msdn™ training

Module 2: Overview of C#

Contents

Overview	1
Structure of a C# Program	2
Basic Input/Output Operations	9
Recommended Practices	15
Compiling, Running, and Debugging	22
Lab 2: Creating a Simple C# Program	36
Review	45

This course is based on the prerelease Beta 1 version of Microsoft® Visual Studio .NET. Content in the final release of the course may be different from the content included in this prerelease version. All labs in the course are to be completed with the Beta 1 version of Visual Studio .NET.

Microsoft™

Information in this document is subject to change without notice. The names of companies, products, people, characters, and/or data mentioned herein are fictitious and are in no way intended to represent any real individual, company, product, or event, unless otherwise noted. Complying with all applicable copyright laws is the responsibility of the user. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Microsoft Corporation. If, however, your only means of access is electronic, permission to print one copy is hereby granted.

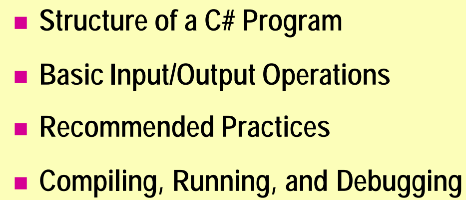
Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these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

© 2001 Microsoft Corporation. All rights reserved.

Microsoft, ActiveX, BizTalk, IntelliSense, JScript, Microsoft Press, MSDN, PowerPoint, Visual Basic, Visual C++, Visual #, Visual Studio, Windows, and Windows Media are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries.

Other product and company names mentioned herein may be the trademarks of their respective owners.

Overview



In this module, you will learn about the basic structure of a C# program by analyzing a simple working example. You will learn how to use the **Console** class to perform some basic input and output operations. You will also learn about some best practices for handling errors and documenting your code. Finally, you will compile, run, and debug a C# program.

After completing this module, you will be able to:

- Explain the structure of a simple C# program.
- Use the **Console** class of the **System** namespace to perform basic input/output operations.
- Handle exceptions in a C# program.
- Generate Extensible Markup Language (XML) documentation for a C# program.
- Compile and execute a C# program.
- Use the debugger to trace program execution.

Structure of a C# Program

In this section, you will learn about the basic structure of a C# program. You will analyze a simple program that contains all of the essential features. You will also learn how to use Microsoft® Visual Studio® to create and edit a C# program.

Hello, World

```
using System;

class Hello
{
  public static int Main( )
  {
    Console.WriteLine("Hello, World");
    return 0;
  }
}
```

The first program most people write when learning a new language is the inevitable Hello, World. In this module, you will get a chance to examine the C# version of this traditional first program.

The example code on the slide contains all of the essential elements of a C# program, and it is easy to test! When executed from the command line, it simply displays the following:

```
Hello, World
```

In the following topics, you will analyze this simple program to learn more about the building blocks of a C# program.

The Class

- A C# Application Is a Collection of Classes, Structures, and Types
- A Class Is a Set of Data and Methods
- Syntax

```
class name
{
      ...
}
```

- A C# Application Can Consist of Many Files
- A Class Cannot Span Multiple Files

In C#, an application is a collection of one or more classes, data structures, and other types. In this module, a class is defined as a set of data combined with methods (or functions) that can manipulate that data. In later modules, you will learn more about classes and all that they offer to the C# programmer.

When you look at the code for the Hello, World application, you will see that there is a single class called **Hello**. This class is introduced by using the keyword **class**. Following the class name is an open brace ({). Everything up to the corresponding closing brace (}) is part of the class.

You can spread the classes for a C# application across one or more files. You can put multiple classes in a file, but you cannot span a single class across multiple files.

Note for Java developers The name of the application file does not need to be the same as the name of the class.

Note for C++ developers C# does not distinguish between the definition and the implementation of a class in the same way that C++ does. There is no concept of a definition (.hpp) file. All code for the class is written in one file.

The Main Method

- **When Writing Main, You Should:**
  - Use an uppercase "M," as in "Main"
  - Designate one **Main** as the entry point to the program
  - Declare **Main** as **public static int Main**
- **Multiple Classes Can Have a Main**
- **When Main Finishes, or Returns, the Application Quits**

Every application must start somewhere. When a C# application is run, execution starts at the method called **Main**. If you are used to programming in C, C++, or even Java, you are already familiar with this concept.

Important The C# language is case sensitive. **Main** must be spelled with an uppercase "M" and with the rest of the name in lowercase.

Although there can be many classes in a C# application, there can only be one entry point. It is possible to have multiple classes each with **Main** in the same application, but only one **Main** will be executed. You need to specify which one should be used when the application is compiled.

The signature of **Main** is important too. If you use Visual Studio, it will be created automatically as **public static int**. (You will learn what these mean later in the course.) Unless you have a good reason, you should not change the signature.

Tip You can change the signature to some extent, but it must always be static, otherwise it might not be recognized as the application's entry point by the compiler.

The application runs either until the end of **Main** is reached or until a **return** statement is executed by **Main**.

The using Directive and the System Namespace

- The .NET Framework Provides Many Utility Classes
  - Organized into namespaces
- System Is the Most Commonly Used Namespace
- Refer to Classes by Their Namespace

```
System.Console.WriteLine("Hello, World");
```

- The using Directive

```
using System;
…
Console.WriteLine("Hello, World");
```

As part of the Microsoft .NET Framework, C# is supplied with many utility classes that perform a range of useful operations. These classes are organized into *namespaces*. A namespace is a set of related classes. A namespace may also contain other namespaces.

The .NET Framework is made up of many namespaces, the most important of which is called **System**. The **System** namespace contains the classes that most applications use for interacting with the operating system. The most commonly used classes handle input and output (I/O). As with many other languages, C# has no I/O capability of its own and therefore depends on the operating system to provide a C# compatible interface.

You can refer to objects in namespaces by prefixing them explicitly with the identifier of the namespace. For example, the **System** namespace contains the **Console** class, which provides several methods, including **WriteLine**. You can access the **WriteLine** method of the **Console** class as follows:

```
System.Console.WriteLine("Hello, World");
```

However, using a fully qualified name to refer to objects can be unwieldy and error prone. To ease this burden, you can specify a namespace by placing a **using** directive at the beginning of your application before the first class is defined. A **using** directive specifies a namespace that will be examined if a class is not explicitly defined in the application. You can put more than one **using** directive in the source file, but they must all be placed at the beginning of the file.

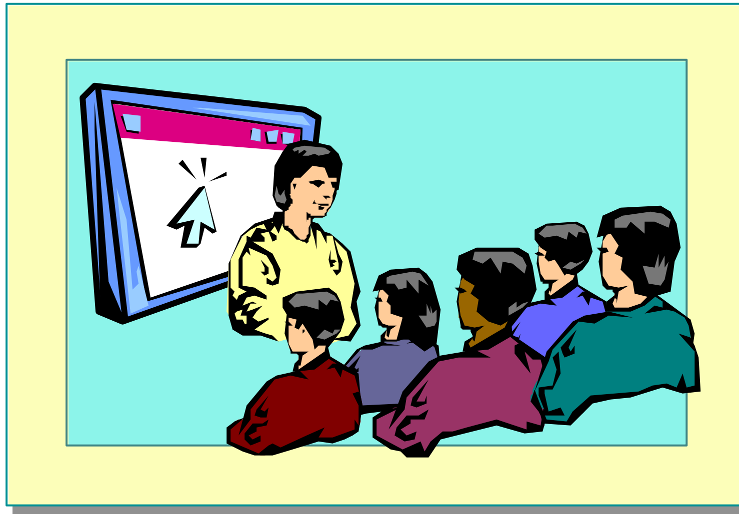
With the **using** directive, you can rewrite the previous code as follows:

```
using System;
...
Console.WriteLine("Hello, World");
```

In the Hello, World application, the **Console** class is not explicitly defined. When the Hello, World application is compiled, the compiler searches for **Console** and finds it in the **System** namespace instead. The compiler generates code that refers to the fully qualified name **System.Console**.

Note The classes of the **System** namespace, and the other core functions accessed at run time, reside in an assembly called mscorlib.dll. This assembly is used by default. You can refer to classes in other assemblies, but you will need to specify the locations and names of those assemblies when the application is compiled.

Demonstration: Using Visual Studio to Create a C# Program



In this demonstration, you will learn how to use Visual Studio to create and edit C# programs.

◆ Basic Input/Output Operations

- The Console Class
- Write and WriteLine Methods
- Read and ReadLine Methods

In this section, you will learn how to perform command-based input/output operations in C# by using the **Console** class. You will learn how to display information by using the **Write** and **WriteLine** methods, and how to gather input information from the keyboard by using the **Read** and **ReadLine** methods.

The Console Class

- Provides Access to the Standard Input, Standard Output, and Standard Error Streams
- Only Meaningful for Console Applications
  - Standard input - keyboard
  - Standard output - screen
  - Standard error - screen
- All Streams May Be Redirected

The **Console** class provides a C# application with access to the standard input, standard output, and standard error streams.

Standard input is normally associated with the keyboard—anything that the user types on the keyboard can be read from the standard input stream. Similarly, the standard output stream is usually directed to the screen, as is the standard error stream.

Note These streams and the **Console** class are only meaningful to console applications. These are applications that run in a Command window.

You can direct any of the three streams (standard input, standard output, standard error) to a file or device. You can do this programmatically, or the user can do this when running the application.

Write and WriteLine Methods

- Console.Write and Console.WriteLine Display Information on the Console Screen
  - **WriteLine** outputs a line feed/carriage return
- Both Methods Are Overloaded
- A Format String and Parameters Can Be Used
  - Text formatting
  - Numeric formatting

You can use the **Console.Write** and **Console.WriteLine** methods to display information on the console screen. These two methods are very similar; the main difference is that **WriteLine** appends a new line/carriage return pair to the end of the output, and **Write** does not.

Both methods are overloaded. You can call them with variable numbers and types of parameters. For example, you can use the following code to write "99" to the screen:

```
Console.WriteLine(99);
```

You can use the following code to write the message "Hello, World" to the screen:

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

Text Formatting

You can use more powerful forms of **Write** and **WriteLine** that take a format string and additional parameters. The format string specifies how the data is output, and it can contain markers, which are replaced in order by the parameters that follow. For example, you can use the following code to display the message "The sum of 100 and 130 is 230":

```
Console.WriteLine("The sum of {0} and {1} is {2}", 100, 130,
100+130);
```

Important The first parameter that follows the format string is referred to as parameter zero: {0}.

You can use the format string parameter to specify field widths and whether values should be left or right justified in these fields, as shown in the following code:

```
Console.WriteLine("Left justified in a field of width 10: {0,-
➥10}", 99);
Console.WriteLine("Right justified in a field of width 10:
➥{0,10}", 99);
```

This will display the following on the console:

“Left justified in a field of width 10: 99   ”

“Right justified in a field of width 10:   99”

Note You can use the backward slash (\) character in a format string to turn off the special meaning of the character that follows it. For example, "\{" will cause a literal "{" to be displayed, and "\\" will display a literal "\". You can use the at sign (@) character to represent an entire string verbatim. For example, "@\\server\share" will be processed as "\\server\share."

Numeric Formatting

You can also use the format string to specify how numeric data is to be formatted. The full syntax for the format string is `{N,M:FormatString}`, where `N` is the parameter number, `M` is the field width and justification, and `FormatString` specifies how numeric data should be displayed. The table below summarizes the items that may appear in **FormatString**. In all of these formats, the number of digits to be displayed, or rounded to, can optionally be specified.

Item	Meaning
C	Display the number as currency, using the local currency symbol and conventions.
D	Display the number as a decimal integer.
E	Display the number by using exponential (scientific) notation.
F	Display the number as a fixed-point value.
G	Display the number as either fixed point or integer, depending on which format is the most compact.
N	Display the number with embedded commas.
X	Display the number by using hexadecimal notation.

The following code shows some examples of how to use numeric formatting:

```
Console.WriteLine("Currency formatting - {0:C}  {1:C4}", 88.8,
➥-888.8);
Console.WriteLine("Integer formatting - {0:D5}", 88);
Console.WriteLine("Exponential formatting - {0:E}", 888.8);
Console.WriteLine("Fixed-point formatting - {0:F3}",
➥888.8888);
Console.WriteLine("General formatting - {0:G}", 888.8888);
Console.WriteLine("Number formatting - {0:N}", 8888888.8);
Console.WriteLine("Hexadecimal formatting - {0:X4}", 88);
```

When the previous code is run, it displays the following:

```
Currency formatting - $88.80  ($888.8000)
Integer formatting - 00088
Exponential formatting - 8.888000E+002
Fixed-point formatting - 888.889
General formatting - 888.8888
Number formatting - 8,888,888.80
Hexadecimal formatting – 0058
```

Note Custom format specifiers are available for dates and times. There are also custom format specifiers that allow you to create your own user-defined formats.

Read and ReadLine Methods

- Console.Read and Console.ReadLine Read User Input
  - **Read** reads the next character
  - **ReadLine** reads the entire input line

You can obtain user input from the keyboard by using the **Console.Read** and **Console.ReadLine** methods.

The Read Method

Read reads the next character from the keyboard. It returns the **int** value –1 if there is no more input available. Otherwise it returns an **int** representing the character read.

The ReadLine Method

ReadLine reads all characters up to the end of the input line (the carriage return character). The input is returned as a string of characters. You can use the following code to read a line of text from the keyboard and display it to the screen:

```
string input = Console.ReadLine( );
Console.WriteLine("{0}", input);
```

◆ Recommended Practices

In this section, you will learn some recommended practices to use when writing C# applications. You will be shown how to comment applications to aid readability and maintainability. You will also learn how to handle the errors that can occur when an application is run.

Commenting Applications

It is important to provide adequate documentation for all of your applications. Provide enough comments to enable a developer who was not involved in creating the original application to follow and understand how the application works. Use thorough and meaningful comments. Good comments add information that cannot be expressed easily by the code statements alone—they explain the "why" rather than the "what." If your organization has standards for commenting code, then follow them.

C# provides several mechanisms for adding comments to application code: single-line comments, multiple-line comments, and XML-generated documentation.

You can add a single-line comment by using the forward slash characters –(//). When you run your application, everything following these two characters until the end of the line is ignored.

You can also use block comments that span multiple lines. A block comment starts with the /* character pair and continues until a matching */ character pair is reached. You cannot nest block comments.

Generating XML Documentation

```
/// <summary> The Hello class prints a greeting
/// on the screen
/// </summary>
class Hello
{
  /// <remarks> We use console-based I/O.
  /// For more information about WriteLine, see
  /// <seealso cref="System.Console.WriteLine"/>
  /// </remarks>
  public static void Main( )
  {
    Console.WriteLine("Hello, World");
  }
}
```

You can use C# comments to generate XML documentation for your applications.

Documentation comments begin with three forward slashes (///) followed by an XML documentation tag. For examples, see the slide.

There are a number of suggested XML tags that you can use. (You can also create your own.) The following table shows some XML tags and their uses.

Tag	Purpose
<summary> …</summary>	To provide a brief description. Use the <remarks> tag for a longer description.
<remarks> …</remarks>	To provide a detailed description. This tag can contain nested paragraphs, lists, and other types of tags.
<para> …</para>	To add structure to the description in a <remarks> tag. This tag allows paragraphs to be delineated.
<list type="…"> …</list>	To add a structured list to a detailed description. The types of lists supported are "bullet," "number," and " table." Additional tags (<term> …</term> and <description> …</description>) are used inside the list to further define the structure.
<example> …</example>	To provide an example of how a method, property, or other library member should be used. It often involves the use of a nested <code> tag.
<code> …</code>	To indicate that the enclosed text is application code.

(*continued*)

Tag	Purpose
<c> …</c>	To indicate that the enclosed text is application code. The <code> tag is used for lines of code that must be separated from any enclosing description; the <c> tag is used for code that is embedded within an enclosing description.
<see cref="*member*"/>	To indicate a reference to another member or field. The compiler checks that "member" actually exists.
<seealso cref="*member*"/>	To indicate a reference to another member or field. The compiler checks that "member" actually exists. The difference between <see> and <seealso> depends upon the processor that manipulates the XML once it has been generated. The processor must be able to generate See and See Also sections for these two tags to be distinguished in a meaningful way.
<exception> …</exception>	To provide a description for an exception class.
<permission> …</permission>	To document the accessibility of a member.
<param name="*name*"> … </param>	To provide a description for a method parameter.
<returns> …</returns>	To document the return value and type of a method.
<value> …</value>	To describe a property.

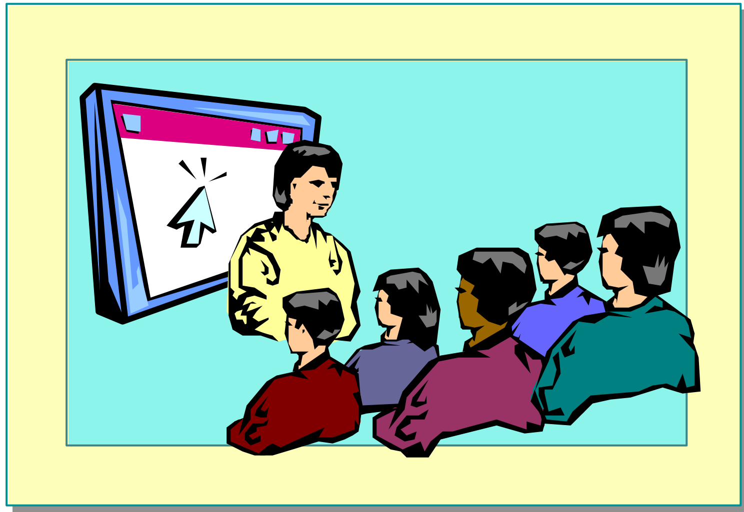
You can compile the XML tags and documentation into an XML file by using the C# compiler with the /doc option:

```
csc myprogram.cs /doc:mycomments.xml
```

If there are no errors, you can view the XML file that is generated by using a tool such as Internet Explorer.

Note The purpose of the /doc option is only to generate an XML file. To render the file, you will need another processor. Internet Explorer displays a simple rendition that shows the structure of the file and allows tags to be expanded or collapsed, but it will not, for example, display the <list type="bullet"> tag as a bullet.

Demonstration: Generating and Viewing XML Documentation



In this demonstration, you will see how to compile the XML comments that are embedded in a C# application into an XML file. You will also learn how to view the documentation file that is generated.

Exception Handling

```
using System;
public class Hello
{
  public static int Main(string[ ] args)
  {
     try {
         Console.WriteLine(args[0]);
     } catch (Exception e) {
         Console.WriteLine("Exception at
        ➥{0}", e.StackTrace);
     }
     return 0;
  }
}
```

A robust C# application must be able to handle the unexpected. No matter how much error checking you add to your code, there is inevitably something that can go wrong. Perhaps the user will type an unexpected response to a prompt, or will try to write to a file in a folder that has been deleted. The possibilities are endless.

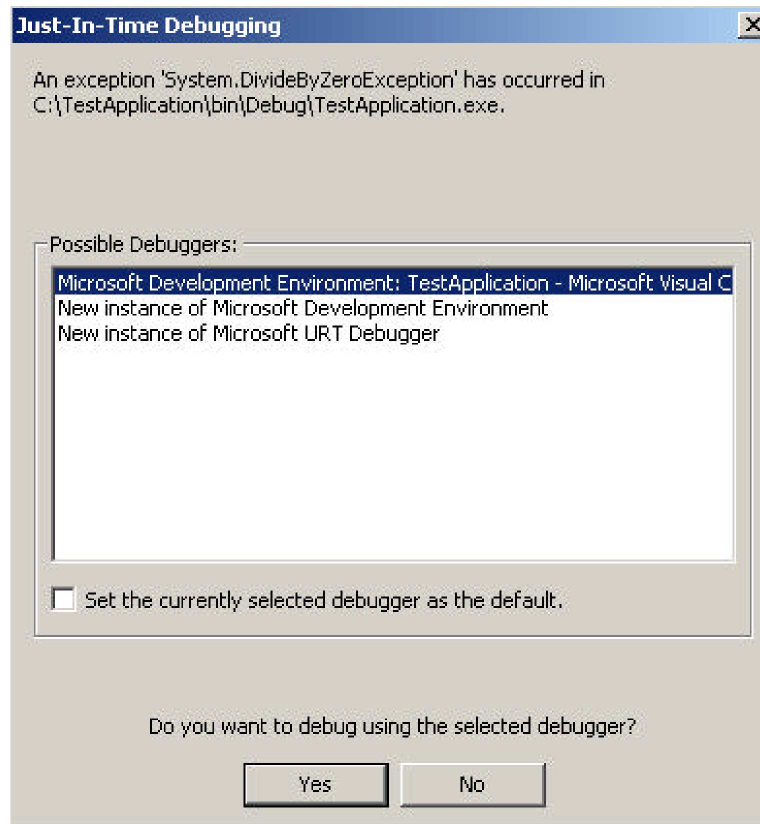
When a run-time error occurs in a C# application, the operating system throws an exception. Trap exceptions by using a **try-catch** construct as shown on the slide. If any of the statements in the **try** part of the application cause an exception to be raised, execution will be transferred to the **catch** block.

You can find out information about the exception that occurred by using the **StackTrace**, **Message**, and **Source** properties of the **Exception** object. You will learn more about handling exceptions in a later module.

Note If you print out an exception, by using **Console.WriteLine** for example, the exception will format itself automatically and display the **StackTrace**, **Message**, and **Source** properties.

Tip It is far easier to design exception handling into your C# applications from the start than it is to try to add it later.

If you do not use exception handling, a run-time exception will occur. If you want to debug your program using Just-in-time debugging instead, you need to enable it first. If you have enabled Just-in-time debugging, depending upon which environment and tools are installed, Just-in-time debugging will prompt you for a debugger to be used.



To enable Just-in-time debugging, perform the following steps:

1. On the **Tools** menu, click **Options**.
2. In the **Options** dialog box, click the **Debugging** folder.
3. In the **Debugging** folder, click **General**.
4. Click the **Settings** button.
5. Enable or disable Just-in-time (JIT) debugging for specific program types (for example, Win32 applications) in the **JIT Debugging Settings** dialog box, and then click **Close**.
6. Click **OK**.

You will learn more about the debugger later in this module.

Compiling, Running, and Debugging

- Invoking the Compiler
- Running the Application
- Demonstration: Compiling and Running a C# Program
- Debugging
- Multimedia: Using the Visual Studio Debugger
- The SDK Tools
- Demonstration: Using ILDASM

In this section, you will learn how to compile and debug C# programs. You will see the compiler executed from the command line and from within the Visual Studio environment. You will learn some common compiler options. You will be introduced to the Visual Studio Debugger. Finally, you will learn how to use some of the other tools that are supplied with the Microsoft .NET Framework software development kit (SDK).

Invoking the Compiler

Before you execute a C# application, you must compile it. The compiler converts the source code that you write into machine code that the computer understands. You can invoke the C# compiler from the command line or from Visual Studio.

Note Strictly speaking, C# applications are compiled into Microsoft intermediate language (MSIL) rather than native machine code. The MSIL code is itself compiled into machine code by the Just-in-time (JIT) compiler when the application is run. However, it is also possible to compile directly to machine code and bypass the JIT compiler if required.

Common Compiler Switches

You can specify a number of switches for the C# compiler by using the **csc** command. The following table describes the most common switches.

Switch	Meaning
/?, /help	Displays the compiler options on the standard output.
/out	Specifies the name of the executable.
/main	Specifies the class that contains the **Main** method (if more than one class in the application includes a **Main** method).
/optimize	Enables and disables the code optimizer.
/warn	Sets the warning level of the compiler.
/warnaserror	Treats all warnings as errors that abort the compilation.
/target	Specifies the type of application generated.

(*continued*)

Switch	Meaning
/checked	Indicates whether arithmetic overflow will generate a run-time exception.
/doc	Processes documentation comments to produce an XML file.
/debug	Generates debugging information.

Compiling from the Command Line

To compile a C# application from the command line, use the **csc** command. For example, to compile the Hello, World application (Hello.cs) from the command line, generating debug information and creating an executable called Greet.exe, the command is:

```
csc /debug+ /out:Greet.exe Hello.cs
```

Important Ensure that the output file containing the compiled code is specified with an .exe suffix. If it is omitted, you will need to rename the file before you can run it.

Compiling from Visual Studio

To compile a C# application by using Visual Studio, open the project containing the C# application, and click **Build** on the **Build** menu.

Note By default, Visual Studio opens the debug configuration for projects. This means that a debug version of the application will be compiled. To compile a release build that contains no debug information, change the solution configuration to release.

You can change the options used by the compiler by updating the project configuration:

1. In Solution Explorer, right-click the project icon.
2. Click **Properties**.
3. In the **Property Pages** dialog box, click **Configuration Properties**, and then click **Build**.
4. Specify the required compiler options, and then click **OK**.

Locating Errors

If the C# compiler detects any syntactic or semantic errors, it will report them.

If the compiler was invoked from the command line, it will display messages indicating the line numbers and the character position for each line in which it found errors.

If the compiler was invoked from Visual Studio, the Task List window will display all lines that include errors. Double-clicking each line in this window will take you to the respective error in the application.

Tip It is common for a single programming mistake to generate a number of compiler errors. It is best to work through errors by starting with the first ones found because correcting an early error may automatically fix a number of later errors.

Running the Application

- **Running from the Command Line**
  - Type the name of the application
- **Running from Visual Studio**
  - Click **Start Without Debugging** on the **Debug** menu

You can run a C# application from the command line or from within the Visual Studio environment.

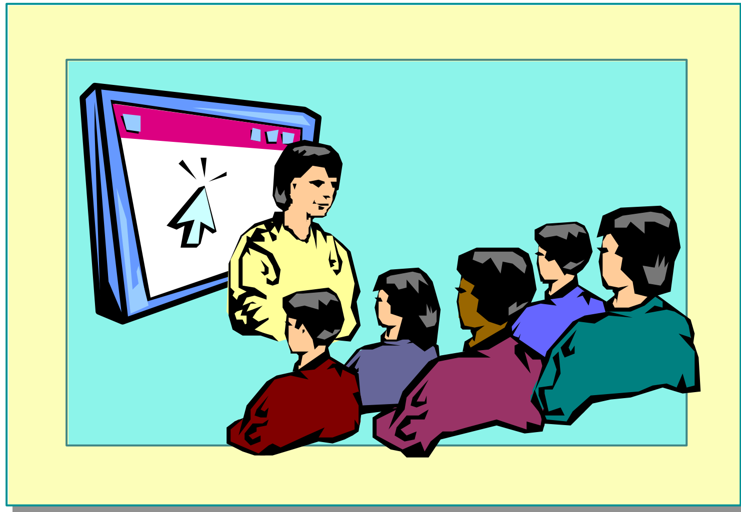
Running from the Command Line

If the application is compiled successfully, an executable file (a file with an .exe suffix) will be generated. To run it from the command line, type the name of the application (with or without the .exe suffix).

Running from Within Visual Studio

To run the application from Visual Studio, click **Start Without Debugging** on the **Debug** menu, or press CTRL+F5. If the application is a Console Application, a console window will appear automatically, and the application will run. When the application has finished, you will be prompted to press any key to continue, and the console window will close.

Demonstration: Compiling and Running a C# Program



In this demonstration, you will see how to compile and run a C# program by using Visual Studio. You will also see how to locate and correct compile-time errors.

Debugging

- **Exceptions and JIT Debugging**
- **The Visual Studio Debugger**
  - Setting breakpoints and watches
  - Stepping through code
  - Examining and modifying variables

Exceptions and JIT Debugging

If your application throws an exception and you have not written any code that can handle it, Common Language Runtime will instigate JIT debugging. (Do not confuse JIT debugging with the JIT compiler.)

Assuming that you have installed Visual Studio, a dialog box will appear giving you the choice of debugging the application by using the Visual Studio Debugger (Microsoft Development Env ironment), or the debugger provided with the .NET Framework SDK.

If you have Visual Studio available, it is recommended that you select the Microsoft Development Environment debugger.

Note The .NET Framework SDK provides another debugger: cordbg.exe. This is a command-line debugger. It includes most of the facilities offered by the Microsoft Development Environment, except for the graphical user interface. It will not be discussed further in this course.

Setting Breakpoints and Watches in Visual Studio

You can use the Visual Studio Debugger to set breakpoints in your code and examine the values of variables.

To bring up a menu with many useful options, right-click a line of code. Click **Insert Breakpoint** to insert a breakpoint at that line. You can also insert a breakpoint by clicking in the left margin. Click again to remove the breakpoint. When you run the application in debug mode, execution will stop at this line and you can examine the contents of variables.

The Watch window is useful for monitoring the values of selected variables while the application runs. If you type the name of a variable in the **Name** column, its value will be displayed in the **Value** column. As the application runs, you will see any changes made to the value. You can also modify the value of a watched variable by typing over it.

Important To use the debugger, ensure that you have selected the Debug solution configuration rather than Release.

Stepping Through Code

Once you have set any breakpoints that you need, you can run your application by clicking **Start** on the **Debug** menu, or by pressing F5. When the first breakpoint is reached, execution will halt.

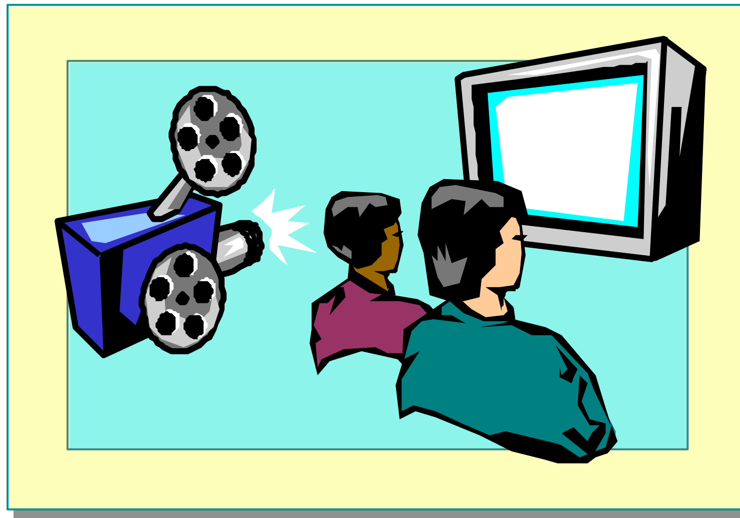
You can continue running the application by clicking **Continue** on the **Debug** menu, or you can use any of the single-stepping options on the **Debug** menu to step through your code one line at a time. You can use **Set Next Statement** on the **Debug** menu to jump backward or forward in your application and continue running from that point.

Tip The breakpoint, stepping, and watch variable options are also available on the **Debug** toolbar.

Examining and Modifying Variables

You can view the variables defined in the current method by clicking **Locals** on the **Debug** toolbar or by using the Watch window. You can change the values of variables by typing over them (as you can in the Watch window).

Multimedia: Using the Visual Studio Debugger



This multimedia demonstration will show you how to use the Visual Studio Debugger to set breakpoints and watches. It will also show you how to step through code and how to examine and modify the values of variables.

The SDK Tools

- General Tools and Utilities
- Win Forms Design Tools and Utilities
- Security Tools and Utilities
- Configuration and Deployment Tools and Utilities

The .NET Framework SDK is supplied with a number of tools that provide additional functionality for developing, configuring, and deploying applications. These tools can be run from the command line.

General Tools and Utilities

You may find some of the following general-purpose tools useful.

Tool name	Command	Description
NGWS Runtime Debugger	cordbg.exe	The command-line debugger.
MSIL Assembler	ilasm.exe	An assembler that takes MSIL as input and generates an executable file.
MSIL Disassembler	ildasm.exe	A disassembler that can be used to inspect the MSIL and metadata in an executable file.
PEVerify	peverify.exe	Validates the type safety of code and metadata prior to release.
Win Forms Class Viewer	wincv.exe	Locates managed classes and displays information about them.

Win Forms Design Tools and Utilities

You can use the following tools to manage and convert ActiveX® controls and Win Forms controls.

Tool name	Command	Description
Win Forms ActiveX Control Importer	aximp.exe	Generates a wrapper from an ActiveX control type library that allows the control to be hosted by a Win Forms form.
License Compiler	lc.exe	Produces a binary .licenses file for managed code from files containing licensing information.
Resource File Generation Utility	ResGen.exe	Produces a binary .resources file for managed code from text files that describe the resources.
ResX Resource Compiler	ResXToResources.exe	Produces a binary .resources file for managed code from .ResX (XML-based resource format) files that describe the resources.
Win Forms Designer Test Container	windes.exe	A tool for testing Win Forms controls.

Security Tools and Utilities

You can use the following tools to provide security and encryption features for .NET managed assemblies and classes.

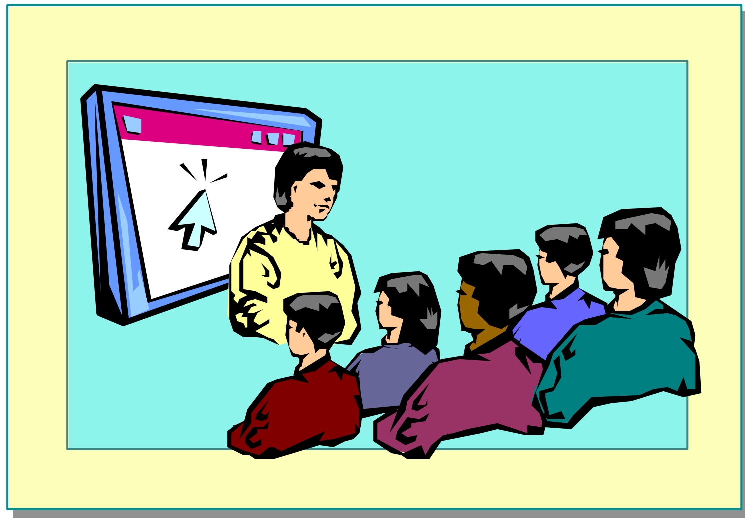
Tool name	Command	Description
Code Access Security Policy Utility	caspol.exe	Maintains machine and user code security policies.
Software Publisher Certificate Test Utility	cert2spc.exe	Creates a Software Publisher's Certificate from an X.509 certificate. This tool is used only for testing purposes.
Certificate Creation Utility	makecert.exe	An enhanced version of cert2spc.exe. It is also used only for testing purposes.
Certificate Manager Utility	certmgr.exe	Maintains certificates, certificate trust lists, and certificate revocation lists.
Certificate Verification Utility	chktrust.exe	Verifies the validity of a signed file.
Permissions View Utility	permview.exe	Views the permissions requested for an assembly.
Secutil Utility	SecUtil.exe	Locates public key or certificate information in an assembly.
Set Registry Utility	setreg.exe	Modifies registry settings related to public key cryptography.
File Signing Utility	signcode.exe	Signs an executable file or assembly with a digital signature.
Strong Name Utility	Sn.exe	Helps create assemblies that have strong names. It guarantees name uniqueness and provides some integrity. It also allows assemblies to be signed.

Configuration and Deployment Tools and Utilities

Many of the following tools are specialized tools that you will use only if you are integrating .NET managed code and COM classes.

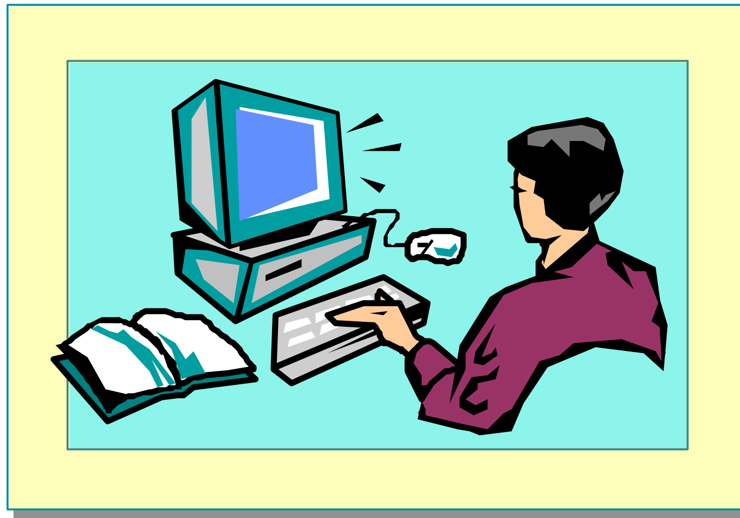
Tool name	Command	Description
Assembly Generation Utility	al.exe	Generates an assembly manifest from MSIL and resource files.
Assembly Registration Tool	RegAsm.exe	Enables .NET managed classes to be called transparently by COM components.
Services Registration Tool	RegSvcs.exe	Makes managed classes available as COM components by loading and registering the assembly and by generating and installing a COM+ type library and application.
Assembly Cache Viewer	shfusion.dll	Views the contents of the global cache. It is a shell extension used by Microsoft Windows® Explorer.
Isolated Storage Utility	storeadm.exe	Manages isolated storage for the user that is currently logged on.
Type Library Exporter	TlbExp.exe	Converts a .NET assembly into a COM type library.
Type Library Importer	Tlbimp.exe	Converts COM type library definitions into the equivalent metadata format for use by .NET.
Web Service Utility	WebServiceUtil.exe	Installs and uninstalls managed code Web services.
NGWS Runtime XML Schema Definition Tool	xsd.exe	Used for defining schemas that follow the World Wide Web Consortium (W3C) XML Schema Definition language.

Demonstration: Using ILDASM



In this demonstration, you will learn how to use Microsoft Intermediate Language (MSIL) Disassembler (ildasm.exe) to examine the manifest and MSIL code in a class.

Lab 2: Creating a Simple C# Program



Objectives

After completing this lab, you will be able to:

- Create a C# program.
- Compile and run a C# program.
- Use the Visual Studio Debugger.
- Add exception handling to a C# program.

Estimated time to complete this lab: 60 minutes

Exercise 1
Creating a Simple C# Program

In this exercise, you will use Visual Studio to write a C# program. The program will ask for your name and will then greet you by name.

To create a new C# console application

1. Start Microsoft Visual Studio.NET.
2. On the **File** menu, point to **New**, and then click **Project**.
3. Click **Visual C# Projects** in the **Project Types** box.
4. Click **Console Application** in the **Templates** box.
5. Type **Greetings** in the **Name** box.
6. Type ***install folder*\Labs\Lab02** in the **Location** box and click **OK**.
7. Type an appropriate comment for the summary.
8. Change the name of the class to **Greeter**.
9. Select and delete the public **Greeter( )** method.
10. Save the project by clicking **Save All** on the **File** menu.

To write statements that prompt and greet the user

1. In the **Main** method, before the **return** statement, insert the following line:

```
string myName;
```

2. Write a statement that prompts users for their name.
3. Write another statement that reads the user's response from the keyboard and assigns it to the *myName* string.
4. Add one more statement that prints "Hello *myName*" to the screen (where *myName* is the name the user typed in).
5. When completed, the **Main** method should contain the following:

```
public static int Main(string[ ] args)
{
  string myName;

  Console.WriteLine("Please enter your name");
  myName = Console.ReadLine( );
  Console.WriteLine("Hello {0}", myName);
  return 0;
}
```

6. Save your work.

To compile and run the program

1. On the **Build** menu, click **Build** (or press CTRL+SHIFT+B).
2. Correct any compilation errors and build again if necessary.
3. On the **Debug** menu, click **Start Without Debugging** (or press CTRL+F5).
4. In the console window that appears, type your name when prompted and press **ENTER**.
5. After the hello message is displayed, press a key at the "Press any key to continue" prompt.

Exercise 2
Compiling and Running the C# Program from the Command Line

In this exercise, you will compile and run your program from the command line.

To compile and run the application from the command line

1. Open a Command window.
2. Go to the *install folder*\Labs\Lab02\Greetings folder.
3. Compile the program by using the following command:

```
csc /out:Greet.exe Class1.cs
```

4. Run the program by entering the following:

```
Greet
```

5. Close the Command window.

Exercise 3
Using the Debugger

In this exercise, you will use the Visual Studio Debugger to single-step through your program and examine the value of a variable.

✓ To set a breakpoint and start debugging by using Visual Studio

1. Start Visual Studio.NET if it is not already running.
2. On the **File** menu, point to **Open** and then click **Project**.
3. Open the Greetings.sln project in the *install folder*\Labs\Lab02\Greetings folder.
4. Click in the left margin on the line containing the first occurrence of **Console.WriteLine** in the class **Greeter**.

   A breakpoint (a large red dot) will appear in the margin.
5. On the **Debug** menu, click **Start** (or press F5).

   The program will start running, a console window will appear, and the program will then halt at the breakpoint.

✓ To watch the value of a variable

1. On the **Debug** menu, point to **Windows**, and then click **Watch**.
2. In the Watch window, add the variable *myName* to the list of watched variables.
3. The *myName* variable will appear in the Watch window with a value of **null**.

✓ To single-step through code

1. On the **Debug** menu, click **Step Over** (or press F10) to run the first **Console.WriteLine** statement.
2. Bring the console window to the foreground.

   The prompt will appear.
3. Return to Visual Studio and single-step the next line containing the **Console.ReadLine** statement by pressing F10.
4. Return to the console window and type your name, and then press the RETURN key.

   You will automatically be returned to Visual Studio. The value of *myName* in the Watch window will be your name.
5. Single-step the next line containing the **Console.WriteLine** statement by pressing F10.

6. Bring the console window to the foreground.

   The greeting will appear.

7. Return to Visual Studio. On the **Debug** menu, click **Continue** (or press F5) to run the program to completion.

Note If you try to modify the value of *myName* in the Watch window, it will not change. This is because strings in C# are immutable and are handled differently than other types of variables, such as integers or other numerics (which would change as expected).

Exercise 4
Adding Exception Handling to a C# Program

In this exercise, you will write a program that uses exception handling to trap unexpected run-time errors. The program will prompt the user for two integer values. It will divide the first integer by the second and display the result.

To create a new C#program

1. Start Visual Studio.NET if it is not already running.
2. On the **File** menu, point to **New**, and then click **Project**.
3. Click **Visual C# Projects** in the **Project Types** box.
4. Click **Console Application** in the **Templates** box.
5. Type **Divider** in the **Name** box.
6. Type ***install folder*\Labs\Lab02** in the **Location** box and click **OK**.
7. Type an appropriate comment for the summary.
8. Change the name of the class to **DivideIt**.
9. Select and delete the public **DivideIt( )** method.
10. Save the project by clicking **Save All** on the **File** menu.

To write statements that prompt the user for two integers

1. In the **Main** method, before the **return** statement, insert the following lines:

```
int i, j;
string temp;
```

2. Write a statement that prompts the user for the first integer.
3. Write another statement that reads the user's response from the keyboard and assigns it to the *temp* string.
4. Add a statement to convert the string value in *temp* to an integer and to store the result in *i* as follows:

```
i = Int32.Parse(temp);
```

5. Add statements to your code to:
   a. Prompt the user for the second integer.
   b. Read the user's response from the keyboard and assign it to *temp.*
   c. Convert the value in *temp* to an integer and store the result in *j*.

Your code should look similar to the following:

```
int i, j;
string temp;

Console.WriteLine("Please enter the first integer");
temp = Console.ReadLine( );
i = Int32.Parse(temp);

Console.WriteLine("Please enter the second integer");
temp = Console.ReadLine( );
j = Int32.Parse(temp);
```

6. Save your work.

↙ To divide the first integer by the second and display the result

1. Write code to create a new integer variable *k* that is given the value resulting from the division of *i* by *j*, and insert it at the end of the previous procedure. Your code should look like the following:

```
int k = i / j;
```

2. Add a statement that displays the value of *k*.
3. Save your work.

↙ To test the program

1. On the **Debug** menu, click **Start Without Debugging** (or press CTRL+F5).
2. Type **10** for the first integer value and press ENTER.
3. Type **5** for the second integer value and press ENTER.
4. Check that the value displayed for *k* is 2.
5. Run the program again by pressing CTRL+F5.
6. Type **10** for the first integer value and press ENTER.
7. Type **0** for the second integer value and press ENTER.
8. The program causes an exception to be thrown (divide by zero).

To add exception handling to the program

1. Place the code in the **Main** method inside a **try** block as follows:

```
try {
    int i, j;
    string temp;
    ...
    int k = i / j;
    Console.WriteLine(...);
}
```

2. Add a **catch** statement to **Main**, before the **return** statement. The **catch** statement should print a short message, as is shown in the following code:

```
catch(Exception e) {
  Console.WriteLine("An exception was thrown: {0}" , e);
}
return 0;
...
```

3. Save your work.
4. The completed **Main** method should look similar to the following:

```
public static int Main(string[ ] args)
{
  try {
    int i, j;
    string temp;

     Console.WriteLine ("Please enter the first integer");
     temp = Console.ReadLine( );
     i = Int32.Parse(temp);

     Console.WriteLine ("Please enter the second integer");
     temp = Console.ReadLine( );
     j = Int32.Parse(temp);

     int k = i / j;
     Console.WriteLine("The result of dividing {0} by {1}
       ➥is {2}", i, j, k);
  } catch(Exception e) {
    Console.WriteLine("An exception was thrown: {0}", e);
  }

  return 0;
}
```

To test the exception-handling code

1. Run the program again by pressing CTRL+F5.
2. Type **10** for the first integer value and press ENTER.
3. Type **0** for the second integer value and press ENTER.

   The program still causes an exception to be thrown (divide by zero), but this time the error is caught and your message appears.

Review

- Structure of a C# Program
- Basic Input/Output Operations
- Recommended Practices
- Compiling, Running, and Debugging

1. Where does execution start in a C# application?

2. When does application execution finish?

3. How many classes can a C# application contain?

4. How many **Main** methods can an application contain?

5. How do you read user input from the keyboard in a C# application?

6. What namespace is the **Console** class in?

7. What happens if your C# application causes an exception to be thrown that it is not prepared to catch?