C# PROGRAMMING

TEXT BOOK:

E. Balagurusamy, “Programming in C# :A Primer”, Fourth Edition, McGraw Hill Education Private Limited, 2016.

Prepared by

B.Loganathan

C# PROGRAMMING

- **UNIT-V: Managing Errors and Exceptions**: Debugging – Types of errors - Exceptions-Syntax of exception-Handling code -Multiple catch statements -Exception hierarchy-General catch handler- Using finally statement- Nested try blocks- Throwing our own exceptions- Checked and unchecked operators-Using exceptions for debugging - **Window forms and Web-Based Application on .NET:** Creating window Forms - Customizing a form - Overview of design patterns-Web-based application on .NET.

Managing Errors and Exceptions

- Rarely does a program run successfully in the very first attempt. It is common to make mistakes while developing as well as typing a program. A mistake might lead to an error causing the program to produce unexpected results. *Errors* are mistakes that can make a program go wrong.
- An error may produce an incorrect output or may terminate the execution of the program abruptly or even may cause the system to crash. It is therefore important to detect and manage properly all the possible errors and error conditions in the program so that they do not terminate or cause the system to crash during execution.

DEBUGGING

- Debugging is the process of identifying and fixing errors in a software program so as to ensure that it behaves in the intended manner. In the software development domain, such errors are com1110nly referred as *bugs*.
- There are a number of debugging tools or debuggers that can be used for tracing the exact piece of code that is making the software behave in an inappropriate manner. Most of the IDEs are equipped with such in-built debuggers that help the programmers fix the bugs during development.

TYPES OF ERROR

- Errors may be broadly classified into two categories:
  - Compile-time errors
  - Run-time errors
- 1. Compile -Time Errors
- All syntax errors will be detected and displayed by the C# compiler and therefore these errors are known as *compile-time errors*. Whenever the coınpiler displays an error, it will not create the .cs file. It is therefore necessary that we fix all the errors before we can successfully compile and run the program .

2. **Run- time Errors**

- Sometimes , a program may compile successfully creating the **.exe** file but may not run properly. Such prograins may produce wrong results due to wrong logic or may terminate due to errors such as stack overflow. Most common run-time errors are:
  - Dividing an integer by zero.
  - Accessing an element that is out of the bounds of an array.
  - Trying to store a value into an array of an incompatible class or type.

EXCEPTIONS

- An *exception* is a condition that is caused by a run-time error in the program. When the C# compiler encounters an error such as dividing an integer by zero, it creates an exception object and throws it (i.e., informs us that an error has occurred).
- If the exception object is not caught and handled properly, the compiler will display an error message and will terminate the program. If we want the program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective actions. This task is known as *exception handling*.

- When writing programs, we must always be on the lookout for places in the program where an exception could be generated. Some common exceptions that we must catch are listed:
- SystemException : A foiled run-time check; used as a base class for other exceptions
- AccessExcepıion : Failure to access a type member, such as a method or field
- ArgumentException :An argument to a method was invalid
- ArgumenıNullException :A null argument was passed to a method that does not accept it
- ArgumentOutofRangeException :Argument value is out of range
- AriihmcıicException :Arithmetic over-or underflow has occurred
- ArrayType MismatchException :Attempt to store the wrong type of object in an array
- BadlmageFormatException :Image is in the wrong format
- CoreException :Base class for exceptions thrown by the runtime

- DivideByZeroExcepiton : An attempt was made to divide by zero
- FonnatExccption : The format of an argument is wrong
- lndexOu1ofRangeException :An array index is out of bounds
- lnva lidCastExccption :An attempt was made 10 cast 10 an invalid class
- lnvalidOperationException :A method was called at an invalid time
- Missing MemerExccption :An invalid version of a DLL was accessed
- NotFiniteNumberExceprio n :A number is not valid
- No1Suppor1edException :Indicates that a method is not implemented by a class
- NullReferenceExcpetion :Attempt to use an unassigned reference
- OutofMemoryException :Not enough memory 10 continue execution
- StackOverflow Exception :A s tack has overflowed

SYNTAX OF EXCEPTION HANDLING CODE

- The basic concepts of exception handling are *throwing* an exception and *catching* it.
- C# uses a keyword **try** to preface a block of code that is likely to cause an error condition and' throw' an exception. A catch block defined by the keyword **catch** 'catches' the exception ' thrown' by the try block and bandies it appropriately.
- The catch block is added immediately after the try block. The following example illustrates the use of simple **try** and **catch** statements.

- try
- {
- statement; / / generates an exception
- }
- catch *(Exception* e)
- {
- statement; / / processes the exception
- }
- The try block can have one or more statements that could generate an exception. If any one statement generates an exception, the remaining statements in the block are skipped and execution jumps to the catch block that is placed next to the try block.

MULTIPLE CATCH STATEMENT

- It is possible to have more than one catch statement in the catch block as illustrated below:

```
try
{
 statement          // generates an exception
}
 catch (Exception-Type-1 e)
{
statement;          // processes exception type 1
}
catch (Exception-Type-2 e)
{
statement;          //  processes exception type 2
}
 ……
 catch (Exception-type-N e)
{
statement ;         // processes exception type N
}
```

- When an exception in a try block is generated, the C# treats the multiple catch statements like cases in a **switch** statement. The first statement whose parameter matches with the exception object will be executed, and the remaining statements will be skipped.
- Note that C# does not require any processing of the exception at all. We can simply have a catch statement with an empty block to avoid program abortion.
- *Example:*
- cat ch (Excep tion e){ }
- This statement will catch an exception and then ignore it.

THE EXCEPTION HIERARCHY

- All C# exceptions are derived from the class **Exception.** When an exception occurs, the proper **catch** handler is determined by matching the type of exception to the name of the exception mentioned. If we are going to catch exceptions at different levels in the hierarchy, we need to put them in the right order.
- The rule is that we must always put the handlers for the most derived exception class first. Consider the following code snippet:
- try
- {
- .. . //throw Divide by Zero Except ion

- {
- catch(Exception e)
- {
- …
- }
- catch (DivideByZeroExcept ion e)
- {
- ….
- }
- This code will generate a compiler error, because the exception is caught by the first catch (which is a more general one) and the second catch is therefore unreachable.

- In C#, having unreachable code is always an error. The code must be rewritten as follows:
- try
- {
- .... //throw Divide By Zero Except ion
- }
- catch(DivideByZeroException e)
- {
- ....
- }
- catch(Exception e)
- {
- ....
- }

GENERAL CATCH HANDLE

- A catch block which will catch any exception is called a general catch handler. A general catch handler does not specify any parameter and can be written as:
- try
- {
- ..... // causes an exception
- }
- catch // no parameters
- [
- .... // handles error
-
- Note that **catch (Exception e)** can handle all the exceptions thrown by the C# code and therefore can be used as a general catch handler. However, if the program uses libraries written in other languages, then there may be an exception that is not derived from the class **Exception.**

- Such exceptions can be bandied by the parameter-less **catch** statement. This handler is always placed at the end. Since there is no parameter, it does not catch any information about the exception and the re fore we do not know what went wrong.

- **USING FINALLY STATEMENT**
- C# another statement known as a **finally** statement that can be used to handle an exception that is not caught by any of the previous catch statements. A **finally** block can be used to handle any exception generated within a try block.

- It may be added immediately after the try block or after the last catch block shown as follows:
- try
- {
- ….
- }
- catch(…)
- {
- ….
- }
- finally
- {
- ……
- }

- When a **finally** block is defined, the program is guaranteed to execute, regardless of how control leaves the try, whether it is due to normal termination, due to an exception occurring or due to a jump statement. As a result, we can use it to perform certain house-keeping operations such as closing files and releasing system resources.
- we may include the last two statements inside a finally block as shown below:
- finally
- {
- int y = a[1]/a[0J;
- Console .WriteLine("y = "+y);
- }

NESTED TRY BLOCKS

- C# permits us to nest try blocks inside each other. *Example:*
- try
- {
- ….(Point P1)
- try
- {
- . . . . (Point P2)
- }
- catch
- {
- ….(Point P3) //Inner try block
- }
- finally
- {

- }
- .....(Point P4)
- }
- catch
- {
- ....
- }
- finally
- {
- .....
- }

- For simplicity, we have shown only one **catch** bandier in each **try** block. However, we can string several **catch** handlers together in each place.
- When nested try blocks are executed, the exceptions that are thrown at various points are handled as follows:
  - The points P1 and P4 are outside the inner try block and therefore any exceptions thrown at these points will be bandied by the **catch** in the outer block. The inner block is simply ignored.
  - Any exception thrown at point P2 will be handled by the inner **catch** handler and the inner finally will be executed. The execution will continue at point P4 in the program.

- If there is no suitable catch handler to catch an exception thrown at P2, the control will leave the inner block (after executing the inner finally) and look for a suitable catch handler in the outer block. If a suitable one is found, then that handler is executed followed by the outer finally code. Remember, the code at point P4 will be skipped.
- If an exception is thrown at point P3, it is treated as if it had been thrown by the outer try block and, therefore, the control will immediately leave the inner block (of course, after executing the inner finally) and search for a suitable catch handler in the outer block.
- In case, a suitable **catch** handler is not found, then the system will terminate program execution with an appropriate message.

THROWING OUR OWN EXCEPTIONS

- There may be times when we would like to throw our own exceptions. We can do this by using the keyword throw as follows:
- throw *new Throwable_subclass;*
- *Examples:*
- throw *new* ArithmeticException( );
- throw *new* FormatException( );
- The object e which contains the error message "Number is too small" is caught by the **catch** block which then displays the message using the **Message** property.
- A rule of thumb when creating our own exceptional classes is that we must implement all the three **System. Exception** constructors .

CHECKED AND UNCHECKED OPERATORS

- Stack overflows are usual problems during arithmetic operations and conversion of integer types. C# supports two operators, **checked** and **unchecked**, which can be used for checking (or unchecking) stack overflows during program execution.
- If an operation is checked, then an exception will be thrown if overflow occurs. If it is not checked, no exception will be raised but we will lose data. For example, consider the code:
- int a = 200000
- int b = 300000

- try
- {
- int m = checked (a*b);
- }
- ca tch (OverflowException e)
- {
- Console.WriteLine (e);
- }
- Since a*b produces a value that will easily exceed the maximum value for an **int**, an overflow occurs. As the operation is checked with operator **checked**, an overflow exception will be thrown.

- In this case, we will get output like this:
- System.OverflowException : An exception
- Of type System.OverflowExecpt ion was thrown at …….
- If we want to suppress the overflow checking, we can mark the code as **unchecked**
- int n = **unchecked** (a* b);
- In this case, no exception will be raised, but we will lose data.
- **USING EXCEPTIONS FOR DEBUGGING**
- The exception-handling mechanism can be used to hide errors from rest of the program. It is possible that the programmers may misuse this technique for hiding errors rather than debugging the code. Exception handling mechanism may be effectively used to locate the type and place of errors. Once we identify the errors, we must try to find out why these errors occur before we cover them up with exception handlers.

WindowForms and Web-based Application Development on .NET

- .NET is a soft ware programming architecture provided by Microsoft for developing applications, which can be run on the Web through a network connection such as Internet or Intranet. The applications, which are developed for the Web using .NET are called Web based applications.
- NET supports the ASP. NET technology with C# programming for allowing software developers to create Web based applications, which can be used to perform tasks such as validating data received by the server from client computers and doing business on the Internet. The Web-based application can be created for allowing users who are surfing the Internet to do online shopping..

CREATING WINDOWFORMS

- Form is the very first entity typically included in a Windows-based application. It hosts a number of other controls for performing desired functions. At runtime, a form continuously waits for an event to occur, such as the clicking of the mouse or pressing of a key.
- As soon as an event occurs, it triggers the corresponding event-handling code. In C#, a form can be created by inheriting the **Form** class contained in the **System.Windows .Forms** namespace.

- The **Form** class already supports a number of properties and methods, which make the job of the programmer a lot easier. Let us now create a simple blank form by making use of the **Form** class:
- // Program - SampleForm.cs
- using System.Windows.Forms;
- public class SampleForm : Form
- {
- public static void Main ()
- {
- SampleForm F1 = new Sampleform();
- Application. Run(F1);
- }
- }
- In the above code, a **Windows** form named **SampleForm** has been created by inheriting the **Form** base class.

CUSTOMIZING A FORM

- We can customize a form's look and feel by making use of the various properties and methods of the **Form** class. Here, we'll modify the **SampleForm.cs** program that we created earlier to customize the form's caption bar, size, color and border.
- **1.Customizing the Caption Bar**
- The **Form** class supports a number of propeıties to enable the programmer to customize the form's caption as per his require ments. Some of these properties are:
  - **ControlBox:** Enables or disables the control box.
  - **Maximize Box:** Enables or disables the Maximize button.
  - **MinimizeBox:** Enables or disables the Minimize button.
  - **Text:** Helps add a caption for the form.

2. Customizing the Size

- The **Form** class supports a number of properties to enable the programmer to specify the size related settings of a form. Some of these propeıties are:
  - **DefaultSize:** Sets the default size of a form
  - **Height:** Sets the height of a form
  - **Width:** Sets the width of a form
  - **MaximumSize:** Sets the maxim um size of a form
  - **Mi nimumSize:** Sets the minimuıı size of a form
  - **StartPosition:** Helps specify the initial position of the form

3. Customizing the Colors

- The **BackColor** property of the **Form** class enables us to modify the background color of a form. The choice of the colors can be made from the **Color** structure contained in the **System.Drawing** namespace.

4. **Customizing the Borders**

- We can customize the border of a form by making use of the **FormBorderStyle** property of the **Form** class. The **FormBorderStyle** property allows us to not only change the border style but it also enables us to configure the resizing capability of a form. Some of the values that **FormBorderStyle** property can assume are following :
  - **None:** Removes the form's border
  - **Sizeable:** Makes the form resizable
  - **Fixcd3D:** Makes the form non resizable with a 3D border
  - **FixedSingle:** Makes the form non resizable with a single line border

OVERVIEW OF DESIGN PATTERNS

- Design patterns form the basis of software development by specifying design and implementation strategies to be used during development. Each design pattern has its own set of advantages and limitations. Thus, the choice of a particular design pattern depends primarily on the requirements to be fulfilled by the application as well its target environment. Some of the key design patterns suppoiied in .NET are :
  - Factory
  - Singleton
  - Chain of responsibility

- **1.Factory Design Pattern**
- The **Factory** design pattern makes use of abstract classes and interfaces to enable the programmer to choose the type of objects to instantiate. As the name suggests, the factory interface allows the creation of different product objects, the functionality for which is specified during the implementation of interface methods.

2.Singleton Design Pattern

- In this design pattern, a class is instantiated only once; that is, only a single unique object is created for the class. This object is then shared among different client applications through a static method call.
- The usability of such a design pattern can be seen in scenarios where some locking or synchronization mechanisms are required to be implemented.

3. **Chain of Responsibility Design Pattern**

- As the name suggests, the chain of responsibility design pattern links a series of objects together, each possessing the capability to handle the incoming request. The request is passed along the chain until one of the chained objects receives and handles the request.
- **WEB-BASED APPLICATION ON .NET**
- In Microsoft Visual Studio, we can use ASP.NET technology and a programming language such as C# to create a Web based application. The controls available in the **Toolbox** of the Microsoft Visual Studio IDE help us to create the user interface of a Web based application.

- For example, we can create a Web based application to accept name and city from the user and when the user clicks a button, the entered nanıe and city can be displayed.
- 1. **Creating and Running a Websitei Web-based Application**
- Websitei Web based application allows users to enter their information, such as name and address. When the user clicks a button on the **Default.aspx** page of the Websitei Web-based application, another page appears displaying the entered information.
- We can run the Website l Wcb-based application by pressing the **F5** key or the **Start Debugging** button of the toolbar, which is present in the Microsoft Visual Studio IDE.

2. **Creating a Website1 Web-based Application**

- A Website1 Web based application is an ASP.NET website containing pages such as **Default.aspx** and **show.aspx.** This is a web application in C# using ASP.NET.
- This website makes uses of **label, textbox, check box, radio button, dropdownlist, button** and **hyperlink** controls. The form checks that all the values are inserted and displays the data in another page. To create the Website1 Web-based application, we must:
- Create the **Default.aspx** page
- Create the **show.aspx** page
- Add code to the **show.aspx** page
- Add code for the **Default.aspx** page
- Create the **tcrms.aspx** page

- ***Creating the Default.aspx Page***
- The **Default.aspx** page of the Website1 Web-based application allows a user to enter details such as name and address and click a button to submit the details. To create the Default.asp x page:
- I . Open Microsoft Visual Studio IDE.
- 2. Select *File*-> *New*-> *Web Site* to display the **New Web Site** dialog box.
- 3 . Accept the default settings and click OK to close the **New Web Site** dialog box. The Microsoft Visual Studio IDE appears with the Source view of the **Dcfault.aspx** page, which is the default page of the Web Site1 Web based application.

- Change the value of the title element that appears in the **Source View** to *Using Controls in* ASP.NET and C#.
- Click the **Design** tab that appears at the bottom to display the **Design view** of the **Default.aspx** page.
- Drag a **Label Control** from **Toolbox** on to the **Dcfault.aspx** page to add the control to the page.
- Change the value of the **Tex t** property for the **Label** co ntrol to **Using Controls** in ASP.NET and
- C# using the **Properties** window.
- Change the **BackColor** propeıty of the **Label** control using the **Properties** window as required.

- Similarly, add four more Label controls to the Dcfault.aspx page.
- Change the Text Property of the first Label control that we have added to *Name* using the Proper ties dialog box.
- Change the Text Property of the second Label control that we have added to *Address* using the **Properties** dialog box .
- Change the **Text Property** of the third **Label** control that you have added to *City* using the **Properties** dialog box.
- Change the **Text Property** of the first **Label** control that we have added to *Gender* using the **Properties** dialog box.
- Drag a **TextBox** control from the **Toolbox** on to **Default.aspx** page to add the control to the page.