create conditional statements.
- Create iteration statements.

Lesson: Understanding the Fundamentals of a C# Program

- 
- What Is the Structure of a C# Program?
 - How to Format Code in C#

Introduction

This lesson describes the structure of a C# program. This information is provided as a resource for developers who have no experience with a C-style language.

Lesson objectives

After completing this lesson, you will be able to:

- Identify C# statements.
- Use braces to group statements.
- Include comments in code.

Lesson agenda

This lesson includes the following topics:

- What Is the Structure of a C# Program?
- How to Format Code in C#

What Is the Structure of a C# Program?

- Program execution begins at Main()
- The **using** keyword refers to resources in the .NET Framework class library
- Statements are commands that perform actions
 - A program is made up of many separate statements
 - Statements are separated by a semicolon
 - Braces are used to group statements

```
using System;
class HelloWorld {
    static void Main() {
        Console.WriteLine ("Hello, World");
    }
}
```

Introduction

Before you write your first lines of code in C#, it is helpful to understand the structure of the language.

Definition

The structure of a programming language specifies the elements that you must include in your application and defines how to organize those elements so that the compiler understands your code.

Example of C# structure

The following code shows the basic structure of a C# application:

```
using System;

class HelloWorld {
    static void Main() {
        Console.WriteLine ("Hello, World");
    }
}
```

The elements and organizing principles that are shown in the preceding six lines of code are briefly described line by line in the following sections.

The using keyword

The **using** keyword refers to resources in the Microsoft .NET Framework class library. Typically, you insert this keyword at the beginning of the program file, usually several times, to reference various resources.

The System namespace

System is a *namespace* that provides access to all of the system functionality upon which your application is built.

Class

Programming in C#, or any object-oriented language, consists of writing classes, which are used to create objects. In the preceding code example, the class is named **HelloWorld**.

The Main method

Methods describe the behavior of a class. In the third line, **static void Main** is a global method that tells the compiler where to begin execution of the application. Every C# application must include a **Main** method in one of the classes.

Statements

Statements are instructions that are completed to perform actions in C# applications. Statements are separated by a semicolon to enable the compiler to distinguish between them.

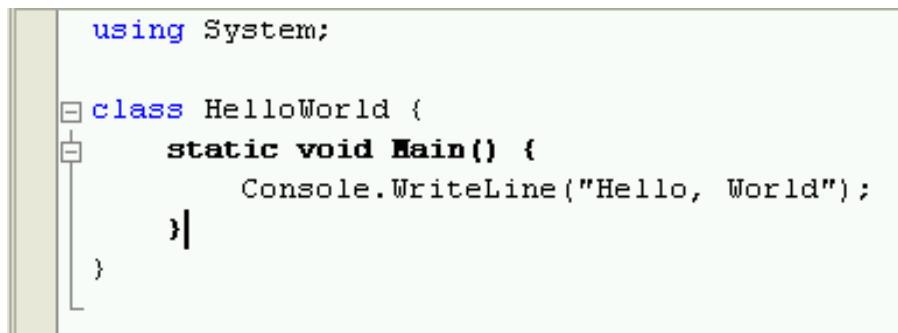
Some languages place one statement on one line. In C#, you can include multiple statements on one line, or one statement on multiple lines. It is good practice to write one statement per line; although, for the purpose of readability, you may want to break a long statement into several lines.

Braces

Braces, { and }, are used to identify the beginning and end of blocks of code in your application. Braces are used to group statements together. Every opening brace must have one matching closing brace.

In the example, the braces following “class HelloWorld” enclose the items that are in the **HelloWorld** class. The braces following “Main” are used to enclose the statements that are in the **Main** method.

Microsoft Visual Studio® .NET provides several visual cues that help to ensure that your braces are correctly matched. When you type a closing brace, the enclosing element is briefly shown in bold. Also, the document outline indicators to the left show the extent of a group of statements.



```
using System;

class HelloWorld {
    static void Main() {
        Console.WriteLine("Hello, World");
    }
}
```

The screenshot shows a code editor with the following C# code. The opening brace of the `class HelloWorld` block is highlighted with a blue background. The opening brace of the `static void Main()` block is also highlighted with a blue background. The closing brace of the `Main` method is highlighted with a blue background, and the `static void Main()` block is briefly shown in bold. To the left of the code, there are document outline indicators: a small square for the `class HelloWorld` block and a small square for the `static void Main()` block, with a vertical line connecting them.

Note You do not need to add a semicolon after braces because the braces themselves indicate the end of a group of statements, implying that the statements within the braces are complete and separate blocks of code.

How to Format Code in C#

- Use indentation to indicate enclosing statements
- C# is case sensitive
- White space is ignored
- Indicate single line comments by using //
- Indicate multiple-line comments by using /* and */

```
using System;
class HelloWorld {
    static void Main() {
        Console.WriteLine ("Hello, World");
    }
}
```

Introduction

Formatting is another element of program design that helps you to organize your code. You are encouraged to use formatting conventions to improve the structure and readability of your code.

Example

The following code sample demonstrates how to apply the formatting principles of indentation, case sensitivity, white space, and comments:

```
using System;

class HelloWorld {
    static void Main() {
        Console.WriteLine ("Hello, World");
        //writes Hello, World
    }
}
```

Indentation

Indentation indicates that a statement is within an enclosing statement. Statements that are in the same block of statements should all be indented to the same level. This is an important convention that improves the readability of your code. Although indenting is not a requirement, or enforced by the compiler, it is a recommended best practice.

Case sensitivity

C# is case sensitive, which means that the compiler distinguishes between uppercase and lowercase characters. For example, the words “code,” “Code,” and “CODE” are differentiated in your application; you cannot substitute one for the other.

White space

White space is ignored by the compiler. Therefore, you can use spaces to improve the readability and formatting of your code. The only exception is that the compiler does not ignore spaces between quotation marks.

Comments

You can include single-line comments in your application by inserting a double slash (//) followed by your comment.

Alternately, if your comment is lengthy and spans multiple lines, you can use slash asterisk (/*) to indicate the beginning of a comment and asterisk slash (*/) to indicate the end of your comments. The following example of a multiple line comment includes an asterisk at the beginning of each line. These asterisks are optional and you can include them to make your comment easier to identify.

Multiple-line comment example

```
/*
 * Multiple line comment
 * This example code shows how to format
 * multiple line comments in C#
 */

/* alternative use of this comment style */
```

Layout

You can place the opening brace at the end of the line that starts a statement group, or you can place the opening brace on the line following the method or class, as shown in the following example:

```
using System;

class HelloWorld
{
    static void Main()
    {
        Console.WriteLine("Hello, World");
    }
}
```

Both layouts are acceptable and correct. It is important, however, to be consistent. In the examples in Course 2609, *Introduction to C# Programming with Microsoft .NET*, the opening brace is placed at the end of the line. Your organization should choose one layout that everyone uses.

Lesson: Using C# Predefined Types

- What Are Predefined Types?
- How to Declare and Initialize Variables
- How to Declare and Initialize Strings
- How to Create and Use Constants
- How to Create and Use Enumeration Types
- How to Convert Between Types

Introduction

This lesson introduces the basic syntax of the C# language and the .NET common type system, including how to use types, variables, constants, enumerations, and strings.

When you write any application, you must represent data in some way. This process fundamentally depends upon working with types.

Lesson objectives

After completing this lesson, you will be able to:

- Declare and initialize variables.
- Create and use strings.
- Create and use constants.
- Create and use enumerated types.
- Convert between types.

Lesson agenda

This lesson includes the following topics and activity:

- What Are Predefined Types?
- How to Declare and Initialize Variables
- How to Declare and Initialize Strings
- How to Create and Use Constants
- How to Create and Use Enumeration Types
- How to Convert Between Types
- Practice: Using C# Types

What Are Predefined Types?

- **Types are used to declare variables**
- **Variables store different kinds of data**
 - Let the data that you are representing determine your choice of variable
- **Predefined types are those provided by C# and the .NET Framework**
 - You can also define your own
- **Variables must be declared before you can use them**

Introduction

Whenever your application must store data temporarily for use during execution, you store that data in a *variable*. You can think of variables as storage boxes. These boxes come in different sizes and shapes, called *types*, which provide storage for various kinds of data. For example, the type of variable that is used to store a number is different than one that is used to store a person's name.

Definition

Predefined types are those that are supplied by the C# language and the .NET Framework. The following table lists the predefined types and describes the data that they are designed to store.

Predefined type	Definition	# Bytes
byte	Integer between 0 and 255	1
sbyte	Integer between -128 and 127	1
short	Integer between -32768 and 32767	2
ushort	Integer between 0 and 65535	2
int	Integer between -2147483648 and 2147483647	4
uint	Integer between 0 and 4294967295	4
long	Integer between -9223372036854775808 and 9223372036854775807	8
ulong	Integer between 0 and 18446744073709551615	8
bool	Boolean value: true or false	1
float	Single-precision floating point value (non-whole number)	4
double	Double-precision floating point value	8
decimal	Precise decimal value to 28 significant digits	12
object	Base type of all other types	N/A
char	Single Unicode character between 0 and 65535	2
string	An unlimited sequence of Unicode characters	N/A

Storing data

Suppose that you are writing an application that allows a user to purchase items over the Internet with a credit card. Your application must handle several pieces of information: the person's name, the amount of the purchase, the credit card number, and the expiration date on the card. To represent this information in your application, you use different types.

Choosing a type

Let the data that you are representing determine your choice of type. For example, if something can be only true or false, a **bool** type is the obvious choice. A **decimal** type is a good choice for currency. When working with integers, an **int** type is the typical choice, unless there is a specific reason to choose another type.

In addition to the predefined types that are supplied by the .NET Framework, you can define your own types to hold whatever data you choose.

How to Declare and Initialize Variables

The diagram is divided into three sections, each with a numbered list of steps and a corresponding code example in a box.

- Declaring**
 - 1 Assign a type
 - 2 Assign a name
 - 3 End with a semicolon

```
int numberOfVisitors;
```

```
string bear;
```
- Initializing**
 - 1 Use assignment operator
 - 2 Assign a value
 - 3 End with a semicolon

```
string bear = "Grizzly";
```
- Assigning literal values**
 - 1 Add a type suffix

```
decimal deposit = 100M;
```

Introduction

A *variable* is a storage location for a particular type. For example, if your application must process a currency value, it requires a variable to hold that value.

Before you can use a variable, you must declare it. By declaring a variable, you are actually reserving some storage space for that variable in memory. After declaring a variable, you must initialize it by assigning a value to it.

Syntax

The syntax for declaring a variable is the type declaration followed by the variable name. For example:

```
int    myInteger;  
bool   fileWasClosed;
```

Naming variables

The following list identifies some best practices for naming your variables:

- Assign meaningful names to your variables.
- Use camel case. In camel case, the first letter of the identifier is lowercase, and the first letter of each subsequent word in the identifier is capitalized, such as `newAccountBalance`.
- Do not use C# keywords.
- Although C# is case sensitive, do not create variables that differ only by case.

Initializing variables

To initialize a variable, you assign it a value. To assign a value to a variable, use the assignment operator (=), followed by a value, and then a semicolon, as shown in the following example:

```
int myVariable;  
myVariable = 1;
```

You can combine these steps, as shown in the following example:

```
int myVariable = 1;
```

More examples of declaring variables are shown in the following code:

```
int x = 25;  
int y = 50;  
bool isOpen = false;  
sbyte b = -55;
```

Assigning literal values

When you assign 25 to x in the preceding code, the compiler places the literal value 25 in the variable x. The following assignment, however, generates a compilation error:

```
decimal bankBalance = 3433.20; // ERROR!
```

This code causes an error because the C# compiler assumes that any literal number with a decimal point is a double, unless otherwise specified. You specify the type of the literal by appending a suffix, as shown in the following example:

```
decimal bankBalance = 3433.20M;
```

The literal suffixes that you can use are shown in the following table. Lowercase is permitted.

Category	Suffix	Description
Integer	U	Unsigned
	L	Long
	UL	Unsigned long
Real number	F	Float
	D	Double
	M	Decimal
	L	Long

Characters

You specify a character (char type) by enclosing it in single quotation marks:

```
char myInitial = 'a';
```

Escape characters

Some characters cannot be specified by being placed in quotation marks—for example, a newline character, a beep, or a quotation mark character. To represent these characters, you must use **escape** characters, which are shown in the following table.

Escape sequence	Character name
\'	Single quotation mark
\"	Double quotation mark
\\	Backslash
\0	Null
\a	Alert
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

For example, you can specify a quotation mark as follows:

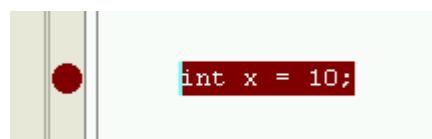
```
char quoteMark = '\'';
```

Examining variables in Visual Studio .NET

The Visual Studio .NET development environment provides useful tools that enable you to examine the values of variables while your application is running.

To examine the value of a variable, set a breakpoint at the variable that you want to examine, run your application in debug mode, and then use the debug windows to examine the values.

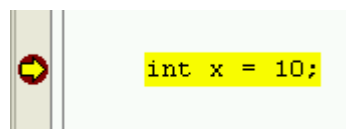
1. Set a breakpoint by clicking in the left margin in the source window. The breakpoint is indicated by a red dot. You can also set breakpoints by clicking **New Breakpoint** on the **Debug** menu, or by pressing SHIFT+B.



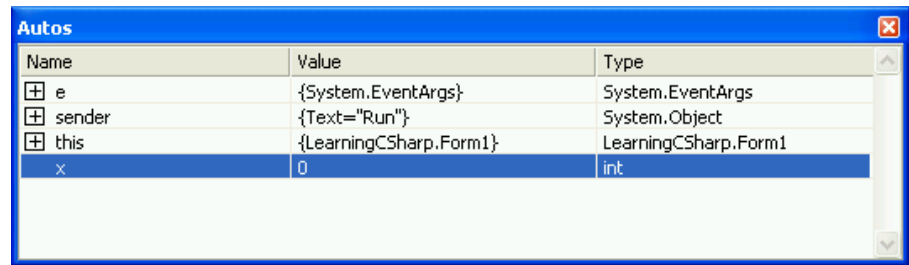
2. Run your application in debug mode by clicking **Start** on the standard toolbar, or by clicking **Start** on the **Debug** menu, or by pressing F5.



3. Your application runs until it encounters a breakpoint. When the application encounters a breakpoint, it pauses, and the development environment highlights the line of code that will be executed next.



4. Use the debug windows to view the value of the variables. To open debug windows, on the **Debug** menu, point to **Windows**, and then click **Autos**, or click **Locals**, or click **This**.



The **Autos** window displays variables used in the current and previous statements. By default, the **Autos** window is visible at the bottom of the development environment when you are in debug mode.

The **Locals** window displays local variables, and the **This** window shows objects that are associated with the current method.

5. To proceed to the next breakpoint when you are ready to continue executing your program, you can press F5 or click **Continue** in the **Debug** menu. Or, you can execute one program step at a time by pressing F10 or clicking **Step Over** on the **Debug** menu.

How to Declare and Initialize Strings

■ Example string

```
string s = "Hello World"; // Hello World
```

■ Declaring literal strings

```
string s = "\"Hello\""; // "Hello"
```

■ Using escape characters

```
string s = "Hello\nWorld"; // a new line is added
```

■ Using verbatim strings

```
string s = @"Hello\n"; // Hello\n
```

■ Understanding Unicode

```
The character "A" is represented by "U+0041"
```

Introduction

Strings are one of the most commonly used types.

Definition

A *string* variable contains a sequence of alphanumeric characters that are used as input for calculations or searches.

Note There is no limit to the number of characters that can make up a string.

Syntax

You declare a string the same way you declare any other variable, by assigning a type (string) and giving it a name.

Declaring literal strings

You can assign a literal value to the string variable by enclosing the value in quotation marks.

```
string sample = "Hello World";
```

Using escape characters

You can also include escape characters in a string.

For example, if you want to create a string that is written on two lines, you can insert a line break within your string by using the `\n` escape character, as shown in the following example:

```
string sample = "Hello\nWorld";
```

This code produces the following output:

```
Hello
World
```


If you want to insert a tab, use the `\t` escape character, as shown in the following example:

```
string sample = "Hello\tWorld"; // produces Hello    World
```

To insert a backslash, which is useful for including file path locations, use the `\\` escape character, as shown in the following example:

```
string sample = "c:\\My Documents\\sample.txt";  
// produces c:\My Documents\sample.txt
```

Using verbatim strings

A *verbatim* string is a string that is interpreted by the compiler exactly as it is written, which means that even if the string spans multiple lines or includes escape characters, these are not interpreted by the compiler and they are included with the output. The only exception is the quotation mark character, which must be escaped so that the compiler can recognize where the string ends.

A verbatim string is indicated with an at sign (`@`) character followed by the string enclosed in quotation marks. For example:

```
string sample = @"Hello";  
string sample = @"Hello\tWorld"; // produces "Hello\tWorld"
```

The following code shows a more useful example:

```
string sample = @"c:\My Documents\sample.txt";  
// produces c:\My Documents\sample.txt
```

If you want to use a quotation mark inside a verbatim string, you must escape it by using another set of quotation marks. For example, to produce "Hi" you use the following code:

```
string s = @"""Hi"""; // Note: three quotes on either side
```

The preceding code produces the following string:

```
"Hi"
```

Understanding Unicode

The .NET Framework uses Unicode UTF-16 (Unicode Transformation Format, 16-bit encoding form) to represent characters. C# also encodes characters by using the international Unicode Standard. The Unicode Standard is the current universal character encoding mechanism that is used to represent text in computer processing. The previous standard was ASCII.

The Unicode Standard represents a significant improvement over ASCII because Unicode assigns a unique numeric value, called a code point, and a name to each character that is used in all the written languages of the world. ASCII defined only 128 characters, which meant that some languages could not be correctly displayed in a computer application.

For example, the character “A” is represented by the code point “U+0041” and the name “LATIN CAPITAL LETTER A”. Values are available for over 65,000 characters, and there is room to support up to one million more. For more information, see The Unicode Standard at www.unicode.org.

How to Create and Use Constants

- Declared using the **const** keyword and a type
- You must assign a value at the time of declaration

```
const int earthRadius = 6378;//km  
const long meanDistanceToSun = 149600000;//km  
const double meanOrbitalVelocity = 29.79D;//km sec
```

Introduction	A constant is a variable whose value remains constant. Constants are useful in situations where the value that you are using has meaning and is a fixed number, such as pi, the radius of the earth, or a tax rate.
Benefits	Constants make your code more readable, maintainable, and robust. For example, if you assign a value of 6378 to a constant named earthRadius , when you use this value in calculations it is immediately apparent what value you are referring to, and it is not possible for someone to assign a different value to earthRadius .
Syntax	You declare a constant by using the const keyword and a type. You must assign a value to your constants at the time that you declare them.
Examples	<pre>const int earthRadius = 6378; // km const long meanDistanceToSun = 149600000; // km const double meanOrbitalVelocity = 29.79D; // km/sec</pre>

How to Create and Use Enumeration Types

■ Defining Enumeration Types

```
enum Planet {  
    Mercury,  
    Venus,  
    Earth,  
    Mars  
}
```

■ Using Enumeration Types

```
Planet aPlanet = Planet.Mars;
```

■ Displaying the Variables

```
Console.WriteLine("{0}", aPlanet); //Displays Mars
```

Introduction

An *enumeration type* specifies a group of named numeric constants. An enumeration type is a *user-defined type*, which means that you can create an enumeration type, declare variables of that type, and assign values to those variables. The purpose of an enumeration type is to represent constant values.

Benefits

In addition to providing all the advantages of constants, enumerations:

- Make your code easier to maintain by ensuring that your variables are assigned only anticipated values.
- Allow you to assign easily identifiable names to the values, thereby making your code easier to read.
- Make your code easier to type, because as you assign enumeration values, Microsoft IntelliSense® displays a list of the possible values that you can use.
- Allow you to specify a set of constant values and define a type that will accept values from only that set.

Syntax

You create an enumeration type by using the **enum** keyword, assigning a name, and then listing the values that your enumeration can take.

It is recommended that you use Pascal case for the type name and each enumeration member. In Pascal case, you capitalize the initial letter of each word in the identifier, such as **ListOfThePlanets**.

Example

An enumeration type is shown in the following example:

```
enum Planet {  
    Mercury,  
    Venus,  
    Earth,  
    Mars  
}
```

The preceding code creates a new type, **Planet**. You can declare variables of this type and assign them values from the enumeration list.

Referring to a specific member

When you want to refer to a specific member in an enumeration, you use the enumeration name, a dot, and the member name.

For example, the following code declares a variable **innerPlanet** of type **Planet**, and assigns it a value:

```
Planet innerPlanet = Planet.Venus;
```

You can declare an enumeration in a class or a namespace but not in a method.

Assigning values to enumeration members

If the members of your enumeration must have a specific value, you can assign that value when you declare the enumeration. The following code assigns a value based on the equatorial radius of the inner planets:

```
enum Planets {  
    Mercury = 2437,  
    Venus = 6095,  
    Earth = 6378  
}
```

Enumeration base types

You can use any integer except `char` as the base type that is used for the enumeration by specifying the type after the name of the enumeration type. For example:

```
enum Planets : uint {  
    Mercury = 2437,  
    Venus = 6095,  
    Earth = 6378  
}
```

How to Convert Between Types

■ Implicit

- Performed by the compiler on operations that are guaranteed not to truncate information

```
int x = 123456; // int is a 4-byte integer
long y = x; // implicit conversion to a long
```

■ Explicit

- Where you explicitly ask the compiler to perform a conversion that otherwise could lose information

```
int x = 65537;
short z = (short) x;
// explicit conversion to a short, z == 1
```

Introduction

When designing applications, you often must convert data from one type to another. Conversion can be necessary when you perform operations on two types that are not the same.

Definitions

There are two types of conversions in the .NET Framework: *implicit* and *explicit* conversions.

- An *implicit* conversion is a conversion that is automatically performed by the common language runtime on operations that are guaranteed to succeed without truncating information.
- An *explicit* conversion is a conversion that requires you to explicitly ask the compiler to perform a conversion that otherwise could lose information or produce an error.

Why convert?

For example, when a currency value is entered on a Web page, the type of the data may actually be text. A programmer must then convert that text to a numeric value.

Another reason for conversion is to avoid number overflow. If you try to add two bytes, the compiler returns an **int**. It returns an **int** because a byte can hold only eight bits, up to a value of 255, so the result of adding two bytes could easily result in a number greater than 255. Therefore, the resulting value is converted by the compiler and returned as an **int**.

Implicit conversions

The following table shows the implicit type conversions that are supported in C#:

From	To
sbyte	short, int, long, float, double, decimal
byte	short, ushort, int, uint, long, ulong, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long, ulong	float, double, decimal
float	double
char	ushort, int, uint, long, ulong, float, double, decimal

Notice that implicit conversions can be performed only from a smaller type to a larger type or from an unsigned integer to a signed integer.

Example

The following example shows an implicit conversion:

```
int x = 123456; // int is a 4-byte integer
long y = x; // implicit conversion to a long
```

Explicit conversions

The syntax for performing an explicit conversion is shown in the following code:

```
type variable1 = (cast-type) variable2;
```

The type in the parentheses indicates to the compiler that the value on the right side is to be converted to the type specified in the parentheses.

Example

The following example shows explicit type conversion:

```
int x = 500;
short z = (short) x;
// explicit conversion to a short, z contains the value 500
```

It is important to remember that explicit conversions can result in data loss. For example, in the following code, a decimal is explicitly converted to an int:

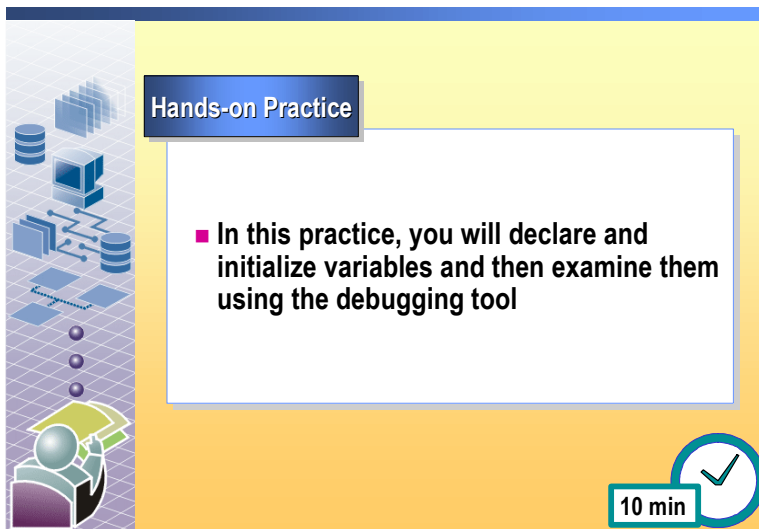
```
decimal d = 1234.56M;
int x = (int) d;
```

The result of this conversion is that x is assigned a value of **1234**.

Other conversions

The .NET Framework class library also provides support for type conversions in the **System.Convert** class.

Practice: Using C# Types



In this practice, you will declare and initialize several variables and then examine them with the debugging tool.

The starter code contains descriptions of several tasks for you to perform. Under each task is a line similar to the following:

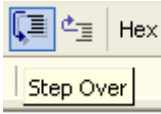
```
Output( null );
```

When you replace the word **null** with the variable that you declared, the value of the variable appears in the form. For example, if you are asked to declare an integer and assign it the value **42**, write the following code:

```
int x = 42;
Output( x );
```

The solution code for this practice is located in *install_folder*\Practices\Mod02\Types_Solution\Types.sln. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Start Visual Studio .NET, and then open <i>install_folder</i> \Practices\Mod02\Types\Types.sln.	a. Start a new instance of Visual Studio .NET. b. On the Start Page, click Open Project . c. In the Open Project dialog box, browse to <i>install_folder</i> \Practices\Mod02\Types, click Types.sln , and then click Open . d. In Solution Explorer, click Form1.cs , and then press F7 to open the Code Editor.
2. Review the tasks for this practice.	a. On the View menu, point to Show Tasks , and then click All . b. Review the tasks that are listed in the Task List window.

Tasks	Detailed steps
3. Declare, initialize, and display a variable with the value Suzan Fine .	<ol style="list-style-type: none"> In the Task List, double-click TODO: 1 Initialize the suzanName variable with the value “Suzan Fine”. Assign the value Suzan Fine to the variable suzanName. Change the word null on the following line to suzanName.
4. Declare, initialize, and display a variable with the value 135.20 .	<ol style="list-style-type: none"> In the Task List, double-click TODO: 2 Declare and initialize a variable to hold a currency amount (135.20). Declare a variable, and assign it the value 135.20. Remember that the value 135.20 is assumed to be a double, unless you append a suffix that indicates otherwise. Change the word null on the following line to the name of your variable.
5. Declare a Planet variable, assign the value Planet.Earth to it, and display its value.	<ol style="list-style-type: none"> In the Task List, double-click TODO 3: Using the Planet enumeration, assign Planet.Earth to ourPlanet. At the top of the source code file, locate the Planet enumeration. Declare a variable of the enumeration type: Planet ourPlanet; Assign the value Planet.Earth to the variable. Change the word null on the following line to the name of your variable.
6. Use the debugging tool to step through your code, examining the values of the variables by using the Locals window. 	<ol style="list-style-type: none"> Locate the line int x = 42; and set a breakpoint at that line. Press F5 to compile and run your application. In Visual Studio .NET, on the Debug menu, point to Windows, and then click Locals. In your application window, click Run. Step through your code, one line at a time, by clicking the Step Over button shown on the left, or by pressing F10. Examine the Locals and Autos windows to check that your program is assigning values correctly.
7. Save your application, and then quit Visual Studio .NET.	<ol style="list-style-type: none"> Save your application. Quit Visual Studio .NET.

Optional: The solution code declares additional variables.

- Use the debugging tool to examine the value of the variables.
- Explain why **myShort** has the value of **1** after the assignment.

Lesson: Writing Expressions

- What Are Expressions and Operators?
- How to Determine Operator Precedence

Introduction

This lesson explains how to use operators to create expressions.

Lesson objective

After completing this lesson, you will be able to use operators to create expressions.

Lesson agenda

This lesson includes the following topics and activity:

- What Are Expressions and Operators?
- How to Determine Operator Precedence
- Practice: Using Operators

What Are Expressions and Operators?

■ Operators Are Symbols Used in Expressions

Common Operators	Example
• Increment / decrement	++ --
• Arithmetic	* / % + -
• Relational	< > <= >=
• Equality	== !=
• Conditional	&& ?:
• Assignment	= *= /= %= += -= <=> >>= &= ^= =

Introduction

The purpose of writing an expression is to perform an action and return a value. For example, you can write an expression to perform a mathematical calculation, assign a value, or compare two values.

Definitions

An *expression* is a sequence of operators and operands. An *operator* is a concise symbol that indicates the action that you want to occur in your expression. An *operand* is the value on which an operation is performed. An operator is specifically designed to produce a new value from the value that is being operated on.

Types of operators

Some of the common types of operators that you can use in your C# applications include:

- *Increment and decrement.* Used to increase or decrease a value by one.
- *Arithmetic.* Used to perform arithmetic calculations like addition.
- *Relational.* Used to define greater than, greater than or equal to, less than, and so on.
- *Equality.* Used to state equal to, or not equal.
- *Conditional.* Used to define and/or situations.
- *Assignment.* Used to assign a value to a variable.

Most operators work only with numeric data, but equality and assignment operators can also work on strings of text.

The following table lists all the operators that can be used in a C# application:

Operator type	Operator
Primary	(x), x.y, f(x), a[x], x++, x--, new, typeof, sizeof, checked, unchecked
Unary	+, -, !, ~, ++x, --x, (T)x
Mathematical	+, -, *, /, %
Shift	<<, >>
Relational	<, >, <=, >=, is
Equality	==
Logical	&, , ^
Conditional	&&, , ?
Assignment	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, =

Note It is important to notice the difference between the assignment operator and the equality operator. Notice that “is equal to” is represented by two equal signs (==), because a single equal sign (=) is used to assign a value to a variable.

Example

```
int x = 10;    // assignment
int y = 20;
int z = x + y; // mathematical plus (z == 30)
```

Operator shortcuts

C# makes it possible for you to use concise syntax to manipulate data in complex ways. The following table lists the C# operator shortcuts.

Shortcut	Identical Expression
x++, ++x	x = x + 1 The first form increments x after the expression is evaluated; the second form increments x before the expression is evaluated.
x--, --x	x = x - 1
x += y	x = x + y
x -= y	x = x - y
x *= y	x = x * y
x /= y	x = x / y
x %= y	x = x % y
x >>= y	x = x >> y
x <<= y	x = x << y
x &= y	x = x & y
x = y	x = x y
x ^= y	x = x ^ y

Example

```
int x = 11;
int z = 20;
z += x;
```

After the preceding expressions are evaluated, z has the value **31**.

Increment and decrement

The increment and decrement operators can occur either before or after an operand. For example, `x++` and `++x` are both equivalent to `x=x+1`. However, when these operators occur in expressions, `x++` and `++x` behave differently.

`++x` increments the value of `x` *before* the expression is evaluated. In other words, `x` is incremented and then the new value of `x` is used in the expression.

Example 1

```
int x = 5;
(++x == 6) // true or false?
```

The answer is **true**.

`x++` increments the value of `x` after the expression is carried out; therefore, the expression is evaluated using the original value of `x`.

Example 2

```
x = 5
(x++ == 6) // true or false?
```

The answer is **false**.

Example 3

```
int x = 10
int y = x++; // y is equal to ten
int z = x + y; // z is equal to twenty-one
```

Tip To improve the readability of your code, place increment and decrement operators in separate statements.

Logical negation operator

An exclamation point (!) is the logical negation operator. It is used in an assignment to reverse the value of a Boolean.

If `bool b` is false, `!b` is true.

If `b` is true, `!b` is false.

For example:

```
bool isAwake = true;
bool isAsleep = !isAwake;
```

Mathematical operators

In addition to the obvious `+`, `-`, `*` and `/` operators, there is a remainder operator (`%`) that returns the remainder of a division operation. For example:

```
int x = 20 % 7;    // x == 6
```

Logic operators

C# provides logic operators, as shown in the following table.

Logic operator type	Operator	Description
Conditional	&&	x && y returns true if x is true AND y is true; y is evaluated only if x is true
		x y returns true if x is true OR y is true; y is evaluated only if x is false
Boolean	&	x & y returns true if x AND y are both true
		x y returns true if either x OR y is true
	^	x ^ y returns true if x OR y is true, but false if they are both true or both false

Developers often use conditional logic operators. These operators follow the same rules as Boolean logic operators but have the useful characteristic that the expressions are evaluated only if they need to be evaluated.

Using operators with strings

You can also apply the plus and the equality operators to string types. The plus concatenates strings whereas the string equality operator compares strings.

```
string a = "semi";  
string b = "circle";  
string c = a + b;  
string d = "square";
```

The string c has the value **semicircle**.

```
bool sameShape = ( "circle" == "square" );
```

```
sameShape = ( b == d );
```

The Boolean **sameShape** is **false** in both statements.

How to Determine Operator Precedence

- Expressions are evaluated according to operator precedence

```
10 + 20 / 5      result is 14
```

- Parentheses can be used to control the order of evaluation

```
(10 + 20) / 5      result is 6
10 + (20 / 5)      result is 14
```

- Operator precedence is also determined by associativity
 - Binary operators are left-associative
 - Assignment and conditional operators are right-associative

Introduction

Developers often create expressions that perform more than one calculation, comparison, or a combination of the two. In these situations, the *precedence* of the operators controls the order in which the expression is evaluated. If you want the operations performed in a different order; you must tell the compiler to evaluate the expression differently by using parentheses.

Evaluation order

The order in which operators are evaluated in an expression is shown in the following precedence table.

Operator type	Operator
Primary	x.y, f(x), a[x], x++, x--, new, typeof, checked, unchecked
Unary	+, -, !, ~, ++x, --x, (T)x
Multiplicative	*, /, %
Additive	+, -
Shift	<<, >>
Relational	<, >, <=, >=, is, as
Equality	==, !=
Logical	&, ^,
Conditional	&&, , ?:
Assignment	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, =

For example, the plus operator + has a lower precedence than the multiplication operator, so `a + b * c` means multiply `b` and `c`, and then add the sum to `a`.

Parentheses

Use parentheses to show the order of evaluation and to make the evaluation order of your expressions more readable. Extra parentheses are removed by the compiler and do not slow your application in any way, but they can make an expression much more readable.

For example, in the following expression, the compiler will multiply b by c and then add d.

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

Using parentheses, in the following expression, the compiler first evaluates what is in parentheses, (c + d), and then multiplies by b.

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

The following examples demonstrate operator precedence and the use of parentheses for controlling the order of evaluation in an expression:

```
10 + 20 / 5 (result is 14)
(10 + 20) / 5 (result is 6)
10 + ( 20 / 5 ) (result is 14)
((10 + 20) * 5) + 2 (result is 152)
```

Associativity

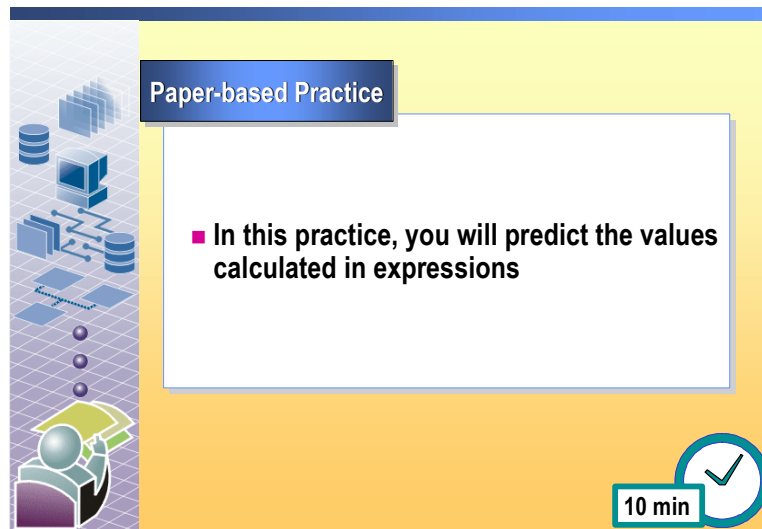
All binary operators, those that take two operands, are left-associative, meaning that the expression is evaluated from left to right, except for assignment operators. Assignment operators and conditional operators are right-associative.

For example:

x + y + z is evaluated as (x + y) + z



x = y = z is evaluated as x = (y = z)

Practice: Using Operators



In this paper-based practice, look at each line of code, and then answer the question. Assume that the code is executed in sequence, as written.

Tasks	Detailed steps
1. Read the code in the right column, and then answer the following question.	a. Read the following code: <pre>int x = 10; int y = x++;</pre> b. Answer the following question.
? What is the value of y ? Why? _____ _____	
2. Read the code in the right column, and then answer the following question.	a. Read the following code, which is continued from the preceding step: <pre>x += 10;</pre> b. Answer the following question.
? What is the value of x ? Why? _____ _____	

Tasks	Detailed steps
3. Read the code in the right column, and then answer the following question.	<p>a. Read the following code, which is continued from the preceding step:</p> <pre>int z = 30; int a = x + y * z;</pre> <p>b. Answer the following question.</p>
<p> What is the value of a? Why? Write this in a more readable form.</p> <hr/> <hr/>	
4. Read the code in the right column, and then answer the following question.	<p>a. Read the following code, which is continued from the preceding step:</p> <pre>int a = 10; int b = a++; bool myBool = (a == b);</pre> <p>b. Answer the following question.</p>
<p> What does this code do? What is the value of myBool?</p> <hr/> <hr/>	
5. Start Visual Studio .NET, and then open <i>install_folder</i> \Practices\Mod02\Operators\Operators.sln.	<p>a. Start a new instance of Visual Studio .NET.</p> <p>b. On the Start Page, click Open Project.</p> <p>c. In the Open Project dialog box, browse to <i>install_folder</i>\Practices\Mod02\Operators, click Operators.sln, and then click Open.</p> <p>d. In Solution Explorer, click Form1.cs, and then press F7 to open the Code Editor.</p>

Tasks	Detailed steps
6. Check your answers by stepping through the code.	<ul style="list-style-type: none">a. Locate the line int x = 10; and set a breakpoint at that line.b. Press F5 to compile and run the application.c. If the Locals window is not visible, in Visual Studio .NET, on the Debug menu, point to Windows, and then click Locals.d. In your application window, click Run.e. Step through your code, a line at a time, by clicking the Step Over button, or by pressing F10.f. Examine the Locals and Autos windows to check that your application assigns values correctly.
7. Quit Visual Studio .NET.	<ul style="list-style-type: none">▪ Quit Visual Studio .NET.

Lesson: Creating Conditional Statements

- How and When to Use the **if** Statement
- How and When to Use the **switch** Statement

Introduction

This lesson introduces you to conditional statements. You learn how and when to use **if** and **switch** statements.

Lesson objectives

After completing this lesson, you will be able to:

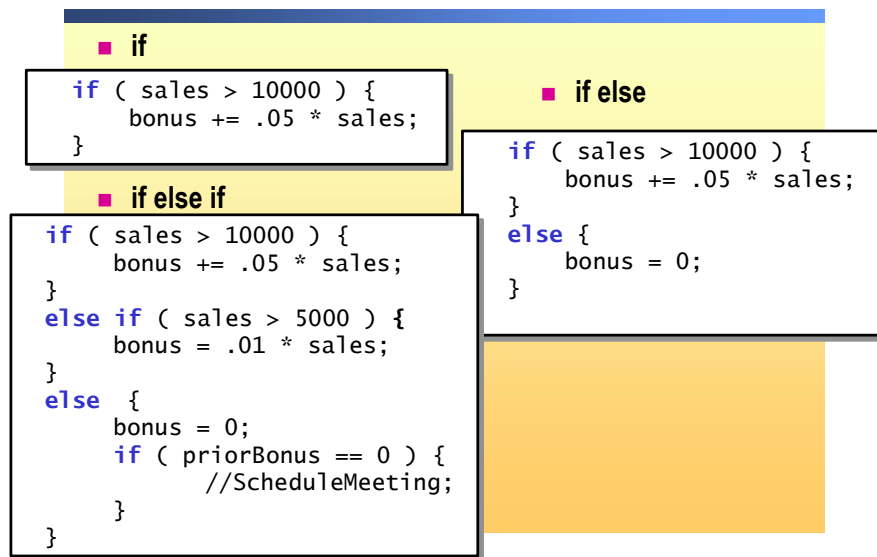
- Use the **if...else** conditional statement to manage the flow of control in an application.
- Use the **switch** conditional statement to manage the flow of control in an application.

Lesson agenda

This lesson includes the following topics and activity:

- How and When to Use the **if** Statement
- How and When to Use the **switch** Statement
- Practice: Using Conditional Statements

How and When to Use the *if* Statement



Introduction

A conditional statement allows you to control the flow of your application by selecting the statement that is executed, based on the value of a Boolean expression. There are three variations to the conditional **if** statement, including: **if**, **if else**, and **if else if**.

When the expression that is being evaluated is **true**, the code following the **if** statement is executed.

Declaring an if statement

The syntax of an **if** statement is as follows:

```
if (boolean-expression) statement
```

In the following example, if the value of **sales** is greater than **10000**, the bonus calculation statement is performed:

```
if ( sales > 10000 ) {  
    bonus += .05 * sales;  
}
```

Declaring an if else statement

The syntax for declaring an **if else** statement is as follows:

```
if ( boolean-expression ) statement1 else statement2
```

Statement1 is executed if the Boolean expression is **true**. Otherwise, statement2 is executed.

For example:

```
if ( sales > 10000 ) {  
    bonus += .05 * sales;  
}  
else {  
    bonus = 0;  
}
```

Declaring an if else if statement

You can nest **if** statements by writing them in the form of an **if else if** statement, as shown in the following example:

```
if ( sales > 10000 ) {  
    bonus += .05 * sales;  
}  
else if ( sales > 5000 ) {  
    bonus = .01 * sales;  
}  
else {  
    bonus = 0;  
    if ( priorBonus == 0 ) {  
        // Schedule a Meeting;  
    }  
}
```

Evaluating multiple expressions

You can evaluate more than one expression in an **if** statement. For example, the following **if** statement evaluates to **true** if the value of **sales** is greater than 10,000 but less than 50,000:

```
if ( (sales > 10000) && (sales < 50000) ) {  
    // sales are between 10001 and 49999 inclusive  
}
```

Using the ternary operator

The ternary operator (?) is a shorthand form of the **if...else** statement. It is useful when you want to perform a comparison and return a Boolean value.

For example, the following expression assigns the value **0** to **bonus** if the value of **sales** is less than 10000:

```
bonus = ( sales > 10000 ) ? ( sales * .05 ) : 0 ;
```

How and When to Use the *switch* Statement

```
int moons;
switch (aPlanet){
    case Planet.Mercury:
        moons = 0;
        break;
    case Planet.Venus:
        moons = 0;
        break;
    case Planet.Earth:
        moons = 1;
        break;
}
```

■ Default case

Introduction

A **switch** statement selects the code to execute based upon the value of a test. However, a **switch** statement enables you to test for multiple values of an expression rather than just one condition.

Switch statements are useful for selecting one branch of execution from a list of mutually-exclusive choices. Using **switch** statements makes your application more efficient and your code more readable than using multiple, nested **if** statements.

Syntax

A **switch** statement takes the form of a switch expression followed by a series of switch blocks, indicated by case labels. When the expression in the argument evaluates to one of the values in a particular case, the code immediately following that case executes. When no match occurs, a default condition is executed, if one is defined.

Break

You must include a break statement at the end of each switch block, or a compile error occurs. It is not possible to fall through from one switch block to the following switch block.

Example

In the following **switch** statement, assume that “x” is an integer:

```
switch ( x ) {
    case 0:
        // x is 0
        break;
    case 1:
        // x is 1
        break;
    case 2:
        // x is 2
        break;
}
```

Execution sequence

The execution sequence is as follows:

1. x is evaluated.
2. If one of the constant values in the case label is equal to the value of the switch expression, control is passed to the statement following that case label.
3. If none of the case labels match the value of the expression, control is passed to the end point of the case statement, or to the default case, which is described in the following section.

If x has the value **1**, the statements following the case 1 label are selected and executed.

Defining a default condition

Often, you want to define a default condition so that values that are not handled specifically can still be caught. The following example shows how to define a default condition:

```
switch ( x ) {  
    case 0:  
        // x is 0  
        break;  
    case 1:  
        // x is 1  
        break;  
    case 2:  
        // x is 2  
        break;  
    default:  
        // x is not 0, 1 or 2  
        break;  
}
```

The default label catches any values that are not matched by the case labels.

**Using enumerations
with switch statements**

The type that is evaluated in the expression must be an integer type, a character type, a string, an enumeration type; or a type that can be implicitly converted to one of these types. You will often use **switch** statements with enumeration types.

In the following example, the **switch** statement selects the case based on the value of the enumeration type. This **switch** statement makes the code very readable.

```
enum Animal {  
    Antelope,  
    Elephant,  
    Lion,  
    Osprey  
}  
.  
.  
.  
  
switch( favoriteAnimal ) {  
    case Animal.Antelope:  
        // herbivore-specific statements  
        break;  
    case Animal.Elephant:  
        // herbivore-specific statements  
        break;  
    case Animal.Lion:  
        // carnivore-specific statements  
        break;  
    case Animal.Osprey:  
        // carnivore-specific statements  
        break;  
}
```


Combining cases

You can use multiple case labels on a single switch expression as shown in the following example:

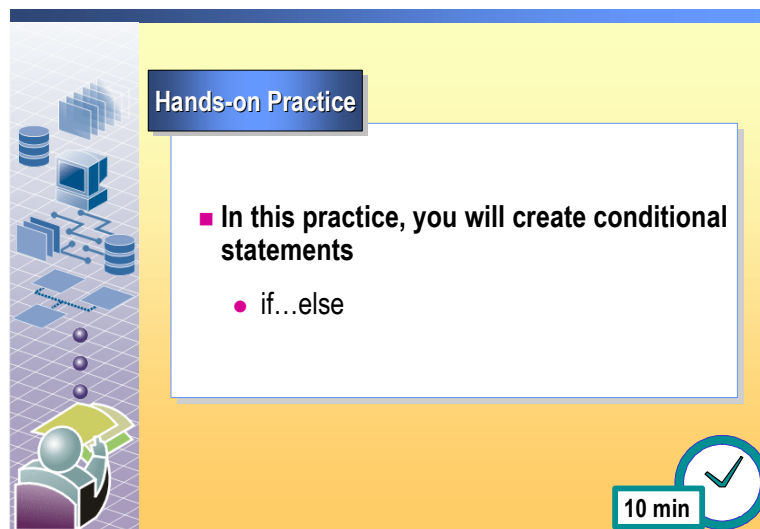
```
switch( favoriteAnimal ) {  
    case Animal.Antelope:  
    case Animal.Elephant:  
        // herbivore-specific statements  
        break;  
    case Animal.Lion:  
    case Animal.Osprey:  
        // carnivore-specific statements  
        break;  
}
```

In this case, if **favoriteAnimal** is either **Lion** or **Osprey**, the carnivore-specific statements are executed.

You cannot place a statement between the Antelope and the Elephant cases unless you also place a **break** statement between them.

Note If you are familiar with switch statements in C or C++, it is important to note that C# does not allow you to fall through a switch expression to the next switch expression. In C#, every case that has statements must also have a break statement.

Practice: Using Conditional Statements

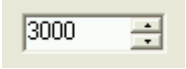


In this practice, you will complete code that lacks appropriate conditional logic.

Suppose that a zoo needs 5000 visitors per week to meet a budget projection. You will use an **if** statement to check the number of visitors and write a message indicating whether the number of visitors was above or below the goal of 5000 visitors.

The solution code for this practice is located in *install_folder*\Practices\Mod02\Conditions_Solution\conditions.sln. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Start Visual Studio .NET, and then open <i>install_folder</i> \Practices\Mod02\Conditions\Conditions.sln.	<ol style="list-style-type: none"> Start a new instance of Visual Studio .NET. On the Start Page, click Open Project. In the Open Project dialog box, browse to <i>install_folder</i>\Practices\Mod02\Conditions, click Conditions.sln, and then click Open. In Solution Explorer, click Form1.cs, and then press F7 to open the Code Editor.
2. Locate the task TODO 1: using if statements.	<ol style="list-style-type: none"> On the View menu, point to Show Tasks, and then click All. In the Task List, double-click TODO 1: using if statements.
3. If the value in visitors is 5000 or more, then use the Output method to display a message indicating that the target was achieved. Otherwise, display a message saying that the target was not achieved.	<ol style="list-style-type: none"> Use an if statement to test whether the value of visitors is 5000 or more. If visitors is 5000 or more, use the Output method to display a message that says that the target has been achieved. For example: <pre>Output("Visitor target achieved");</pre>

Tasks	Detailed steps
<p>4. Test your code.</p> 	<ul style="list-style-type: none"> a. Press F5 to build and run your application. b. In your application window, click Run, and verify that the output matches the rules listed above. c. In the NumericUpDown control, shown at the left, delete the existing value, and then type 5000 d. Click Run and verify that the output matches the rules listed above. e. In the NumericUpDown control, change the value to 4999 f. Click Run and verify that the output matches the rules listed above.
<p>5. Use the debugging tool to step through the code.</p>	<ul style="list-style-type: none"> a. In Solution Explorer, click Form1.cs, and then press F7. b. Locate the following line and set a breakpoint at that line. <code>int visitors = (int) visitorsUpDown.Value;</code> c. Press F5 to compile and run the application. d. If the Locals window is not visible, in Visual Studio .NET, on the Debug menu, point to Windows, and then click Locals. e. In your application window, click Run. f. Step through your code, a line at a time, by clicking the Step Over button, or by pressing F10. g. Examine the Locals and Autos windows to check that your application assigns values correctly. h. Stop debugging by clicking the Close button in the application that you are debugging or by pressing SHIFT+F5. i. Repeat this task and alter the input values to the application, so that the execution follows a different path.
<p>6. Save your application, and then quit Visual Studio .NET.</p>	<ul style="list-style-type: none"> a. Save your application. b. Quit Visual Studio .NET.

Lesson: Creating Iteration Statements

- How to Use a **for** Loop
- How to Use a **while** Loop
- How to Use a **do** Loop

Introduction

C# provides several looping mechanisms, which enable you to execute a block of code repeatedly until a certain condition is met. In each case, a statement is executed until a Boolean expression returns **true**. By using these looping mechanisms, you can avoid typing the same line of code over and over.

Lesson objectives

After completing this lesson, you will be able to:

- Write a **for** loop.
- Write a **while** loop.
- Write a **do** loop.

Lesson agenda

This lesson includes the following topics and activity:

- How to Use a **for** Loop
- How to Use a **while** Loop
- How to Use a **do** Loop
- Practice: Using Iteration Statements

How to Use a *for* Loop

- Use when you know how many times you want to repeat the execution of the code

```
for (initializer; condition; iterator) {  
    statements;  
}
```

Example

```
for (int i = 0; i < 10; i++) {  
    Console.WriteLine("i = {0}", i);  
}  
  
for (int j = 100; j > 0; j -= 10) {  
    Console.WriteLine("j = {0}", j);  
}
```

Introduction

A **for** loop is used to execute a statement block a set number of times. A **for** loop is a commonly-used way of executing a block of statements several times. The **for** loop evaluates a given condition, and while the condition is true, it executes a block of statements.

The **for** loop is called a pretest loop because the loop condition is evaluated before the loop statements are executed. If the loop condition tests false, the statements are not executed.

You use a **for** loop when you know in advance the number of times that you want to repeat execution of your code statement.

Example

For example, suppose that you are designing an application to calculate the amount of money that you will have in your savings account after 10 years with a given starting balance, and you want to display the total that you will have at the end of each year. One way that you can write this code is to write a statement like **balance *= interestRate** in your code ten times, or you can simply write a **for** loop.

Syntax

The syntax for declaring a **for** loop is:

```
for (initializer; condition; iterator) {  
    statement-block  
}
```

Example

```
for ( int i = 0; i < 10; i++ ) {  
    Console.WriteLine( "i = {0}",i );  
}  
  
for ( int j = 100; j > 0; j -= 10 ) {  
    Console.WriteLine( "j = {0}", j );  
}
```

This **for** structure is very flexible. For example, the loop counter can be incremented or decremented for each loop. In this case, you must know the number of loops before you write the loop.

Example of a decrementing loop

In the following example, **i** has decrementing values from **10** through **1**:

```
for ( int i = 10; i > 0; i-- ) {  
    loop statements;  
}
```

Example of an incrementing loop

In the following example, **i** has values of **0** to **100**, in incrementing steps of 10:

```
for ( int i = 0; i <= 100; i = i+10 ) {  
    loop statements;  
}
```

Declaring multiple variables

The **initializer** and **iterator** statements can contain more than one local variable declaration, as shown in the following example:

```
for ( int i = 0, j = 100; i < 100; i++, j-- ) {  
    Console.WriteLine("{0}, {1}", i, j );  
}
```

This sample would produce the following output:

```
0, 100  
1, 99  
2, 98  
.  
.  
.  
99, 1
```

How to Use a *while* Loop

- A Boolean test runs at the start of the loop and if it tests as **False**, the loop is never executed
- The loop executes until the condition becomes false

```
bool readingFile;  
  
// . . .  
  
while ( readingFile == true ) {  
    GetNextLine();  
}
```

- **continue, break**

Introduction

Similar to the **for** loop, the **while** loop is a pretest loop, which means that if the first test evaluates **false**, the statement does not execute. This is useful when you want to make sure that something is true before executing the code in your loop. You also use a **while** loop when you do not know exactly how many times you must execute the loop statements.

Syntax

The syntax for declaring a **while** loop is:

```
while (true-condition) {  
    statement-block  
}
```

Example

```
while ( readingFile == true ) {  
    GetNextLine();  
}
```

Using the **continue** keyword

You can use the **continue** keyword to start the next loop iteration without executing any remaining statements. The following example reads a set of commands from a file. **GetNextLine** gets a line of text; there is one command per line.

```
while ( readingFile == true ) {  
    string command = GetNextLine();  
    if ( command == "Comment" ) {  
        continue;  
    }  
    if ( command == "Set" ) {  
        // do other processing  
    }  
}
```

When the command is a comment, there is no need to process the rest of the line, so the **continue** keyword is used to start the loop again.

The break keyword

You can also break out of a loop. When the **break** keyword is encountered, the loop is terminated, and execution continues at the statement that follows the loop statement.

```
while ( readingFile == true ) {  
    string command = GetNextLine();  
    if ( command == "Exit" ) {  
        break;  
    }  
    if ( command == "Set" ) {  
        // do other processing  
    }  
}
```


How to Use a *do* Loop

- Executes the code in the loop and then performs a Boolean test. If the expression tests as True then the loop repeats until the expression tests as False.

```
do {  
    // something that is always going to happen  
    //at least once  
} while (test is true);
```

Example

```
int i = 1;  
do {  
    Console.WriteLine ("{0}", i++);  
} while (i <= 10);
```

Introduction

In a **do** loop, the statement is executed, a condition is tested, and then the statement is executed again. This process repeats for as long as the condition tests **true**. This is known as a post-test loop. The **do** loop is useful when you want to execute a statement at least once.

Syntax

The syntax for a **do** loop is:

```
do {  
    statements  
} while (boolean-expression);
```

Note The semicolon after the statement is required.

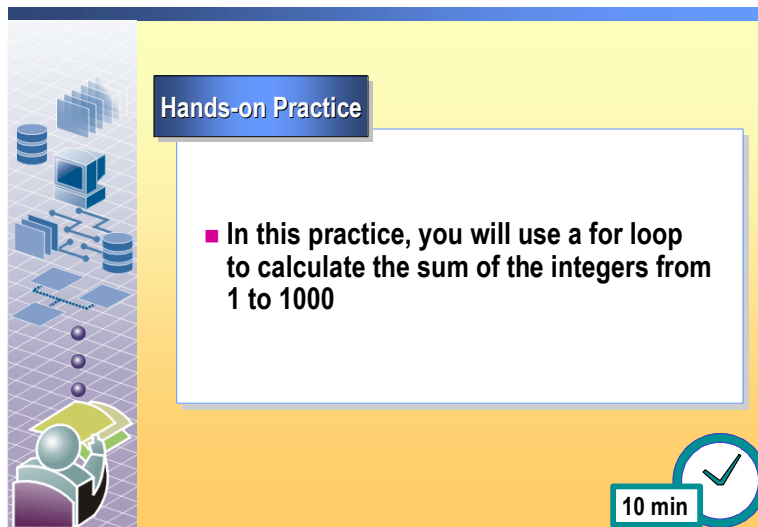
Example

In the following example, a **do** loop is used to write out the numbers from 1 to 10 in a column:

```
int i = 1;  
do {  
    Console.WriteLine("{0}", i++);  
} while ( i <= 10 );
```

In this example, the increment operator is used to increment the value of **i** after the statement is written to the screen for the first time.

Practice: Using Iteration Statements



Hands-on Practice

- In this practice, you will use a **for** loop to calculate the sum of the integers from 1 to 1000

10 min

In this practice, you will use a **for** loop to calculate the sum of the integers from 1 to 1000.

If time permits, perform the same calculation using a **while** loop and a **do** loop.

The solution code for this practice is located in *install_folder*\Practices\Mod02\Loops_Solution\Loops.sln. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Start Visual Studio .NET, and then open <i>install_folder</i> \Practices\Mod02\Loops\Loops.sln.	<ol style="list-style-type: none"> Start a new instance of Visual Studio .NET. On the Start Page, click Open Project. In the Open Project dialog box, browse to <i>install_folder</i>\Practices\Mod02\Loops, click Loops.sln, and then click Open. In Solution Explorer, click Form1.cs, and then press F7 to open the Code Editor.
2. Locate the task TODO 1: writing loops .	<ol style="list-style-type: none"> On the View menu, point to Show Tasks, and then click All. In the Task List, double-click TODO 1: writing loops.
3. Use a for loop to add all of the integers from 1 to 1000.	<ol style="list-style-type: none"> Write a for loop to add all of the integers from 1 to 1000. Place the result in an integer variable named total.
4. Display the result using the code shown in the right column.	<ul style="list-style-type: none"> ■ Display the result by using the following code: Output("result: " + total);
5. (Optional) Repeat steps 3 and 4, using a while loop instead of a for loop.	<ul style="list-style-type: none"> ■ (Optional) Repeat steps 3 and 4, using a while loop instead of a for loop.

Tasks	Detailed steps
6. (Optional) Repeat steps 3 and 4, using a do loop instead of a for loop.	<ul style="list-style-type: none">▪ (Optional) Repeat steps 3 and 4, using a do loop instead of a for loop.
7. Press F5 to build and run your application.	<ul style="list-style-type: none">▪ In your application window, click Run, and verify that the output is correct.
8. Save your application and quit Visual Studio .NET.	<ul style="list-style-type: none">a. Save your application.b. Quit Visual Studio .NET.

Review

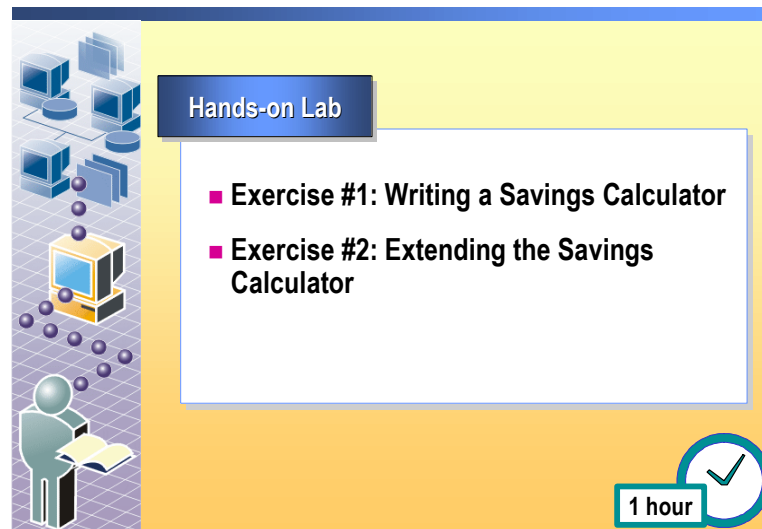
- Understanding the Fundamentals of a C# Program
- Using C# Predefined Types
- Writing Expressions
- Creating Conditional Statements
- Creating Iteration Statements

-
1. What symbol indicates a single-line comment in your code?
 2. True or false: You end a statement with a closing brace and a semicolon.
 3. What is the largest value that can fit in a byte?
 4. In the following expression, what is the value of y?
`int x = 50;`
`int y = ++x;`

5. Fill in the blank: A _____ statement allows you to control the flow of your application by selecting the statement that is executed, based on the value of a Boolean expression.

6. True or False: The **while** loop is a pre-test loop.

Lab 2.1: Writing a Savings Account Calculator



Objectives

After completing this lab, you will be able to:

- Declare variables and assign values to them.
- Convert between types.
- Write looping statements.
- Write conditional statements.

Note This lab focuses on the concepts in this module and as a result may not comply with Microsoft security recommendations.

Prerequisites

Before working on this lab, you must have:

- Knowledge of C# pre-defined types.
- The ability to write looping statements in C#.
- The ability to write conditional statements in C#.

Estimated time to complete this lab:
60 minutes

Exercise 0

Lab Setup

The Lab Setup section lists the tasks that you must perform before you begin the lab.

Task	Detailed steps
<ul style="list-style-type: none">Log on to Microsoft Windows® as Student with a password of P@ssw0rd.	<ul style="list-style-type: none">Log on to Windows with the following account.<ul style="list-style-type: none">User name: StudentPassword: P@ssw0rd <p>Note that the 0 in the password is a zero.</p>

Note that by default the *install_folder* is C:\Program Files\Msdntrain\2609.

Exercise 1

Write a Savings Calculator

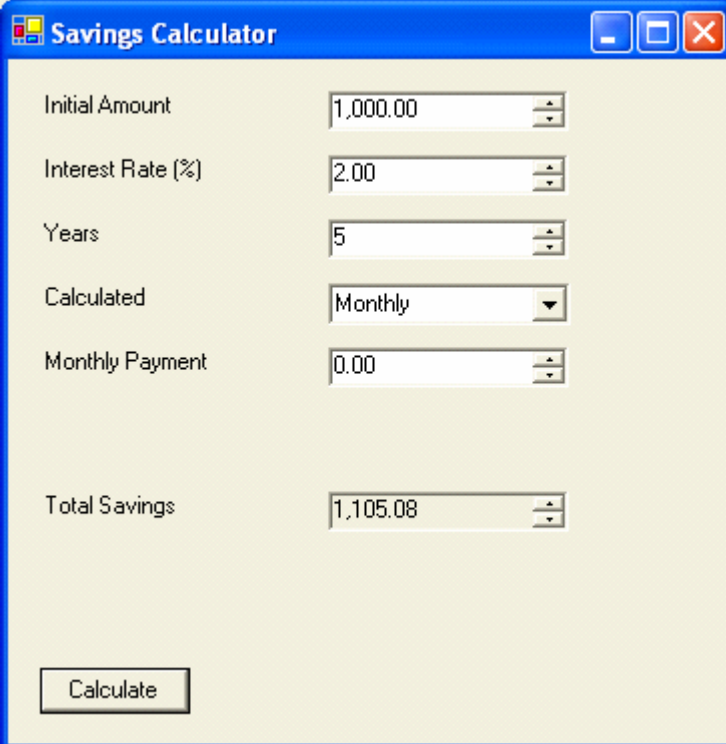
Scenario

Your bank wants to provide a simple savings calculator for account holders.

Details

In this exercise, you will write the code to complete a simple compound interest savings calculator. The user interface portion of the application is complete, but the code that performs the calculation is not written.

The application is shown in the following illustration:



The screenshot shows a Windows-style application window titled "Savings Calculator". The window has a blue title bar with standard minimize, maximize, and close buttons. The main area has a light beige background. It contains several input fields and a button:

- Initial Amount:** A text box containing "1,000.00" with up and down arrow buttons on the right.
- Interest Rate (%):** A text box containing "2.00" with up and down arrow buttons on the right.
- Years:** A text box containing "5" with up and down arrow buttons on the right.
- Calculated:** A dropdown menu currently showing "Monthly".
- Monthly Payment:** A text box containing "0.00" with up and down arrow buttons on the right.
- Total Savings:** A text box containing "1,105.08" with up and down arrow buttons on the right.
- Calculate:** A button located at the bottom left of the window.

Users enter the values, and when they click the **Calculate** button, the total is displayed in the **Total Savings** line.

To illustrate the logic of the program, three usage examples are described, each followed by a description of the logic.

Example 1:

The customer makes an initial payment of 1000, the annual interest rate for the account is 2%, and the calculation is for 5 years. The interest is calculated monthly and the extra money is added to the account.

This scenario requires that the application calculates the monthly interest rate by dividing the annual rate by 12, and then increases the account balance by the monthly interest every month for the period of the calculation.

Example 2:

The customer makes an initial payment of 2000, the annual interest rate for the account is 2.5%, and the calculation is for 10 years. The interest is calculated monthly and an extra monthly payment of 10 is added to the account.

This scenario requires an extra step of adding the extra monthly payment to the new monthly balance. Add this payment *after* the interest has been added to the balance. This will result in a total of 3929.10 (rounded to two decimal places).

Example 3:

Initial Amount = 5000, interest rate = 6%, years = 15, interest calculated monthly, monthly payments made of 100. Total is 41352.34.

Tasks	Detailed steps
1. Start Microsoft Visual Studio .NET, and then open <i>install_folder\Labfiles\Lab02_1\Exercise1\Saving.sln</i> .	<ol style="list-style-type: none"> Start a new instance of Visual Studio .NET. On the Start Page, click Open Project. In the Open Project dialog box, browse to <i>install_folder\Labfiles\Lab02_1\Exercise1</i>, click Saving.sln, and then click Open.
2. View the code of Form1.cs, and review the tasks to be performed in this exercise.	<ol style="list-style-type: none"> In Solution Explorer, click Form1.cs, and then press F7 to open the Code Editor. On the View menu, point to Show Tasks, and then click All. In the Task List, double-click TODO: calculate the value of the account.
3. Using the information provided in the introduction to this lab, write code that calculates the value of the savings account.	<ul style="list-style-type: none"> ▪ Note the following: <ul style="list-style-type: none"> • The variable startAmount contains the initial amount. • The variable rate contains the selected interest rate. • The variable years contains the number of years. • The variable calcFrequency is a variable of enumeration type Compound. This is defined at the top of the code file. It has one possible value—Compound.Monthly. • The variable additional contains the additional amount that the customer plans to save every month. • Assign your calculated total to the variable totalValue to have it displayed on the Windows form.

Tasks	Detailed steps
4. Test your application.	<ul style="list-style-type: none">■ Use the following values to check if your solution is correct (be sure to set the value of Calculated to Monthly):<ul style="list-style-type: none">• Initial Amount: 1000; Interest Rate 2%; Years: 5; Calculated: Monthly; Monthly Payment 0. Total Savings: 1105.08.• Initial Amount: 3500; Interest Rate 3.3%; Years: 7; Calculated: Monthly; Monthly Payment 50. Total Savings: 9125.56.• Initial Amount: 5000; Interest Rate 6.25%; Years: 10; Calculated: Monthly; Monthly Payment 250. Total Savings: 50856.56.
5. Save your application, and then quit Visual Studio .NET.	<ul style="list-style-type: none">a. Save your application.b. Quit Visual Studio .NET.

Exercise 2

Extending the Savings Calculator

In this exercise, you will add an option for quarterly interest calculations to the savings calculator. If you want to continue to use the application that you developed in Exercise 1, skip step 1.

When the quarterly interest option is selected, the interest is calculated per quarter, starting from the third month following the initial deposit. Any additional deposits are added to the balance after the interest for that month has been added.

The solution code for this lab is located at *install_folder*\Labfiles\Lab02_1\Exercise2\Solution_Code\Saving.sln. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Start Microsoft Visual Studio .NET and then open <i>install_folder</i> \Labfiles\Lab02_1\Exercise2\Saving.sln.	<ol style="list-style-type: none"> Start a new instance of Visual Studio .NET. On the Start Page, click Open Project. In the Open Project dialog box, browse to the folder <i>install_folder</i>\Labfiles\Lab02_1\Exercise2, click Saving.sln, and then click Open.
2. Add a new value called Quarterly to the enumeration type Compound .	<ol style="list-style-type: none"> In Solution Explorer, click Form1.cs, and then press F7 to open the Code Editor. Locate the enumeration Compound, at the top of the Code Editor. Add a new value Quarterly to the enumeration.
3. Follow the steps on the right to add the enumeration to the calculationFrequency combo box on the form.	<ol style="list-style-type: none"> Locate the following line of code: <code>calculationFrequency.Items.Add(Compound.Monthly);</code> Immediately after this line, add the following code: <code>calculationFrequency.Items.Add(Compound.Quarterly);</code> <p>Note that this code assumes that you have named your new enumeration value as described in step 2.</p> <p>This code adds the enumeration value to the combo box on the main form.</p>
4. Using the information provided in the introduction to this lab, write code that calculates the value of the savings account when interest is computed quarterly.	<ul style="list-style-type: none"> ■ Note the following: <ul style="list-style-type: none"> • The variable calcFrequency is a variable of enumeration type Compound, and when the user selects Quarterly from the menu, calcFrequency will have the value Quarterly, as defined in step 2.

Tasks	Detailed steps
5. Test your solution.	<ul style="list-style-type: none">■ Use the following values to check if your solution is correct:<ul style="list-style-type: none">• Initial Amount: 1000; Interest Rate 2%; Years: 5; Calculated: Quarterly; Monthly Payment 0. Total Savings: <i>1104.90</i>.• Initial Amount: 3500; Interest Rate 3.3%; Years: 7; Calculated: Quarterly; Monthly Payment 50. Total Savings: <i>9134.21</i>.• Initial Amount: 5000; Interest Rate 6.25%; Years: 10; Calculated: Quarterly; Monthly Payment 250. Total Savings: <i>50969.31</i>.
6. Save your application, and then quit Visual Studio .NET.	<ul style="list-style-type: none">a. Save your application.b. Quit Visual Studio .NET.