OVER 40 YEARS OF ACADEMIC WISDOM

GAIN MORE KNOWLEDGE REACH GREATER HEIGHTS

PRESIDENCY UNIVERSITY

.NET Full stack Development-CSE3152

Department of Computer Science Engineering

School of Engineering

Introduction to .NET

- Consistent **object-oriented programming environment**.
- To provide a **code-execution environment** that minimizes software deployment and versioning conflicts.
- To provide a code-execution environment that promotes safe execution of code, including code created by an unknown or semi-trusted third party.
- To make the developer experience consistent across widely varying types of applications, such as Windows-based applications and Web-based applications.
- To build all communication on industry standards to ensure that code based on the .NET Framework can integrate with any other code.

The .NET Framework provides a comprehensive programming model for building all kinds of applications on Windows, from mobile to web to desktop.

.NET and .NET Framework

- .NET is a developer platform made up of tools, programming languages, and libraries for building many different types of applications.
- There are various implementations of .NET. Each implementation allows .NET code to execute in different places—Linux, macOS, Windows, iOS, Android, and many more.
- **.NET Framework** is the original implementation of .NET. It supports running websites, services, desktop apps, and more on Windows.
- **.NET** is a cross-platform implementation for running websites, services, and console apps on Windows, Linux, and macOS. .NET is open source on GitHub. .NET was previously called .NET Core.
- **Xamarin/Mono** is a .NET implementation for running apps on all the major mobile operating systems, including iOS and Android.

.NET Framework

- Microsoft .NET Framework is a complex technology that provides the infrastructure for building, running, and managing next generation applications.
- In a layered representation, the .NET Framework is a layer positioned between the Microsoft Windows operating system and your applications.
- .NET is a platform but also is defined as a technology because it is composed of several parts such as libraries, executable tools, and relationships and integrates with the operating system.
- Can be described as Development platform or Execution environment which comprises tools and technologies, to develop distributed applications and distributed web services.
- Microsoft started development on the .NET Framework in the late 1990s originally under the name of Next Generation Windows Services.
- It consists of two major components: the **Common Language Runtime** (CLR), which provides memory management and other system services, and an extensive **Class Library**, which includes tested, reusable code for all major areas of application development

Features

- **Memory management**. In many programming languages, programmers are responsible for allocating and releasing memory and for handling object lifetimes. In .NET Framework applications, the CLR provides these services on behalf of the application.

- **A common type system**. In traditional programming languages, basic types are defined by the compiler, which complicates cross-language interoperability. In the .NET Framework, basic types are defined by the .NET Framework type system and are common to all languages that target the .NET Framework.

- **An extensive class library**. Instead of having to write vast amounts of code to handle common low-level programming operations, programmers can use a readily accessible library of types and their members from the .NET Framework Class Library.

- **Development frameworks and technologies**. The .NET Framework includes libraries for specific areas of application development, such as ASP.NET for web applications, ADO.NET for data access, and Windows Communication Foundation for service-oriented applications.

- **Language interoperability**. Language compilers that target the .NET Framework emit an intermediate code named Common Intermediate Language (CIL), which, in turn, is compiled at run time by the common language runtime. With this feature, routines written in one language are accessible to other languages, and programmers can focus on creating applications in their preferred language or languages.
- **Version compatibility**. With rare exceptions, applications that are developed by using a particular version of the .NET Framework can run without modification on a later version.
- **Side-by-side execution**. The .NET Framework helps resolve version conflicts by allowing multiple versions of the common language runtime to exist on the same computer. This means that multiple versions of applications can also coexist, and that an application can run on the version of the .NET Framework with which it was built.
- **Multitargeting.** By targeting the .NET Framework Portable Class Library, developers can create assemblies that work on multiple .NET Framework platforms, such as the .NET Framework, Silverlight, Windows Phone 7, or Xbox 360.

C# Features

C# is object oriented programming language. It provides a lot of **features** that are given below.

1.Simple

2.Modern programming language

3.Object oriented

4.Type safe

5.Interoperability

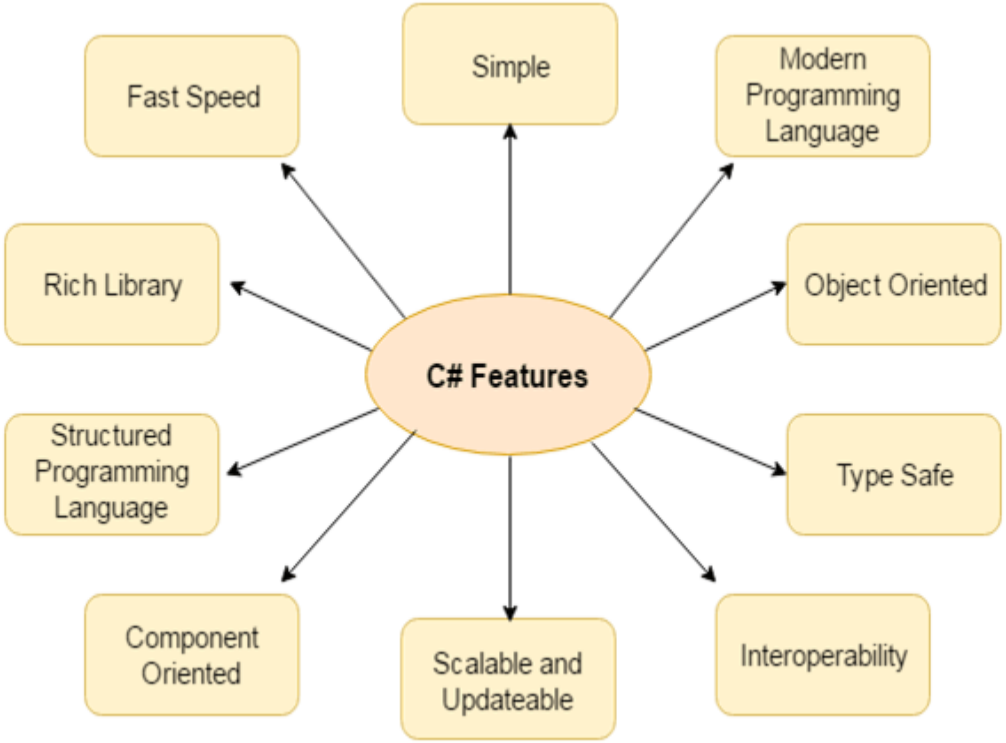
6.Scalable and Updateable

7.Component oriented

8.Structured programming language

9.Rich Library

10.Fast speed



1) Simple
C# is a simple language in the sense that it provides structured approach (to break the problem into parts), rich set of library functions, data types etc.

2) Modern Programming Language
C# programming is based upon the current trend and it is very powerful and simple for building scalable, interoperable and robust applications.

3) Object Oriented
C# is object oriented programming language. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grow.

4) Type Safe
C# type safe code can only access the memory location that it has permission to execute. Therefore it improves a security of the program.

5) Interoperability
Interoperability process enables the C# programs to do almost anything that a native C++ application can do.

6) Scalable and Updateable
C# is automatic scalable and updateable programming language. For updating our application we delete the old files and update them with new ones.

7) Component Oriented
C# is component oriented programming language. It is the predominant software development methodology used to develop more robust and highly scalable applications.

8) Structured Programming Language
C# is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify.

9) Rich Library
C# provides a lot of inbuilt functions that makes the development fast.

10) Fast Speed
The compilation and execution time of C# language is fast.

C# Arrays

Array in C# is a group of similar types of elements that have contiguous memory location. In C#, array is an object of base type **System.Array.** array index starts from 0. We can store only fixed set of elements in C# array.

EX: int[] arr = **new int**[5]; or **int**[] arr = **new int**[5]{ 10, 20, 30, 40, 50 }; or **int**[] arr = { 10, 20, 30, 40, 50 };

Advantages of C# Array

- Code Optimization (less code)
- Random Access
- Easy to traverse data
- Easy to manipulate data
- Easy to sort data etc.

Disadvantages of C# Array

- Fixed size

C# Array Types

There are 3 types of arrays in C# programming:

- Single Dimensional Array
- Multidimensional Array
- Jagged Array

C# Variables

Variables are containers for storing data values.
There are different types of variables

- int - stores integers (whole numbers), without decimals, such as 123 or -123
- double - stores floating point numbers, with decimals, such as 19.99 or -19.99
- char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
- string - stores text, such as "Hello World". String values are surrounded by double quotes
- bool - stores values with two states: true or false

- int myNum = 15;
  Console.WriteLine(myNum);
- string name = "John";
  Console.WriteLine(name);
- int myNum = 5;
- double myDoubleNum = 5.99D;
- char myLetter = 'D';
- bool myBool = true;
- string myText = "Hello";

Constants

This will declare the variable as "constant", which means unchangeable and read-only

The *'const'* keyword is useful when you want a variable to always store the same value, so that others (or yourself) won't mess up your code.

An example that is often referred to as a constant, is PI (3.14159...).

Note: You cannot declare a constant variable without assigning the value. If you do, an error will occur: A const field requires a value to be provided.

```
const int myNum = 15;
  myNum = 20; // error
```

Operators

An operator is a program element that is applied to one or more operands in an expression or statement.

- Operators that take one operand, such as the increment operator (++) or new, are referred to as unary operators.
- Operators that take two operands, such as arithmetic operators (+,-,*,/), are referred to as binary operators.
- One operator, the conditional operator (?:), takes three operands and is the sole ternary operator in C#.

Postfix increment X++ will add 1 to x

```
var x = 42;
x++;
Console.WriteLine(x); // 43
```

Postfix decrement X-- will subtract one

```
var x = 42;
x--;
Console.WriteLine(x); // 41
```

? : Ternary Operator

Returns one of two values depending on the value of a Boolean expression.

Syntax:

condition ? expression_if_true : expression_if_false;

string name = "Frank"; Console.WriteLine(name == "Frank" ? "The name is Frank" : "The name is not Frank");

Assignment operator '='

The assignment operator = sets thr left hand operand's value to the value of right hand operand, and return that

value:

int a = 3; // assigns value 3 to variable a

int b = a = 5; // first assigns value 5 to variable a, then does the same for variable b

Console.WriteLine(a = 3 + 4); // prints 7

Decision and iteration statements

Programming in general often requires a decision or a branch within the code to account for how the code operates under different inputs or conditions.

- If-Else Statement

```
static void PrintPassOrFail(int score)
{
 if (score >= 50) // If score is greater or equal to 50
 {
 Console.WriteLine("Pass!");
 }
 else // If score is not greater or equal to 50
 {
 Console.WriteLine("Fail!");
 }
}
```

- If-Else-if Statement

Following on from the If-Else Statement example. The Else If statement follows directly after the If statement in the If-Else. If-Else structure, but intrinsically has a similar syntax as the If statement. It is used to add more branches to the code than what a simple If-Else statement can.

```
static void PrintPassOrFail(int score)
{
 if (score > 100) // If score is greater than 100
 {
 Console.WriteLine("Error: score is greater than 100!");
 }
 else if (score < 0) // Else If score is less than 0
 {
 Console.WriteLine("Error: score is less than 0!");
 }
 else if (score >= 50) // Else if score is greater or equal to 50
 {
 Console.WriteLine("Pass!");
 }
 else // If none above, then score must be between 0 and 49
 {
 Console.WriteLine("Fail!");
 }
}
```

switch case statement

is a selection statement. C# switch case statement executes code of one of the conditions based on a pattern match with the specified match expression. The C# switch statement is an alternative to using the C# if else statement when there are more than a few options.

SYNTAX:

```
switch (expression)
{
  case expression_value1:
    Statement
    break;
  case expression_value2:
    Statement
  break;
  case expression_value3:
      Statement
  break;
  default:
    Statement
  break;
}
```

// Generate a random value between 1 and 9

```
int caseSwitch = new Random().Next(1, 9);
switch (caseSwitch)
{
  case 1:
    Console.WriteLine("Case 1");
  break;
  case 2:
    Console.WriteLine("Case 2");
  break;
  case 3:
    Console.WriteLine("Case 3");
  break;
  default:
    Console.WriteLine("Value didn't match earlier.");
  break;
}
```

A class in C# is a blueprint or template that is used for declaring an object. However, there is no need to declare an object of the static class. A class Encapsulates member variables, functions, properties etc. A method is a block of code in C# programming.

- The function makes program modular and easy to understand.
- It provides reusability of code and makes c# programming more secure
- It is the basic building block of object-oriented programming

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
 namespace Creating_Class
{    class accept //Creating 1st. class
    {       public string name;
        public void acceptdetails()
        {  Console.Write("Enter your name:\t");
            name = Console.ReadLine();
        }
    }
```

```
class print // Creating 2nd class
    {       public void printdetails()
        {
            //Creating object of 1st. class
            accept a = new accept();
            //executing method of 1st class.
            a.acceptdetails();
            //Printing value of name variable
            Console.WriteLine("Your name is " + a.name);
        }
    }
```

```
class Program //Creating 3rd class
    {
        static void Main(string[] args)
        {
            print p = new print();
            p.printdetails();
            Console.ReadLine();
        }
    }
}
```

OUTPUT:

Enter your name: SanthoshKumarKL

Your name is SanthoshKumarKL

Access Modifiers

Modifier	Description
public	The code is accessible for all classes
private	The code is only accessible within the same class
protected	The code is accessible within the same class, or in a class that is inherited from that class.
internal	The code is only accessible within its own assembly, but not from another assembly.

C# Properties (Get and Set)

- The meaning of Encapsulation, is to make sure that "sensitive" data is hidden from users. To achieve this, we must:

- declare fields/variables as '**private**'
- provide **public get and set** methods, through properties, to access and update the value of a private field

```
class Person
{
  private string name; // field
  public string Name   // property
  {
    get { return name; }
    set { name = value; }
  }
}
```

```
class Program
{
  static void Main(string[] args)
  {
    Person myObj = new Person();
    myObj.Name = "Santhosh";
    Console.WriteLine(myObj.Name);
  }
}
```

Events and Delegates:

Events are user actions such as key press, clicks, mouse movements, etc., or some occurrence such as system generated notifications. Applications need to respond to events when they occur. For example, interrupts. Events are used for inter-process communication.

Using Delegates with Events

- The events are declared and raised in a class and associated with the event handlers using delegates within the same class or some other class.
- The class containing the event is used to publish the event. This is called the publisher class.
- other class that accepts this event is called the subscriber class.
- Events use the publisher-subscriber model.
- A publisher is an object that contains the definition of the event and the delegate. The event-delegate association is also defined in this object. A publisher class object invokes the event and it is notified to other objects.
- A subscriber is an object that accepts the event and provides an event handler. The delegate in the publisher class invokes the method (event handler) of the subscriber class.

Declaring Events

To declare an event inside a class, first of all, you must declare a delegate type for the even as:

- **public delegate string BoilerLogHandler(string str);**

then, declare the event using the event keyword −

- **event BoilerLogHandler BoilerEventLog;**

```
using System;
namespace SampleApp
{
  public delegate string MyDel(string str);
  class EventProgram
 {
    event MyDel MyEvent;
    public EventProgram() {
      this.MyEvent += new MyDel(this.WelcomeUser);
    }
  public string WelcomeUser(string username) {
      return "Welcome To " + username;
    }
static void Main(string[] args) {
      EventProgram obj1 = new EventProgram();
      string result = obj1.MyEvent(“C# CLASSES");
      Console.WriteLine(result);
    }
  }
}
```

Delegates

- A Delegate is an abstraction of one or more function pointers. The .NET has implemented the concept of function pointers in the form of delegates. Delegates allow functions to be passed as parameters, returned from a function as a value, and stored in an array.
- A delegate can be defined as a delegate type. Its definition must be similar to the function signature.
- A delegate can be defined in a namespace and within a class. A delegate cannot be used as a data member of a class or local variable within a method. The prototype is:

  accessibility delegate return type delegatename(parameterlist);

  Delegate declarations look almost exactly like abstract method declarations, and replace the ‘abstract’ keyword with the ‘delegate’ keyword.

EXAMPLE:

```
using System;
namespace Delegates
{
    // Delegate Definition
    public delegate int operation(int x, int y);
    class Program
    {
        // Method that is passes as an Argument
        // It has same signature as Delegates
        static int Addition(int a, int b)
        {
            return a + b;
        }
        static void Main(string[] args)
        {
            // Delegate instantiation
            operation obj = new operation(Program.Addition);

            // output
            Console.WriteLine("Addition is={0}",obj(23,27));
            Console.ReadLine();
        }
    }
}
```

Extension Method

- Extension method is a special kind of static method that is called as if it was an instance method on the extended type.
- Extension method is a static method of a static class, where the "this" modifier is applied to the first parameter.
- The type of the first parameter will be the type that is extended.
- Extension methods enable to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type

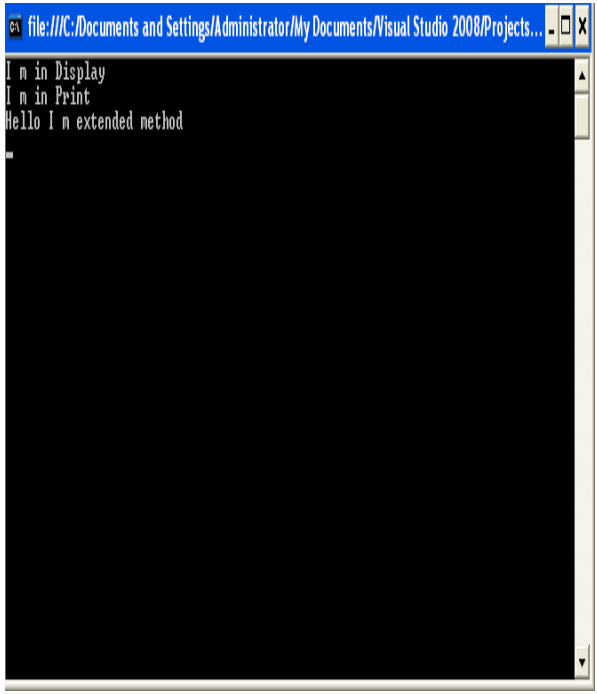
```
using System;
using System.Text;
using ClassLibExtMethod;

namespace ExtensionMethod1
{
    public static class XX
    {

        public static void NewMethod(this Class1 ob)
        {
            Console.WriteLine("Hello I m extended method");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Class1 ob = new Class1();
            ob.Display();
            ob.Print();
            ob.
                                                    y();
        }
    }
}
```

Display
Equals
GetHashCode
GetType
NewMethod
Print
ToString

(extension) void Class1.NewMethod()



Benefits of extension methods

- Extension methods allow existing classes to be extended without relying on inheritance or changing the class's source code.
- If the class is sealed, there is no concept of extending its functionality. For this, a new concept is introduced, in other words, extension methods.
- This feature is important for all developers, especially if you would like to use the dynamism of the C# enhancements in your class's design.

Sealed class:

- Sealed class is used to define the inheritance level of a class.
- class that cannot be inherited by any class but can be instantiated.
- The design intent of a sealed class is to indicate that the class is specialized and there is no need to extend it to provide any additional functionality through inheritance to override its behavior.
- We use sealed classes **to prevent inheritance**. As we cannot inherit from a sealed class, the methods in the sealed class cannot be manipulated from other classes. It helps to prevent security issues.
- Sealed class is the last class in the hierarchy.
- Sealed class can be a derived class but can't be a base class.
- A sealed class cannot also be an abstract class. Because abstract class has to provide functionality and here we are restricting it to inherit.

Sealed Methods

- Sealed method is used to define the overriding level of a virtual method.
- Sealed keyword is always used with override keyword.

Sealed Methods

- Sealed method is used to define the overriding level of a virtual method.
- Sealed keyword is always used with override keyword.

```
using System;
namespace SealedClass {
  class Animal {
    public virtual void makeSound() {
    Console.WriteLine("Animal Sound");
    }
  }
  class Dog : Animal {
    // sealed method
    sealed public override void makeSound() {
      Console.WriteLine("Dog Sound");
    }
  }
```

```
class Puppy : Dog {
    // trying to override sealed method
    public override void makeSound() {
      Console.WriteLine("Puppy Sound");
    }
  }
  class Program  {
    static void Main (string [] args) {
            // create an object of Puppy class
      Puppy d1 = new Puppy();
                          Console.ReadLine();
    }
  }
}
```

Sealed Method:

sealed keyword applies restrictions on the class and method. If you create a sealed class, it cannot be derived. If you create a sealed method, it cannot be overridden. It must be used with override keyword in method.

```
using System;
public class Animal{
    public virtual void eat() {
Console.WriteLine("eating..."); }
    public virtual void run() {
Console.WriteLine("running..."); }
 }
public class Dog: Animal
{
    public override void eat()
 { Console.WriteLine("eating bread..."); }
    public sealed override void run() {
    Console.WriteLine("running very fast...");
    }
}
```

```
public class BabyDog : Dog
{
    public override void eat()
{ Console.WriteLine("eating biscuits..."); }
    public override void run()
 { Console.WriteLine("running slowly..."); }
}
public class TestSealed
{
    public static void Main()
    {
        BabyDog d = new BabyDog();
        d.eat();
        d.run();
    }
}
```

Partial Classes:

It can break the functionality of a single class into many files. When the application is compiled, these files are then reassembled into a single class file. The partial keyword is used to build a partial class.

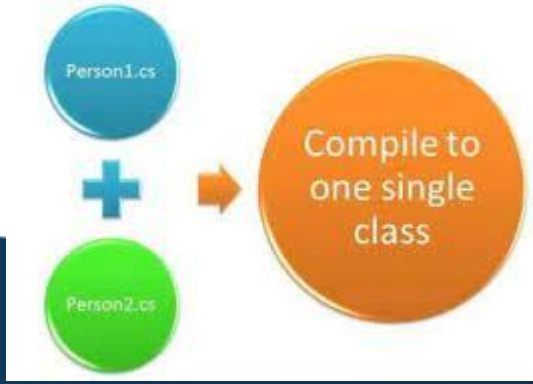
There are several situations when splitting a class definition is desirable:

- When working on large projects, spreading a class over separate files enables multiple programmers to work on it at the same time

- When working with automatically generated source, code can be added to the class without having to recreate the source file. Visual Studio uses this approach when it creates Windows Forms, Web service wrapper code, and so on.

- You can create code that uses these classes without having to modify the file created by Visual Studio.

- When using source generators to generate additional functionality in a class.



The **partial** keyword indicates that other parts of the class, struct, or interface can be defined in the namespace. All the parts must use the partial keyword. All the parts must be available at compile time to form the final type. All the parts must have the same accessibility, such as public , private , and so on

```
class Container
{
    partial class Nested
    {
        void Test() { }
    }

    partial class Nested
    {
        void Test2() { }
    }
}
```

Ex:

PartialClass1.cs

```
using System;

namespace PartialClasses
{
    public partial class PartialClass
    {
    public void HelloWorld()
    {
        Console.WriteLine("Hello, world!");
    }
    }
}
```

PartialClass2.cs

```
using System;

namespace PartialClasses
{
    public partial class PartialClass
    {
    public void HelloUniverse()
    {
        Console.WriteLine("Hello, universe!");
    }
    }
}
```

- Notice that both classes are defined with **the partial keyword** and have the same names.
- Also notice that each of them define a method - **HelloWorld() and HelloUniverse().**
- In this Program.cs we can now use this class as if it was defined in only one place, just like any other class:

```
using System;

namespace PartialClasses
{
    class Program
    {
    static void Main(string[] args)
    {
        PartialClass pc = new PartialClass();
        pc.HelloWorld();
        pc.HelloUniverse();
    }
    }
}
```

collections

collections are used to store and manage groups of related objects. There are two types of collections in C#: **Generic Collections and Non-Generic Collections.**

Generic Collections:

- Generic collections are strongly typed collections that can only hold a specific type of data.
- These collections are defined using generic types, which means that the type of data stored in the collection is determined at compile-time.
- **generic collections in C# are:**

1. **List<T>: It is a dynamic collection that stores a list of objects of a specific type**

List<int> stores a list of integers.

```
List<string> names = new List<string>();
names.Add("John");
names.Add("Alice");
names.Add("Bob");
```

Dictionary<TKey, TValue>:

It is a collection of key-value pairs where each key is associated with a value of a specific type.

Dictionary<string, int> stores a collection of string keys and integer values.

```
Dictionary<string, int> ages = new Dictionary<string, int>();
ages.Add("John", 30);
ages.Add("Alice", 25);
ages.Add("Bob", 40);
```

Non-Generic Collections:

- Non-generic collections are weakly typed collections that can hold objects of any type.
- These collections are defined using the Object class, which means that the type of data stored in the collection is determined at runtime.
- non-generic collections in C# are:

- **ArrayList: It is a collection of objects that can store any type of data.**

```
ArrayList list = new ArrayList();
list.Add("John");
list.Add(30);
list.Add(true);
```

- **Hashtable: It is a collection of key-value pairs where both the key and value can be of any type**

```
Hashtable data = new Hashtable();
 data.Add("Name", "John");
data.Add("Age", 30);
data.Add("IsMarried", true);
```

•Difference between Generic and Non Generic Collections

- The use of generic collections reduces the need for type casting and improves performance as there is no need to box or unbox the data.

- Examples of generic collections are **List<T>, Dictionary<TKey, TValue>, Queue<T>, Stack<T>, HashSet<T>, etc.**

- The use of non-generic collections requires type casting, which can cause runtime errors if the casting is done incorrectly.
- Non-generic collections can also result in performance overhead due to the need to box and unbox data.

- Examples of non-generic collections are **ArrayList, Hashtable, SortedList, Queue, Stack, etc.**

- Generic collections provide better type safety and improved performance, while non-generic collections offer more flexibility in the types of data that can be stored.
- The choice of which collection to use depends on the requirements of the specific use case.

Handling errors and exceptions

- Exception Handling is a process to handle runtime errors.
- We perform exception handling so that normal flow of the application can be maintained even after runtime errors.
- Exception is an event or object which is thrown at runtime.
- All exceptions the derived from **System.Exception class**.
- It is a runtime error which can be handled. If we don't handle the exception, it prints exception message and terminates the program.

Exception **Description**

- **System.FieldAccessException** →Handles the error generated by invalid private or protected field access.

- **System.IO.IOException**→ Handles the Input Output errors.
-
- **System.DivideByZeroException**→Handles the error generated by dividing a number with zero.

- **System.NullReferenceException**→Handles the error generated by referencing the null object.
- **System.InvalidCastException**→ Handles the error generated by invalid typecasting.

Exception Handling Keywords

we use 4 keywords to perform exception handling:

- try
- catch
- finally, and
- throw

try/catch example

```
using System;
public class ExExample
{
    public static void Main(string[] args)
    {
        try
        {
            int a = 10;
            int b = 0;
            int x = a / b;
        }
        catch (Exception e) { Console.WriteLine(e); }

        Console.WriteLine("Rest of the code");
    }
}
```

Finally

“finally” block is used to execute important code which is to be executed whether exception is handled or not. It must be preceded by catch or try block.

```
using System;
public class ExExample
{
    public static void Main(string[] args)
    {
        try
        {
            int a = 10;
            int b = 0;
            int x = a / b;
        }
 catch (Exception e) { Console.WriteLine(e); } // catch (NullReferenceException e) { Console.WriteLine(e); }
        finally { Console.WriteLine("Finally block is executed"); }
        Console.WriteLine("Rest of the code");
    }
}
```

```
using System;
public class InvalidAgeException : Exception
{
    public InvalidAgeException(String message)
        : base(message)
    {

    }
}
public class TestUserDefinedException
{
    static void validate(int age)
    {
        if (age < 18)
        {
            throw new InvalidAgeException("Sorry, Age must be greater than 18");
        }
    }
```

```
public static void Main(string[] args)
    {
        try
        {
            validate(12);
        }
        catch (InvalidAgeException e) { Console.WriteLine(e); }
        Console.WriteLine("Rest of the code");
    }
}
```

File Handling in C#

- File Handling involves management of files.
- Files are a collection of data that is stored in a disk with a name and a directory path.
- Files contain input as well as output streams that are used for reading and writing data respectively.

The System.IO Namespace

- The System.IO Namespace has various classes and types that allow reading of files and writing to files.
- **Stream**

  When we 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.

File.Create Method

- The File.Create method creates a file in the specified folder.

```
string fileName = @"C:\example.txt";
FileStream fs = File.Create(fileName);
fs.Close();
```

File.Delete Method

The File.Delete method deletes a file with the given name in a specified folder.

```
string fileName = @"C:\example.txt";
File.Delete(fileName);
```

File.Move Method

This method moves a specified file to a new location. We can specify a different name for the file in the new location.

```
string sourceFile = @"C:\example.txt";
string destFile = @"C:\example_moved.txt";
File.Move(sourceFile, destFile);
```

File.Open Method

This method is used to return a FileStream object at the specified path.

```
string fileName = @"C:\example.txt";
StreamReader sr = File.OpenText(fileName);
string line = sr.ReadLine();
sr.Close();
```

File.Copy Method

This method copies a file to the specified location.

```
string sourceFile = @"C:\example.txt";
string destFile = @"C:\example_copy.txt";
File.Copy(sourceFile, destFile, true); // set the third parameter to true to overwrite an existing file
```