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);

}

}

**HashSet:**

1). In c#, Hashset is a collection type that provides a way to store and manipulate a set of unique values.

2). The values in Hashset are unordered and duplicates are not allowed. It automatically removes duplicates, so we can be sure that each item in the set is unique.

Common methods of ‘HashSet’:

\*Add: Add an item to the HashSet.

\*Remove: Remove an item from the HashSet.

\*Contains: Check whether the HashSet contains particular item or not.

\*Clear: Removes all the items from the HashSet.

\*Count: count number of items in the HashSet.

Ex:

internal class HashSet

{

public static void Main(String[] args)

{

var set = new HashSet<String>(); //create a new hashset of string

set.Add("first"); //add some strings to the Hashset

set.Add("second");

set.Add("third");

set.Add("snake");

set.Remove("first"); //remove a string from the hashset

foreach (var item in set) //iterating through hashset

{

Console.WriteLine(item); //printing the elements in hashset

}

if(set.Contains("third")) //contains method will check if the string is present in set or not

{

Console.WriteLine("second is present");

}

Console.WriteLine(set.Count+" number of items before clear");

set.Clear(); //clear the hashset

Console.WriteLine(set.Count+" number of items after clear"); //print the number of items present in the Hashset

} //end of main method

}

**SortedSet:**

1. SortedSet is a collection type that provides a way to store and manipulate a unique set of values that are sorted in ascending order.
2. It automatically removes duplicates, so we can be sure that each item in the set is unique.
3. It maintains the sort order of the elements, which means that you can easily retrieve the smallest or largest element in the set.

Common methods of SortedSet:

Add: Adds an item to the set.

Remove: Removes an item from the set.

Contains: checks whether the item is present in the set or not.

Clear:Removes all the items from the set.

Count: Gets the number of items in the SortedSet.

Ex:

internal class SortedSet

{

public static void Main(string[] args)

{

var hashset =new HashSet<string>(); //creating a SortedSet of type String

hashset.Add("first"); //adding items to set

hashset.Add("second");

hashset.Add("third");

hashset.Add("fourth");

hashset.Add("second"); //adding a duplicate item

hashset.Remove("first"); //removing a item from the set

foreach (var item in hashset)

{

Console.WriteLine(item);

}

if(hashset.Contains("first") ) //check whether the particular item is present or not

{

Console.WriteLine("first item present");

}

else

{

Console.WriteLine("first item not present");

}

hashset.Clear(); //Removing all the items from set

hashset.Add("fifth");

hashset.Add("second");

foreach(string s in hashset)

{

Console.WriteLine(s); //printing the items in the set

}

}

}

**Dictionary<K, V>:**

In c#, Dictionary<K, V> is collection type that provides a way to store and manipulate a set of key-value pairs. The ‘K’ and ‘V’ in dictionary<K, V> are the types of the key and values.

It automatically removes duplicate keys, so each key in the dictionary is unique.

It provides methods for performing set-like operations on the keys, such as ‘Keys’ and ‘Values’, which can be useful for comparing and combining dictionaries.

Common methods of Dictionary<K, V>:

Add: Adds the key-value pair to the dictionary.

Remove: Removes key-value pair from the dictionary.

Contains(K key): check whether the particular dictionary contains the specified key.

Clear: Removed all the key-value pairs in the dictionary.

Count: Gets the number of key-value pairs in the dictionary.

Keys: Gets a collection of all keys in the dictionary.

Values: Gets a collection of all values in the dictionary.

Ex:

internal class Dictionary

{

public static void Main(string[] args)

{

var dictionary = new Dictionary<String, int>(); // creates a new dictionary type of String

dictionary.Add("animal", 3); //adds key-value pairs to the dictionary

dictionary.Add("basket", 1);

dictionary.Add("cricket", 2);

dictionary.Add("dodge", 5);

dictionary.Remove("dodge"); //remove a particular key-value pair from the dictionary

foreach(var item in dictionary)

{

Console.WriteLine(item.Key);

}

if(dictionary.ContainsKey("dodge")) //containsKey used to check whether the key is present in the dictionary or not

{

Console.WriteLine("Key present");

}

else

{

Console.WriteLine("key not present");

}

foreach (var item in dictionary)

{

Console.Write(item.Key+" "); //key method is used to take the keys in the dictionary

Console.Write(item.Value); //value method is used to take the values in the dictionary

Console.WriteLine();

}

Console.WriteLine(dictionary.Count+" count"); //print the number of key-value pairs present in the dictionary

dictionary.Clear(); //clears all the key-value pairs in the dictionary

}

}

**SortedDictionary<K, V> :**

In c#, SortedDictionary<K, V> is a collection type that provides a way to store and manipulate a set of key-value pairs in the sorted order based on the keys. It automatically removes the duplicates.

Common methods of SortedDictionary<K, V>:

Add<Key, Value>: Adds a key-value pair to the dictionary.

Remove<Key, Value>: Removes a key-value pair from the dictionary.

Contains<Key>: check whether the particular dictionary contains the specified key.

Clear: Removes all the key-value pairs from the dictionary.

Count: Counts the number of key-value pairs in the dictionary.

Keys: Gets a collection of all keys in the dictionary.

Values: Gets a collection of all values from the dictionary.

Ex:

internal class SortedDictionary

{

public static void Main(string[] args)

{

var dictionary = new SortedDictionary<String, int>();

dictionary.Add("animal", 3); //adds key-value pairs to the dictionary

dictionary.Add("dodge", 5);

dictionary.Add("cricket", 2);

dictionary.Add("basket", 1);

dictionary.Remove("dodge"); //remove a particular key-value pair from the dictionary

foreach (var item in dictionary)

{

Console.WriteLine(item.Key);

}

if (dictionary.ContainsKey("dodge")) //containsKey used to check whether the key is present in the dictionary or not

{

Console.WriteLine("Key present");

}

else

{

Console.WriteLine("key not present");

}

foreach (var item in dictionary)

{

Console.Write(item.Key + " "); //key method is used to take the keys in the dictionary

Console.Write(item.Value); //value method is used to take the values in the dictionary

Console.WriteLine();

}

Console.WriteLine(dictionary.Count + " count"); //print the number of key-value pairs present in the dictionary

dictionary.Clear(); //clears all the key-value pairs in the dictionary

}

}

**Stack**:

In collection, stack is a collection type that represents a last-in, first-out (LIFO) stack of objects.

It automatically maintains the order of items in a LIFO stack, so the last item added to the stack is the first item to be removed.

Common methods of Stack:

Push(): Add an item to the top of the stack.

Pop(): Removes and returns the item at the top of the stack.

Peek(): Returns the item at the top of the stack without removing it.

Clear(); Removes all the items from the stack.

Contains(): checks whether the stack contains the specified item.

Count: Gets the number of items in the stack.

Ex:

internal class Stack

{

public static void Main(string[] args)

{

// Create a new stack of integers

Stack<int> numbers = new Stack<int>();

// Push some numbers onto the stack

numbers.Push(1);

numbers.Push(2);

numbers.Push(3);

// Print the number of items on the stack

Console.WriteLine("The stack contains {0} items.", numbers.Count);

// Pop items off the stack in reverse order

Console.WriteLine("Popping items off the stack:");

while (numbers.Count > 0)

{

int number = numbers.Pop();

Console.WriteLine(number);

}

// Check if the stack is empty

if (numbers.Count == 0)

{

Console.WriteLine("The stack is now empty.");

}

else

{

Console.WriteLine("The stack is not empty.");

}

}

}

**Queue**:

In collection, stack is a collection type that represents a first-in, first-out (LIFO) stack of objects.

Common methods in Queue:

Enqueue(): Adds an item to the end of the queue.

Dequeue(): Removes and returns the item at the beginning of the queue.

Peek(): Returns the item at the beginning of the queue without removing it.

Count(): Returns the number of items in the queue.

Contains(): Returns the Boolean value indicating whether an item is in the queue.

Ex:

internal class Queue

{

public static void Main(string[] args)

{

Queue<string> myQueue = new Queue<string>();

myQueue.Enqueue("item1");

myQueue.Enqueue("item2");

myQueue.Enqueue("item3");

Console.WriteLine("Items in queue:");

foreach (string item in myQueue)

{

Console.WriteLine(item);

}

string firstItem = myQueue.Dequeue();

Console.WriteLine("First item removed from queue: " + firstItem);

string nextItem = myQueue.Peek();

Console.WriteLine("Next item in queue: " + nextItem);

Console.WriteLine("Items in queue after removal:");

foreach (string item in myQueue)

{

Console.WriteLine(item);

}

}

}

**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 states 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 states 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();

}

}

**LoginPage Automation(actiTime):**

POM page:

using OpenQA.Selenium;

using OpenQA.Selenium.Support.PageObjects;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace UnitTestProject1.ObjectRepository.ActitimeLogin

{

public class LoginPageElements

{

//DECLARATION

[FindsBy(How = How.Name, Using = "username")]

private IWebElement username { get; set; }

[FindsBy(How =How.Name, Using = "pwd")]

private IWebElement password { get; set; }

[FindsBy(How =How.ClassName, Using = "initial")]

private IWebElement loginBtn { get; set; }

//initialization

public LoginPageElements(IWebDriver driver)

{

PageFactory.InitElements(driver, this);

}

//utilization

public void Login(string Username, string Password)

{

username.SendKeys(Username);

password.SendKeys(Password);

loginBtn.Click();

}

}

}

ExcelUtility:

using Bytescout.Spreadsheet;

using Bytescout.Spreadsheet.COM;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace UnitTestProject1.GenericUtilities

{

public class ExcelUtilities

{

public string excelData(int rowNo, int columnNo)

{

Spreadsheet sheet = new Spreadsheet();

sheet.LoadFromFile("D:\\C#\\Excel\\Workbook1.xlsx");

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

string data = sh.Cell(rowNo, columnNo).ToString();

return data;

}

}

WebDriverUtility:

using OpenQA.Selenium;

using OpenQA.Selenium.Support.UI;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace UnitTestProject1.GenericUtilities

{

public class IWebDriverUtilities

{

public void ImplicitlyWait(IWebDriver driver, long time)

{

driver.Manage().Timeouts().ImplicitWait=TimeSpan.FromSeconds(time);

}

public void ExplicitWait(IWebDriver driver, long time)

{

WebDriverWait wait = new WebDriverWait(driver, TimeSpan.FromSeconds(time));

// wait.Until(ExpectedConditions.ElementIsVisible(By.Id("logoutLink")));

wait.Until(ExpectedConditions.TitleContains("actiTIME - Enter Time-Track"));

}

public void MaximizeWindow(IWebDriver driver)

{

driver.Manage().Window.Maximize();

}

}

}

TestClass:

using Microsoft.VisualStudio.TestTools.UnitTesting;

using OpenQA.Selenium;

using OpenQA.Selenium.Chrome;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

using UnitTestProject1.GenericUtilities;

using UnitTestProject1.ObjectRepository.ActitimeLogin;

namespace UnitTestProject1.ObjectRepository.Tests.LoginPage

{

[TestClass]

public class LoginPageTest

{

IWebDriver driver;

IWebDriverUtilities webDriverUtilities = new IWebDriverUtilities(); //initializing the webdriver utilities

ExcelUtilities excelUtilities = new ExcelUtilities();

String eTitle = "actiTIME - Enter Time-Track"; //expected title

[TestInitialize]

public void InitializaTest() //test initializer method

{

driver = new ChromeDriver(); //opening the browser

}

[TestCleanup]

public void TestCleanupMethod()

{

driver.Close(); //closing the browser

driver.Dispose(); //clean

}

[TestMethod]

[TestCategory("LoginPage")]

public void VerificationOfTest()

{

LoginPageElements loginPageElements = new LoginPageElements(driver);

webDriverUtilities.ImplicitlyWait(driver, 10); //implicit wait

webDriverUtilities.MaximizeWindow(driver);

String Url = excelUtilities.excelData(0, 1); //fetching url from excelsheet

driver.Navigate().GoToUrl(Url); //go to Url actiTIME

// driver.Url = "Url";

String Username = excelUtilities.excelData(1, 1); //fetching username from excelsheet

String Password = excelUtilities.excelData(2, 1); //fetching password from excel sheet

loginPageElements.Login(Username, Password); //Login method to login to actiTIME

// Thread.Sleep(6000);

webDriverUtilities.ExplicitWait(driver, 10); //explicit wait

String aTitle = driver.Title;

Console.WriteLine(aTitle);

Console.WriteLine(eTitle);

Assert.AreEqual(eTitle,aTitle); //assert statement to compare the expected and actual title

}

}

}