C#

DataTypes in c#:

In C#, data types are used to define the type of data that a variable can hold. Here are the different data types available in C#:

1. Value Types:

Value types are the basic data types that store the actual value within the variable. Value types can be further categorized into the following:

- Boolean: This type represents a Boolean value (true or false).

- Numeric Types: This category includes several types that can hold different sizes of integers and floating-point numbers:

- sbyte, byte, short, ushort, int, uint, long, ulong: These are used to hold signed and unsigned integers of different sizes.

- float, double, decimal: These are used to hold floating-point numbers of different sizes and precision.

- Character Types: This includes char type, which is used to hold a single character value.

2. Reference Types:

Reference types do not store the actual value within the variable, but instead, store a reference to the memory location where the actual data is stored. Reference types can be further categorized into the following:

- Object Types: This includes the Object type, which is the base type for all other types in C#. All other reference types are derived from Object.

- String Type: This type represents a sequence of Unicode characters.

- Dynamic Type: This type allows the type of a variable to be determined at runtime.

- Pointer Type: This type allows you to work with unmanaged memory.

**KeyWords:**

1>.const

========

\*constant is a named memory location, whose value never changes during the execution of program once it is initialized.

There are 2 types of constants in c#:

.....................................

i).Compile time constant : is created using const keyword.

ii).Run time constant: is created using readonly keyword.

I).Compile time constant: \*is created using const keyword. \*must and should be initialized when it is declared.

-------------------------

Syntax: const datatype Name\_of\_constant =value;

------

Ex: const float PI =3.142f;

--- const float E =2.71828f;

II).Run time constant:

----------------------

\*It is created using readonly keyword.

\*can be initialized while declaring itself or later inside the constructor of a class at run time.

\*It is also called readonly variable.

\*run time constant must be a field of a class.

Syntax for declaring a constant using readonly keyword:

----------------------------------------------------

Ex:

---

readonly float PI;

readonly float E;

Syntax for initializing a readonly constant:

-----------------------------------------

Name\_of\_constant =value;

Ex: PI =3.142f;

E =2.71828f;

2).sealed:

==========

Sealed class cannot be inherited.

sealed class cannot behave as a parent class, but can behave as a child class.

Ex:

---

public sealed class Person //declaring a class sealed

{

public String name;

}

public class Student : Person //cannot inherit a sealed class, will throw error

{

public int id;

}

class Program

{

public static void Main(String[] args)

{

console.writeLine(" ");

}

}

3).this keyword:

===============

-this keyword is used to refer to the CURRENT INSTANCE of the class.

-this keyword avoids the name confusion between class fields and constructor parameters.

Ex:

---

public class Student

{

public String name;

public int id;

public Student(String name, int id)

{

this.name =name;

this.id =id;

}

public void GetStudent()

{

console.writleLine(id+" "+name);

}

}

public class Program

{

public static void Main(String[] args)

{

Student student = new Student();

student.GetStudent();

}

}

4).base keyword:

================

The base keyword in c# is used to access members of a base class from within a deriverd class.

Ex:

---

public class Base

{

public void print()

{

Console.WriteLine("base class method");

}

}

public class Derived : Base

{

public void print()

{

base.print();

Console.WriteLine("Derived class method");

}

}

public class MainEx

{

public static void Main(String[] args)

{

Derived derived = new Derived();

derived.print();

}

}

5)virtual keyword:

=================

The virtuak keyword is used for generating a virtual path for its derived classes on implementing method overriding. The virtual keyword is used within a set with an override keyword.

6).override:

===========

-override keyword is used in the derived class of the base class in order to override the base class method.

Ex:

---

public class A1

{

public virtual void show()

{

Console.WriteLine("Hello : Base class");

}

}

public class B1 : A1

{

public override void show()

{

Console.WriteLine("Hello : Derived class");

}

}

7).for each:

===========

-The for each loop is used to iterate over the elements of the collection.

-The collection may be an array or a list.It executes for each element present in the array.

-It is neccessary to enclose the statements of the foreach loop in curly braces {}.

Syntax:

for(data\_type var\_name in collection\_variable)

{

// statements

}

Ex:

---

class FOREACHEx

{

public static void Main(String[] args)

{

int[] array ={ 5, 78, 9, 65, 4, 1, 2 };

foreach(int value in array)

{

Console.WriteLine(value);

}

}

}

Enum:

====

An enum is a special 'class' that represents a group of constants(Unchangeable / read-only varidables).

To create an enum, use the enum keyword (Instead of class or interface), and seperate the enum items with a comma.

Ex:

---

enum Days

{

Monday = 1999,

Tuesday,

Wednesday,

Thursday,

Friday,

Saturday,

}

public class Days1

{

public static void Main(string[] args)

{

Console.WriteLine((int)Days.Monday);

}

}

struct:

======

structure is a value type that encapsulates a set of related data members. It is similar to class in that it can have fields, properties, and methods, but it differs in few important ways.

1. A struct is a value type, which means that when you create an instance of a struct, the actual data is stored in stack rather than heap. This makes struct more efficient it terms of memory usage and performance.

2. Structs cannot inherit from other structs or classes, and they cannot be inherited from. In other words, a struct cannot be used as a base class for another struct or class.

3. Structs are often used for small, lightweight objects are frequently used and passed around by value, such as coordinates or other simple data types.

Ex:

---

public struct Person

{

public String Name;

public int Age;

public int weight;

}

public class StructEx

{

public static void Main(string[] args)

{

Person P1;

P1.Name = "Abhishek";

P1.Age = 26;

P1.weight = 88;

Console.WriteLine("Name is: " + P1.Name + " Age is: " + P1.Age + " weight is: " + P1.weight);

}

}

**MSUnit Annotations:**

MSUnit is a unit testing framework for .NET applications. Annotations in MSUnit are attributes that can be applied to test methods or test classes to provide additional information or behavior to the unit testing framework.

Here are some of the commonly used annotations in MSUnit:

1. [TestMethod]: This annotation is used to mark a method as a test method that needs to be executed by the testing framework.

2. [TestClass]: This annotation is used to mark a class as a test class that contains one or more test methods.

3. [TestInitialize]: This annotation is used to mark a method that should be executed before each test method in the test class. This can be used to set up any resources or state required for the tests.

4. [TestCleanup]: This annotation is used to mark a method that should be executed after each test method in the test class. This can be used to clean up any resources or state that were used during the tests.

5. [TestCategory]: This annotation is used to categorize test methods or test classes into different categories. This can be used to group tests by their functionality, priority, or any other criteria.

6. [Ignore]: This annotation is used to ignore a test method or a test class from being executed by the testing framework. This can be used to temporarily disable tests that are failing or not yet implemented.

**DataDrivenTesting:**

Ex Programs:

[TestClass]

public class UnitTest1

{

[TestMethod]

[TestCategory("DataDriven")]

[Priority(1)]

[DataTestMethod]

[DataRow("https://www.facebook.com/", "Facebook")]

[DataRow("https://medium.com/", "Medium")]

[DataRow("https://www.bbc.com/", "BBC")]

public void DataRowAndDataDriven(String url, String eTitle)

{

IWebDriver driver = new OpenQA.Selenium.Chrome.ChromeDriver();

driver.Url = url;

String aTitle = driver.Title;

Console.WriteLine(aTitle);

try

{

Assert.IsTrue(aTitle.Contains(eTitle));

}

catch(Exception e)

{

Console.WriteLine(eTitle+" title not matching");

Console.WriteLine(e.StackTrace);

}

finally

{

driver.Close();

driver.Dispose();

Assert.IsTrue(aTitle.Contains(eTitle));

}

}

}

Ex2:

[TestClass]

public class UnitTest2

{

public static IEnumerable<object[]> Tests

{

get

{

return new[]

{

new object[] {1,1,2},

new object[] {1,3,3},

new object[] {2,2,4},

};

}

}

[TestMethod]

[TestCategory("DynamicData"),TestCategory("DataDriven")]

[DynamicData(nameof(Tests))]

public void TestMethod(int a, int b, int c)

{

int multiply =a\*b;

bool result = false;

if(multiply == c)

{

result = true;

}

Assert.IsTrue(result, "result not matching");

// MessageBox.Show(" " + c);

}

}

Ex3:

[TestClass]

public class UnitTest3

{

[TestMethod]

[TestCategory("DynamicData"), TestCategory("DataDriven")]

[DynamicData(nameof(Tests),DynamicDataSourceType.Property)]

public void TestMethod(int a, int b, int c)

{

int multiply = a \* b;

bool result = false;

if (multiply == c)

{

result = true;

}

Assert.IsTrue(result, "result not matching");

}

[TestMethod]

[TestCategory("DynamicData"), TestCategory("DataDriven")]

[DynamicData(nameof(numberData),DynamicDataSourceType.Method)]

public void UsingMethodDynamicTestDataMethod(int a, int b, int c)

{

int multiply = a + b;

bool result = false;

if (multiply == c)

{

result = true;

}

Assert.IsTrue(result, "result not matching");

}

public static IEnumerable<object[]> numberData()

{

yield return new object[] { 1, 1, 2 };

yield return new object[] { 1, 2, 3 };

yield return new object[] { 2, 2, 4 };

}

public static IEnumerable<object[]> Tests()

{

yield return new object[] { 1, 1, 2 };

yield return new object[] { 1, 2, 3 };

yield return new object[] { 2, 2, 4 };

}

}

Ex4: Data from excel sheet

[TestClass]

public class ExcelDataDriven

{

Spreadsheet sheet;

[TestMethod]

[TestCategory("Excel Data fetch")]

public void TestMethod1()

{

string data = sheet.Workbook.Worksheets.ByName("Sheet1").Cell(0, 0).ToString();

Console.WriteLine(data);

}

[TestMethod]

[TestCategory("multiple data")]

public void TestMethod2()

{

;

Worksheet sh = sheet.Workbook.Worksheets.ByName("Sheet1");

int rowNum = sh.UsedRangeRowMax;

int colNum = sh.UsedRangeColumnMax;

for (int i = 0; i <= rowNum; i++)

{

for (int j = 0; j <= colNum; j++)

{

string data = sh.Cell(i, j).ToString();

Console.Write(data + " ");

}

Console.WriteLine();

}

}

[TestInitialize]

public void TestInit()

{

sheet = new Spreadsheet();

sheet.LoadFromFile("C:\\Users\\panth\\OneDrive\\Documents\\Excel.xlsx");

}

[TestCleanup]

public void TestCleanup()

{

sheet.Dispose();

}

}