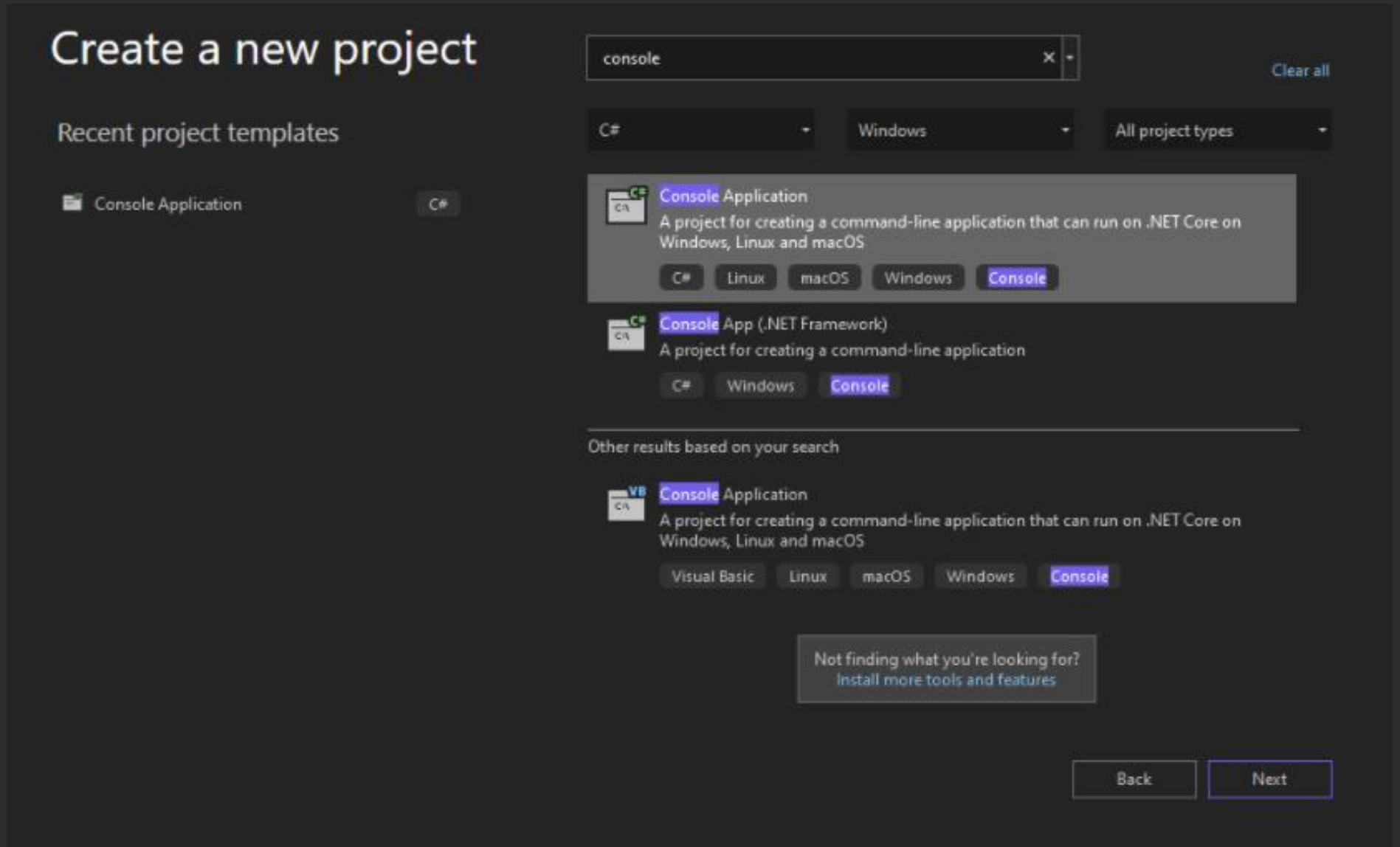
Lecture No 6

Muhammad Siddique

Outline

- Debugging
- Exception Handling
- File Handling

Debugging


Create a new project
console
Clear all
Recent project templates
C#
Windows
All project types
Console Application
C#
Console Application
A project for creating a command-line application that can run on .NET Core on Windows, Linux and macOS
C#
Linux
macOS
Windows
Console
Console App (.NET Framework)
A project for creating a command-line application
C#
Windows
Console
Other results based on your search
Console Application
A project for creating a command-line application that can run on .NET Core on Windows, Linux and macOS
Visual Basic
Linux
macOS
Windows
Console
Not finding what you're looking for?
Install more tools and features
Back
Next

Debugging

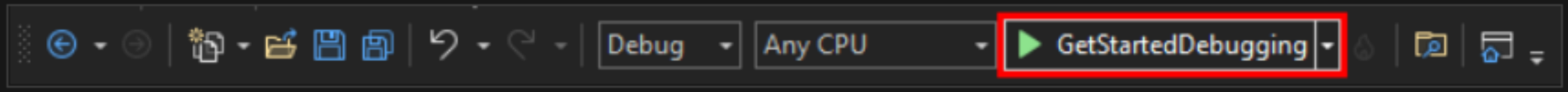
```
using System;

class ArrayExample
{
    static void Main()
    {
        char[] letters = { 'f', 'r', 'e', 'd', ' ', 's', 'm', 'i', 't', 'h'};
        string name = "";
        int[] a = new int[10];
        for (int i = 0; i < letters.Length; i++)
        {
            name += letters[i];
            a[i] = i + 1;
            SendMessage(name, a[i]);
        }
        Console.ReadKey();
    }

    static void SendMessage(string name, int msg)
    {
        Console.WriteLine("Hello, " + name + "! Count to " + msg);
    }
}
```

Debugging

Start the debugger!

Mostly, we use keyboard shortcuts here, because it's a fast way to execute debugger commands. Equivalent commands, such as toolbar or menu commands, are also noted.

1. To start the debugger, select **F5**, or choose the **Debug Target** button in the Standard toolbar, or choose the **Start Debugging** button in the Debug toolbar, or choose **Debug** > **Start Debugging** from the menu bar.



F5 starts the app with the debugger attached to the app process. Since we haven't done anything special to examine the code, the app runs to completion and you see the console output.

Debugging

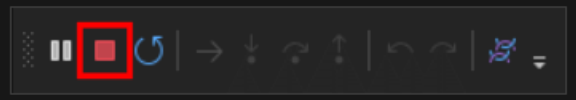
```
Hello, f! Count to 1
Hello, fr! Count to 2
Hello, fre! Count to 3
Hello, fred! Count to 4
Hello, fred ! Count to 5
Hello, fred s! Count to 6
Hello, fred sm! Count to 7
Hello, fred smi! Count to 8
Hello, fred smit! Count to 9
Hello, fred smith! Count to 10
```

Debugging

```
Hello, f! Count to 1
Hello, fr! Count to 2
Hello, fre! Count to 3
Hello, fred! Count to 4
Hello, fred ! Count to 5
Hello, fred s! Count to 6
Hello, fred sm! Count to 7
Hello, fred smi! Count to 8
Hello, fred smit! Count to 9
Hello, fred smith! Count to 10
```

Debugging

2. To stop the debugger, select **Shift+F5**, or choose the **Stop Debugging** button in the Debug toolbar, or choose **Debug** > **Stop Debugging** from the menu bar.



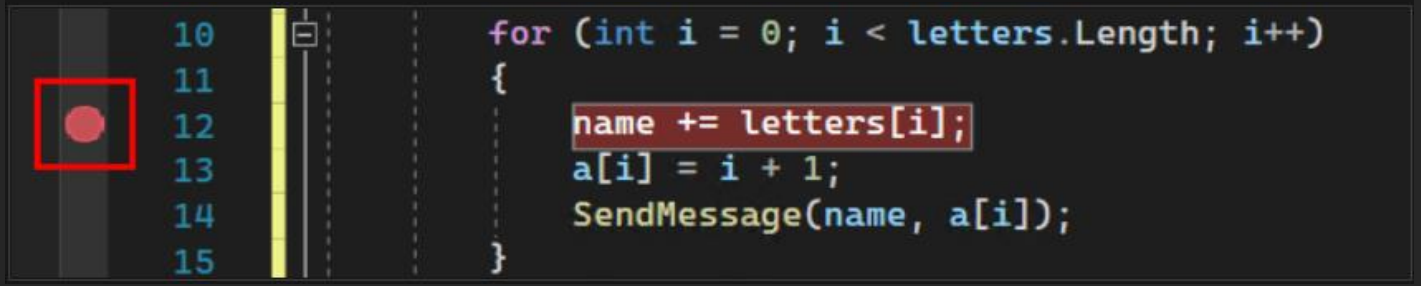
3. In the console window, select any key to close the console window.

Set a breakpoint and start the debugger

1. In the `for` loop of the `Main` function, set a breakpoint by clicking the left margin of the following line of code:

   ```
   name += letters[i];
   ```

   A red circle appears where you set the breakpoint.



Debugging

Breakpoints are an essential feature of reliable debugging. You can set breakpoints where you want Visual Studio to pause your running code so you can look at the values of variables or the behavior of memory, or know whether or not a branch of code is getting run.

2. To start debugging, select **F5**, or choose the **Debug Target** button in the Standard toolbar, or choose the **Start Debugging** button in the Debug toolbar, or choose **Debug** > **Start Debugging** from the menu bar. The app starts and the debugger runs to the line of code where you set the breakpoint.

Debugging

```
1   using System;
2
    0 references
3   class ArrayExample
4   {
        0 references
5       static void Main()
6       {
7           char[] letters = { 'f', 'r', 'e', 'd', ' ', 's', 'm', 'i', 't', 'h' };
8           string name = "";
9           int[] a = new int[10];
10          for (int i = 0; i < letters.Length; i++)
11          {
12              name += letters[i];
13              a[i] = i + 1;
14              SendMessage(name, a[i]);
15          }
16          Console.ReadKey();
17      }
18
        1 reference
19      static void SendMessage(string name, int msg)
20      {
21          Console.WriteLine("Hello, " + name + "! Count to " + msg);
22      }
23  }
```

Debugging

The yellow arrow points to the statement on which the debugger paused. App execution is paused at the same point, with the statement not yet executed.

When the app isn't running, **F5** will start the debugger, which will run the app until it reaches the first breakpoint. If the app is paused at a breakpoint, then **F5** will continue running the app until it reaches the next breakpoint.

Breakpoints are a useful feature when you know the line or section of code that you want to examine in detail. For more about the different types of breakpoints you can set, such as conditional breakpoints, see Using breakpoints.

Debugging

The yellow arrow points to the statement on which the debugger paused. App execution is paused at the same point, with the statement not yet executed.

When the app isn't running, **F5** will start the debugger, which will run the app until it reaches the first breakpoint. If the app is paused at a breakpoint, then **F5** will continue running the app until it reaches the next breakpoint.

Breakpoints are a useful feature when you know the line or section of code that you want to examine in detail. For more about the different types of breakpoints you can set, such as conditional breakpoints, see Using breakpoints.

Exception Handling

- An exception is a problem that arises during the execution of a program.
- A C# exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.
- Exceptions provide a way to transfer control from one part of a program to another.
- C# exception handling is built upon four keywords: **try**, **catch**, **finally**, and **throw**.

Try:

- A try block identifies a block of code for which particular exceptions is activated.
- It is followed by one or more catch blocks.

Exception Handling

Catch:

- A program catches an exception with an exception handler at the place in a program where you want to handle the problem.
- The catch keyword indicates the catching of an exception.

Finally:

- The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown.
- For example, if you open a file, it must be closed whether an exception is raised or not.

Exception Handling

Throw:

- A program throws an exception when a problem shows up.
- This is done using a throw keyword.

Exception Handling

- Assuming a block raises an exception, a method catches an exception using a combination of the try and catch keywords.
- A try/catch block is placed around the code that might generate an exception.
- Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following syntax in the figure.
- You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

```
try {
   // statements causing exception
} catch( ExceptionName e1 ) {
   // error handling code
} catch( ExceptionName e2 ) {
   // error handling code
} catch( ExceptionName eN ) {
   // error handling code
} finally {
   // statements to be executed
}
```

Exception Classes in C#

- C# exceptions are represented by classes.
- The exception classes in C# are mainly directly or indirectly derived from the System.Exception class.
- Some of the exception classes derived from the System.Exception class are the System.ApplicationException and System.SystemException classes.
- The System.ApplicationException class supports exceptions generated by application programs.
- Hence the exceptions defined by the programmers should derive from this class.
- The System.SystemException class is the base class for all predefined system exception.

Exception Classes in C#

- The following table provides some of the predefined exception classes derived from the Sytem.SystemException class.
- **System.IO.IOException**

  Handles I/O errors.

- **System.IndexOutOfRangeException**

  Handles errors generated when a method refers to an array index out of range.

- **System.ArrayTypeMismatchException**

  Handles errors generated when type is mismatched with the array type

- **System.NullReferenceException**

  Handles errors generated from referencing a null object.

Exception Classes in C#

- **System.DivideByZeroException**

  Handles errors generated from dividing a dividend with zero

- **System.InvalidCastException**

  Handles errors generated during typecasting.

- **System.OutOfMemoryException**

  Handles errors generated from insufficient free memory.

- **System.StackOverflowException**

  Handles errors generated from stack overflow.

Exception Handling - Example

```
using System;

namespace ErrorHandlingApplication {
   class DivNumbers {
      int result;

      DivNumbers() {
         result = 0;
      }
      public void division(int num1, int num2) {
         try {
            result = num1 / num2;
         } catch (DivideByZeroException e) {
            Console.WriteLine("Exception caught: {0}", e);
         } finally {
            Console.WriteLine("Result: {0}", result);
         }
      }
      static void Main(string[] args) {
         DivNumbers d = new DivNumbers();
         d.division(25, 0);
         Console.ReadKey();
      }
   }
}
```

Output:

```
Exception caught: System.DivideByZeroException: Attempted to divide by zero.
at ...
Result: 0
```

File Handling (Stream)

- When you open a file for reading or writing, it becomes stream. Stream is a sequence of bytes traveling from a source to a destination over a communication path.
- The two basic streams are input and output streams. Input stream is used to read and output stream is used to write.
- The System.IO namespace includes various classes for file handling.
- The parent class of file processing is stream.
- Stream is an abstract class, which is used as the parent of the classes that actually implement the necessary operations.
- The primary support of a file as an object is provided by a .NET Framework class called File.

File Handling (Stream)

- This static class is equipped with various types of (static) methods to create, save, open, copy, move, delete, or check the existence of a file.

Class Name	Description
FileStream	It is used to read from and write to any location within a file
BinaryReader	It is used to read primitive data types from a binary stream
BinaryWriter	It is used to write primitive data types in binary format
StreamReader	It is used to read characters from a byte Stream
StreamWriter	It is used to write characters to a stream.
StringReader	It is used to read from a string buffer
StringWriter	It is used to write into a string buffer
DirectoryInfo	It is used to perform operations on directories
FileInfo	It is used to perform operations on files

File Handling (StreamWriter Class)

- The StreamWriter class in inherited from the abstract class TextWriter. The TextWriter class represents a writer, which can write a series of characters.
- The following table describes some of the methods used by StreamWriter class.

Methods	Description
Close	Closes the current StreamWriter object and the underlying stream
Flush	Clears all buffers for the current writer and causes any buffered data to be written to the underlying stream
Write	Writes to the stream
WriteLine	Writes data specified by the overloaded parameters, followed by end of line

File Handling (Program to write user input to a file using Stream Writer Class)

```
using System;
using System.Text;
using System.IO;
namespace FileWriting_SW
{
    class Program
    {
        class FileWrite
        {
            public void WriteData()
            {
                FileStream fs = new FileStream("c:\\test.txt", FileMode.Append, FileAccess.Write);
                StreamWriter sw = new StreamWriter(fs);
                Console.WriteLine("Enter the text which you want to write to the file");
                string str = Console.ReadLine();
                sw.WriteLine(str);
                sw.Flush();
                sw.Close();
                fs.Close();
            }
        }
        static void Main(string[] args)
        {
            FileWrite wr = new FileWrite();
            wr.WriteData();
        }
    }
```

File Handling (StreamReader Class)

- The StreamReader class is inherited from the abstract class TextReader.
- The TextReader class represents a reader, which can read series of characters.
- The following table describes some methods of the StreamReader class.

Methods	Description
Close	Closes the object of StreamReader class and the underlying stream, and release any system resources associated with the reader
Read	Reads the next character or the next set of characters from the stream
ReadLine	Reads a line of characters from the current stream and returns data as a string
Seek	Allows the read/write position to be moved to any position with the file

File Handling (Program to write user input to a file using StreamReader Class)

```
using System;
using System.IO;
namespace FileReading_SR
{
    class Program
    {
        class FileRead
        {
            public void ReadData()
            {
                FileStream fs = new FileStream("c:\\test.txt", FileMode.Open, FileAccess.Read);
                StreamReader sr = new StreamReader(fs);
                Console.WriteLine("Program to show content of test file");
                sr.BaseStream.Seek(0, SeekOrigin.Begin);
                string str = sr.ReadLine();
                while (str != null)
                {
                    Console.WriteLine(str);
                    str = sr.ReadLine();
                }
                Console.ReadLine();
                sr.Close();
                fs.Close();
            }
        }
        static void Main(string[] args)
        {
            FileRead wr = new FileRead();
            wr.ReadData();
        }
    }
}
```