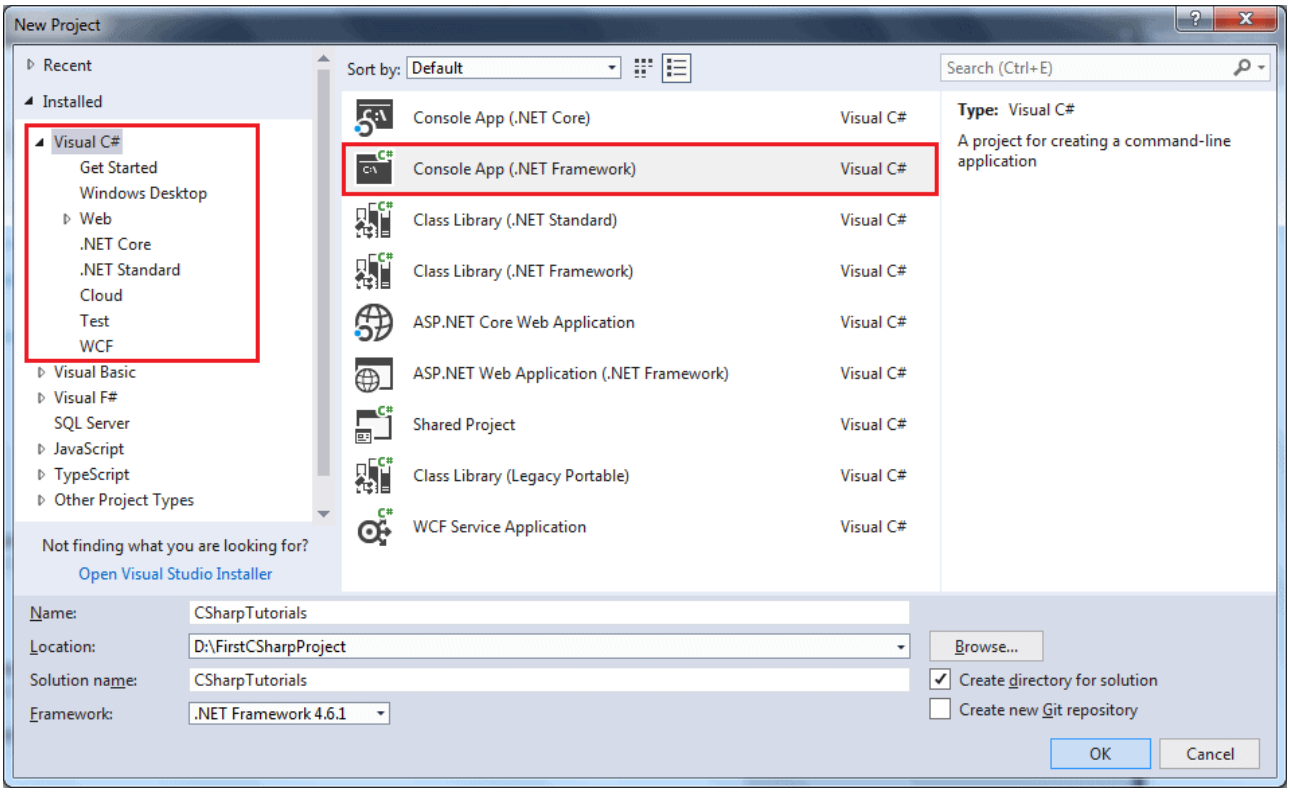
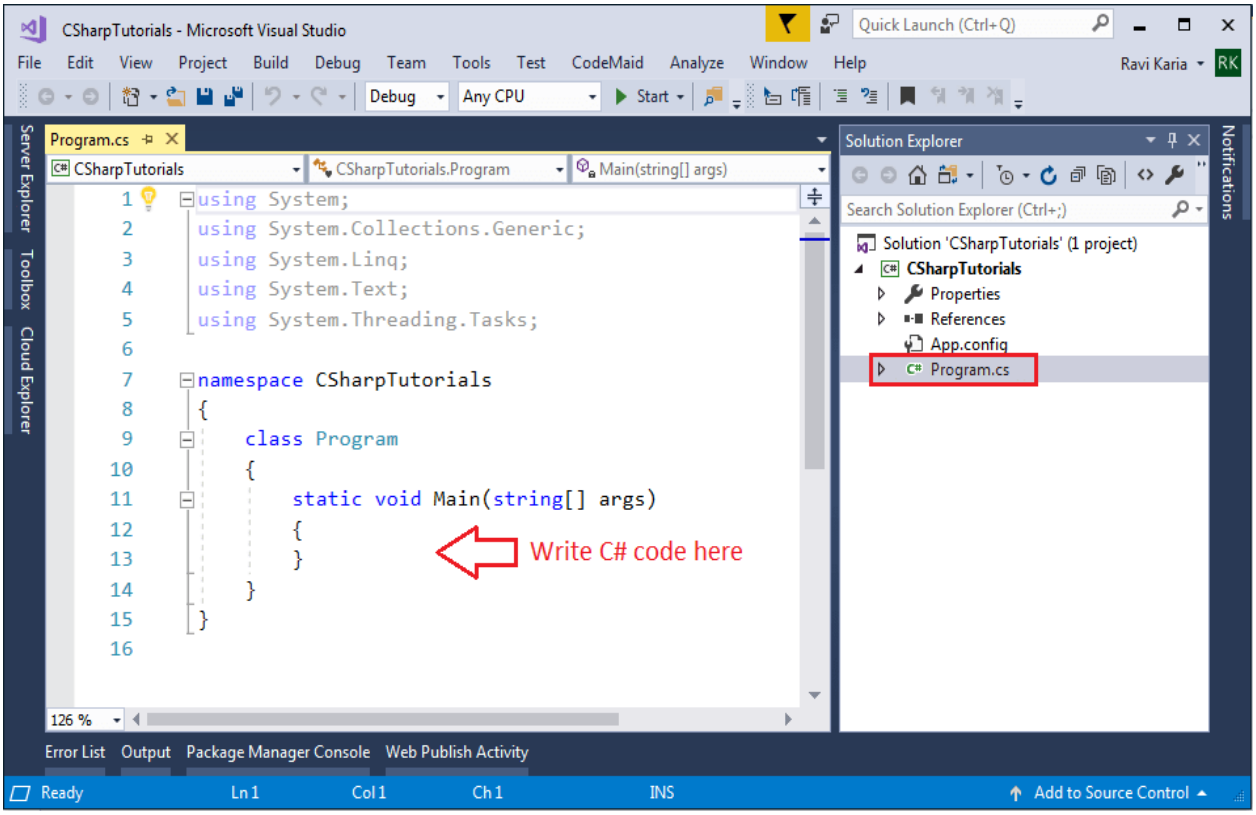
From the **New Project** popup, shown below, select Visual C# in the left side panel and select the Console App in the right-side panel.



Select Visual C# Console App Template

In the name section, give any appropriate project name, a location where you want to create all the project files, and the name of the project solution.

Click OK to create the console project. **Program.cs** will be created as default a C# file in Visual Studio where you can write your C# code in Program class, as shown below. (The .cs is a file extension for C# file.)



C# Console Program

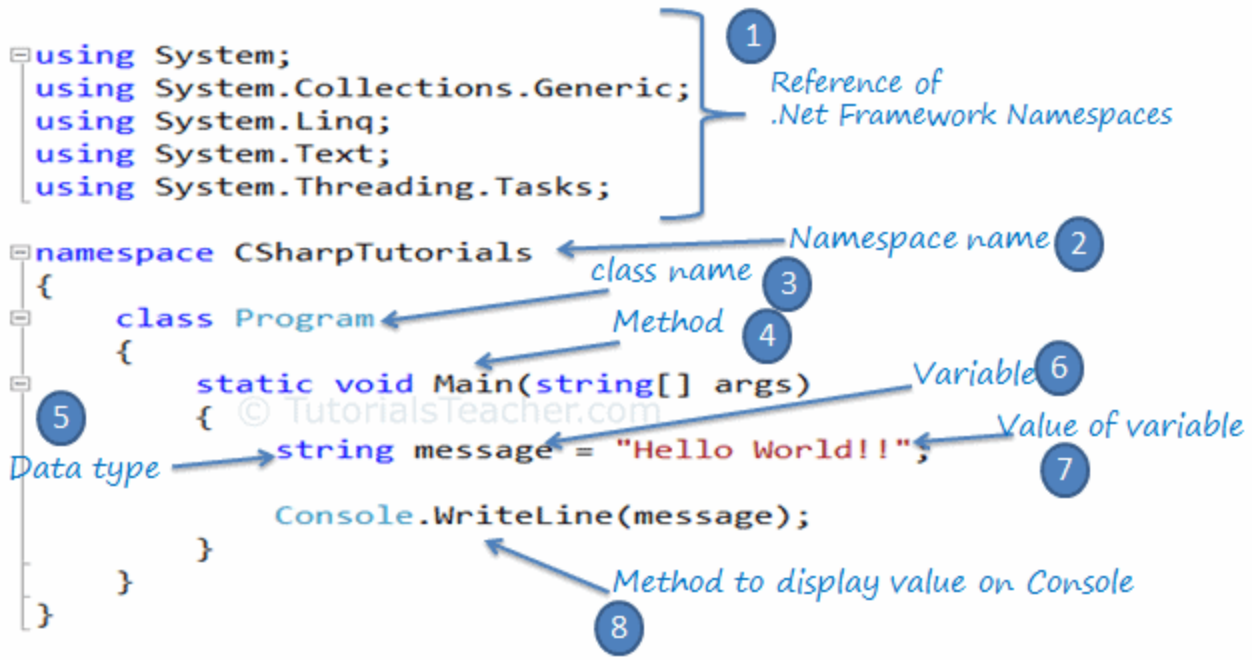
Every console application starts from the `Main()` method of the `Program` class. The following example displays "Hello World!!" on the console.

Example: C# Console Application

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CSharpTutorials
{
    class Program
    {
        static void Main(string[] args)
        {
            string message = "Hello World!!";

            Console.WriteLine(message);
        }
    }
}
```

Try it

The following image illustrates the important parts of the above example.



C# Code Structure

Let's understand the above C# structure.

1. Every .NET application takes the reference of the necessary .NET framework namespaces that it is planning to use with the `using` keyword, e.g., `using System.Text`.
2. Declare the namespace for the current class using the `namespace` keyword, e.g., `namespace CSharpTutorials.FirstProgram`
3. We then declared a class using the `class` keyword: `class Program`
4. The `Main()` is a method of `Program` class is the entry point of the console application.
5. `String` is a data type.
6. A `message` is a variable that holds the value of a specified data type.
7. `"Hello World!!"` is the value of the message variable.
8. The `Console.WriteLine()` is a static method, which is used to display a text on the console.

Note:

Every line or statement in C# must end with a semicolon (;).

Compile and Run C# Program

To see the output of the above C# program, we have to compile it and run it by pressing Ctrl + F5 or clicking the Run button or by clicking the "Debug" menu and clicking "Start Without Debugging". You will see the following output in the console:

Output:

```
Hello World!!
```

So this is the basic code items that you will probably use in every C# code.

C# - Data Types

C# is a strongly-typed language. It means we must declare the type of a variable that indicates the kind of values it is going to store, such as integer, float, decimal, text, etc.

The following declares and initialized variables of different data types.

```
string stringVar = "Hello World!!";
int intVar = 100;
float floatVar = 10.2;
char charVar = 'A';
bool boolVar = true;
var variabledata
dynamic dynamic_data
decimal d1 = 12345678912345678912345678 9.5
string text = "This is a string.";
DateTime dt1 = new DateTime();

//assigns year, month, day
DateTime dt2 = new DateTime(2015, 12, 31);
```

var type of variables are required to be initialized at the time of declaration or else they encounter the compile time error: Implicitly-typed local variables must be initialized.

dynamic type variables need not be initialized when declared

C# includes escaping character \ (backslash) before these special characters to include in a string.

Use backslash \ before double quotes and some special characters such as \,\n,\r,\t, etc. to include it in a string.

```
string text = "This is a \"string\" in C#.";
string str = "xyzdef\\rabc";
string path = "\\\\mypc\\ shared\\project";
```

Verbatim Strings

It is tedious to prefix \ to include every special characters. Verbatim string in C# allows a special characters and line brakes. Verbatim string can be created by prefixing @ symbol before double quotes.

```
string str = @"xyzdef\rabc";
string path = @"\\mypc\shared\project";
string email = @"test@test.com";
```

The @ symbol can also be used to declare a multi-line string.

```
string str1 = "this is a \n" +
```

```
        "multi line \n" +
        "string";

// Verbatim string
string str2 = @"this is a
        multi line
        string";
```

String Concatenation

Multiple strings can be concatenated with + operator.

```
string name = "Mr." + "James " + "Bond" + ", Code: 007";

string firstName = "James";
string lastName = "Bond";
string code = "007";

string agent = "Mr." + firstName + " " + lastName + ", Code: " + code;
```

String Interpolation

String interpolation is a better way of concatenating strings. We use + sign to concatenate string variables with static strings.

C# 6 includes a special character $ to identify an interpolated string. An interpolated string is a mixture of static string and string variable where string variables should be in {} brackets.

```
string firstName = "James";
string lastName = "Bond";
string code = "007";

string fullName = $"Mr. {firstName} {lastName}, Code: {code}";
```

Comparison Operators

Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Logical Operators

Operator	Name	Description	Example
&&	Logical and	Returns True if both statements are true	x < 5 && x < 10
||	Logical or	Returns True if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns False if the result is true	!(x < 5 && x < 10)

Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations:

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	x++
--	Decrement	Decreases the value of a variable by 1	x--

Conditional Statements

C# has the following conditional statements:

- Use **if** to specify a block of code to be executed, if a specified condition is true
- Use **else** to specify a block of code to be executed, if the same condition is false
- Use **else if** to specify a new condition to test, if the first condition is false
- Use **switch** to specify many alternative blocks of code to be executed

Example:

```
int x = 20;
int y = 18;
if (x > y)
{
  Console.WriteLine("x is greater than y");
}
```

Example

```
int time = 20;
if (time < 18)
{
  Console.WriteLine("Good day.");
```

```
}
else
{
  Console.WriteLine("Good evening.");
}
// Outputs "Good evening."
```

Example

```
int time = 22;
if (time < 10)
{
  Console.WriteLine("Good morning.");
}
else if (time < 20)
{
  Console.WriteLine("Good day.");
}
else
{
  Console.WriteLine("Good evening.");
}
```

Example:

```
int time = 20;
string result = (time < 18) ? "Good day." : "Good evening.";
Console.WriteLine(result);
```

Switch Statement

```
switch (day)
{
  case 6:
    Console.WriteLine("Today is Saturday.");
    break;
  case 7:
    Console.WriteLine("Today is Sunday.");
    break;
  default:
    Console.WriteLine("Looking forward to the Weekend.");
    break;
}
```

Loops

There are four different types of loops in C#, While loop, Do While Loop, For Loop, Foreach Loop.

While Loop

```
int i = 0;
while (i < 5)
{
  Console.WriteLine(i);
  i++;
}
```

Do While loop

```
int i = 0;
do
{
  Console.WriteLine(i);
  i++;
}
while (i < 5);
```

For Loop

```
for (int i = 0; i < 5; i++)
{
  Console.WriteLine(i);
}
```

Foreach loop

There is also a foreach loop, which is used exclusively to loop through elements in an **array**:

```
foreach (type variableName in arrayName)
{
  // code block to be executed
}
```

Example

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
foreach (string i in cars)
{
  Console.WriteLine(i);
}
```

C# Class

A class is like a blueprint of a specific object. In the real world, every object has some color, shape, and functionalities - for example, the luxury car Ferrari. Ferrari is an object of the luxury car type. The luxury car is a class that indicates some characteristics like speed, color, shape, interior, etc. So any company that makes a car that meets those requirements is an object of the luxury car type. For example, every single car of BMW, Lamborghini, Cadillac are an object of the class called 'Luxury Car'. Here, 'Luxury Car' is a class, and every single physical car is an object of the luxury car class.

Likewise, in object-oriented programming, a class defines some properties, fields, events, methods, etc. A class defines the kinds of data and the functionality their objects will have.

A class enables you to create your custom types by grouping variables of other types, methods, and events.

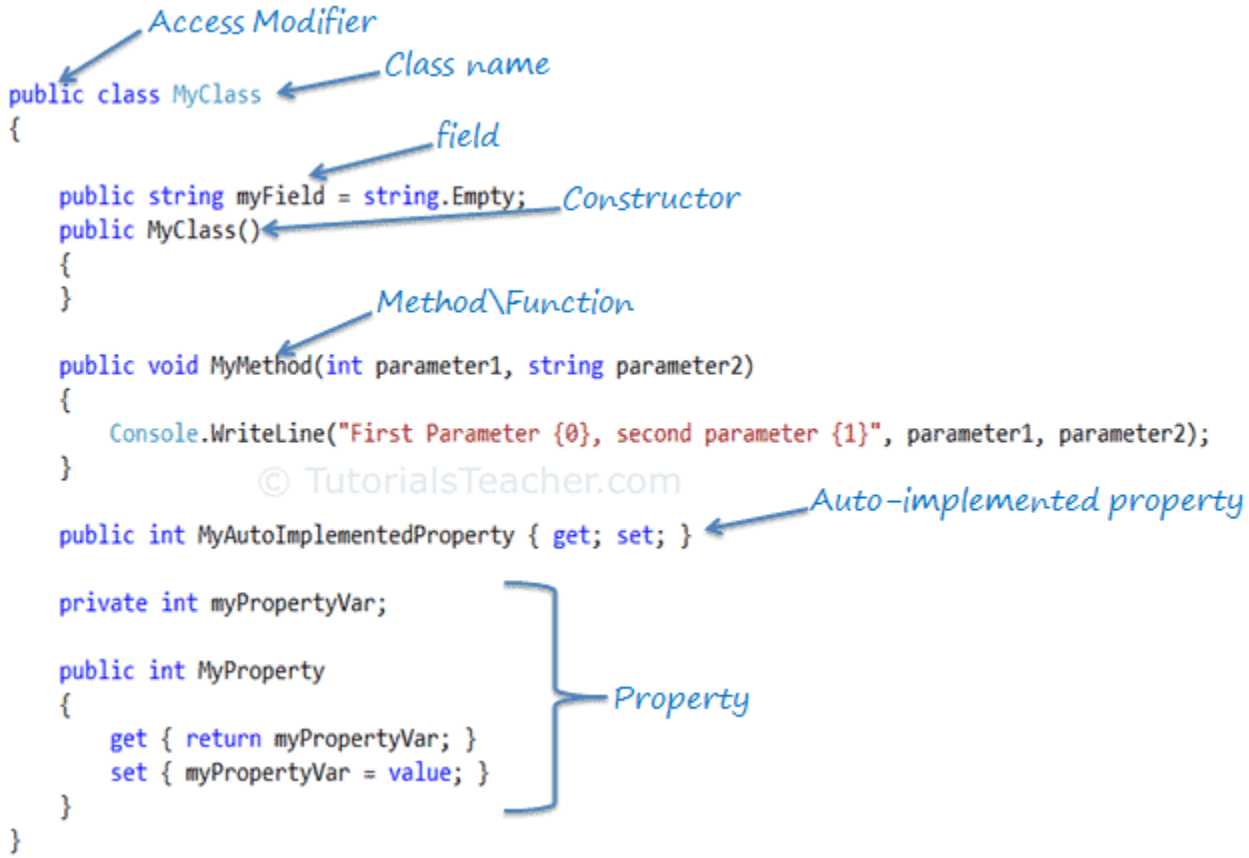
In C#, a class can be defined by using the class keyword.

```
Example:
public class MyClass
{
        Public/private Datatype Datamember1 {get;set;}
}
```

```
Example:

public class MyClass
{
    public string myField = string.Empty;

    public MyClass()
    {
    }

    public void MyMethod(int parameter1, string parameter2)
    {
        Console.WriteLine("First Parameter {0}, second parameter {1}",
                                                    parameter1, parameter2);
    }

    public int MyAutoImplementedProperty { get; set; }

    private int myPropertyVar;

    public int MyProperty
    {
```

```
        get { return myPropertyVar; }
        set { myPropertyVar = value; }
    }
}
```

The following image shows the important building blocks of C# class.



C# Class

C# Access Modifiers

Access modifiers are applied to the declaration of the class, method, properties, fields, and other members. They define the accessibility of the class and its members. Public, private, protected, and internal are access modifiers in C#.

Access Modifier Keywords:

Access modifiers are applied to the declaration of the class, method, properties, fields, and other members. They define the accessibility of the class and its members.

Access Modifiers	Usage
public	The Public modifier allows any part of the program in the same assembly or another assembly to access the type and its members.

Access Modifiers	Usage
private	The Private modifier restricts other parts of the program from accessing the type and its members. Only code in the same class or struct can access it.
internal	The Internal modifier allows other program code in the same assembly to access the type or its members. This is default access modifiers if no modifier is specified.
protected	The Protected modifier allows codes in the same class or a class that derives from that class to access the type or its members.

C# Field

The field is a class-level variable that holds a value. Generally, field members should have a private access modifier and used with property.

C# Enumerations Type - Enum

In C#, an **enum** (or enumeration type) is used to assign constant names to a group of numeric integer values. It makes constant values more readable, for example, **WeekDays.Monday** is more readable then number 0 when referring to the day in a week.

An enum is defined using the enum keyword, directly inside a namespace, class, or structure. All the constant names can be declared inside the curly brackets and separated by a comma. The following defines an enum for the weekdays.

```
enum WeekDays
{
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}
```

Enum Values

If values are not assigned to **enum** members, then the compiler will assign integer values to each member starting with zero by default. The first member of an enum will be 0, and the value of each successive enum member is increased by 1.

```
enum Categories
{
    Electronics = 1,
    Food = 5,
    Automotive = 6,
    Arts = 10,
    BeautyCare = 11,
    Fashion = 15,
    WomanFashion = 15
}
```

C# interface

An interface is a completely "**abstract class**", which can only contain abstract methods and properties (with empty bodies):

It is considered good practice to start with the letter "I" at the beginning of an interface, as it makes it easier for yourself and others to remember that it is an interface and not a class.

By default, members of an interface are abstract and public.

Interfaces can contain properties and methods, but not fields.

Example:

```
interface IAnimal // Interface class
{
  void animalSound(); // interface method (does not have a body)
}

// Pig "implements" the IAnimal interface
class Cat : IAnimal
{
  public void animalSound()
  {
    // The body of animalSound() is provided here
    Console.WriteLine("The Cat says: meow meow");
  }
}

class Program
{
  static void Main(string[] args)
  {
    Cat myCat = new Cat();  // Create a Pig object
    myCat.animalSound();
  }
}
```

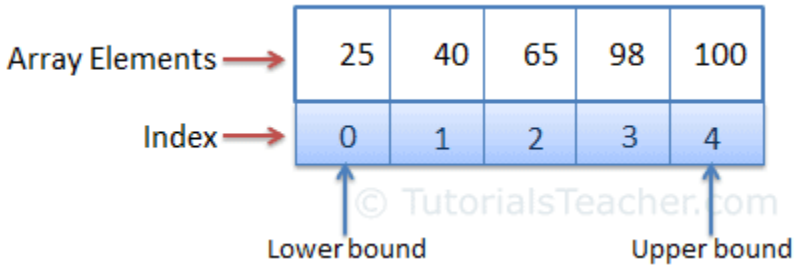
C# Arrays

A variable is used to store a literal value, whereas an array is used to store multiple literal values.

An array is the data structure that stores a fixed number of literal values (elements) of the same data type. Array elements are stored contiguously in the memory.

In C#, an array can be of three types: single-dimensional, multidimensional, and jagged array. Here you will learn about the single-dimensional array.

The following figure illustrates an array representation.



Array Representation

Array Declaration and Initialization

An array can be declared using by specifying the type of its elements with square brackets.

Example: Array Declaration

```
int[] evenNums;  // integer array

string[] cities; // string array

int[] evenNums = new int[5]{ 2, 4, 6, 8, 10 };

string[] cities = new string[3]{ "Islamabad", "Krachi", "Lahore" };
```

C# Jagged Arrays: An Array of Array

A jagged array is an array of array. Jagged arrays store arrays instead of literal values.

A jagged array is initialized with two square brackets [][]. The first bracket specifies the size of an array, and the second bracket specifies the dimensions of the array which is going to be stored.

The following example declares jagged arrays.

```
int[][] jArray1 = new int[2][]; // can include two single-dimensional arrays
int[][,] jArray2 = new int[3][,]; // can include three two-dimensional arrays
```

Example:

```
int[][] jArray = new int[2][];

jArray[0] = new int[3]{1, 2, 3};

jArray[1] = new int[4]{4, 5, 6, 7 };
```

Example:

```
int[][] jArray = new int[2][]{
                new int[3]{1, 2, 3},

                new int[4]{4, 5, 6, 7}
            };

jArray[0][0]; //returns 1
jArray[0][1]; //returns 2
jArray[0][2]; //returns 3
jArray[1][0]; //returns 4
jArray[1][1]; //returns 5
jArray[1][2]; //returns 6
jArray[1][3]; //returns 7
```

Example:

```
int[][] jArray = new int[2][]{
                new int[3]{1, 2, 3},

                new int[4]{4, 5, 6, 7}
            };

for(int i=0; i<jArray.Length; i++)
{
     for(int j=0; j < (jArray[i]).Length; j++)
            Console.WriteLine(jArray[i][j]);
}
```

C# - List<T>

The **List<T>** is a collection of strongly typed objects that can be accessed by index and having methods for sorting, searching, and modifying list. It is the generic version of the **ArrayList** that comes under **System.Collections.Generic** namespace.

List<T> Characteristics

- List<T> equivalent of the ArrayList, which implements IList<T>.
- It comes under System.Collections.Generic namespace.

- List<T> can contain elements of the specified type. It provides compile-time type checking and doesn't perform boxing-unboxing because it is generic.
- Elements can be added using the Add(), AddRange() methods or collection-initializer syntax.
- Elements can be accessed by passing an index e.g. myList[0]. Indexes start from zero.
- List<T> performs faster and less error-prone than the ArrayList.

Creating a List

The **List<T>** is a generic collection, so you need to specify a type parameter for the type of data it can store. The following example shows how to create list and add elements.

Example:

```
List<int> primeNumbers = new List<int>();
primeNumbers.Add(1); // adding elements using add() method
primeNumbers.Add(3);
primeNumbers.Add(5);
primeNumbers.Add(7);

var cities = new List<string>();
cities.Add("New York");
cities.Add("London");
cities.Add("Lahore");
cities.Add("London");
cities.Add(null);// nulls are allowed for reference type list

//adding elements using collection-initializer syntax
var bigCities = new List<string>()
                    {
                        "New York",
                        "London",
                        "Krachi",
                        "Tokyo"
                    };
```

Example:

```
string[] cities = new string[3]{ "Lahore", "London", "New York" };

var popularCities = new List<string>();

// adding an array in a List
popularCities.AddRange(cities);

var favouriteCities = new List<string>();

// adding a List
```

```
favouriteCities.AddRange(popularCities);
```

C# - Dictionary<TKey, TValue>

The **Dictionary<TKey, TValue>** is a generic collection that stores key-value pairs in no particular order.

Dictionary Characteristics

- Dictionary<TKey, TValue> stores key-value pairs.
- Comes under System.Collections.Generic namespace.
- Implements IDictionary<TKey, TValue> interface.
- Keys must be unique and cannot be null.
- Values can be null or duplicate.
- Values can be accessed by passing associated key in the indexer e.g. myDictionary[key]
- Elements are stored as KeyValuePair<TKey, TValue> objects.

Creating a Dictionary

You can create the **Dictionary<TKey, TValue>** object by passing the type of keys and values it can store. The following example shows how to create a dictionary and add key-value pairs.

Example:

```
IDictionary<int, string> numberNames = new Dictionary<int, string>();
numberNames.Add(1,"One"); //adding a key/value using the Add() method
numberNames.Add(2,"Two");
numberNames.Add(3,"Three");

//The following throws run-time exception: key already added.
//numberNames.Add(3, "Three");

foreach(KeyValuePair<int, string> kvp in numberNames)
    Console.WriteLine("Key: {0}, Value: {1}", kvp.Key, kvp.Value);

//creating a dictionary using collection-initializer syntax
var cities = new Dictionary<string, string>(){
     {"UK", "London, Manchester, Birmingham"},
     {"USA", "Chicago, New York, Washington"},
     {"India", "Mumbai, New Delhi, Pune"}
};

foreach(var kvp in cities)
    Console.WriteLine("Key: {0}, Value: {1}", kvp.Key, kvp.Value);
```

C# - Stack<T>

Stack is a special type of collection that stores elements in LIFO style (Last In First Out). C# includes the generic **Stack<T>** and non-generic Stack collection classes. It is recommended to use the generic **Stack<T>** collection.

Stack is useful to store temporary data in LIFO style, and you might want to delete an element after retrieving its value.

Stack<T> Characteristics

- `Stack<T>` is Last In First Out collection.
- It comes under `System.Collection.Generic` namespace.
- `Stack<T>` can contain elements of the specified type. It provides compile-time type checking and doesn't perform boxing-unboxing because it is generic.
- Elements can be added using the `Push()` method. Cannot use collection-initializer syntax.
- Elements can be retrieved using the `Pop()` and the `Peek()` methods. It does not support an indexer.

Creating a Stack

You can create an object of the `Stack<T>` by specifying a type parameter for the type of elements it can store. The following example creates and adds elements in the `Stack<T>` using the `Push()` method. Stack allows null (for reference types) and duplicate values.

Example

```
Stack<int> myStack = new Stack<int>();
myStack.Push(1);
myStack.Push(2);
myStack.Push(3);
myStack.Push(4);

foreach (var item in myStack)
     Console.Write(item + ","); //prints 4,3,2,1,

int[] arr = new int[]{ 1, 2, 3, 4};
Stack<int> myStack = new Stack<int>(arr);

foreach (var item in myStack)
     Console.Write(item + ","); //prints 4,3,2,1,
```

C# - Queue<T>

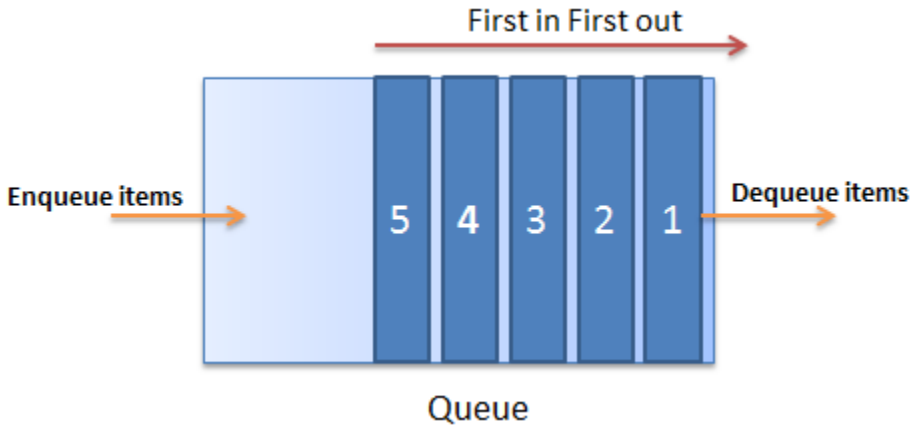
Queue is a special type of collection that stores the elements in FIFO style (First In First Out), exactly opposite of the Stack<T> collection. It contains the elements in the order they were added.

C# includes generic Queue<T> and non-generic Queue collection. It is recommended to use the generic Queue<T> collection.

Queue<T> Characteristics

- Queue<T> is FIFO (First In First Out) collection.
- It comes under System.Collection.Generic namespace.
- Queue<T> can contain elements of the specified type. It provides compile-time type checking and doesn't perform boxing-unboxing because it is generic.
- Elements can be added using the Enqueue() method. Cannot use collection-initializer syntax.
- Elements can be retrieved using the Dequeue() and the Peek() methods. It does not support an indexer.

The following figure illustrates the Queue collection:



Creating a Queue

You can create an object of the Queue<T> by specifying a type parameter for the type of elements it can store. The following example creates and adds elements in the Queue<T> using the Enqueue() method. A Queue collection allows null (for reference types) and duplicate values.

Example: Create and Add Elements in the Queue

```
Queue<int> callerIds = new Queue<int>();
callerIds.Enqueue(1);
callerIds.Enqueue(2);
callerIds.Enqueue(3);
callerIds.Enqueue(4);

foreach(var id in callerIds)
    Console.Write(id); //prints 1234
```

Queue<T> Properties and Methods

Property	Usage
Count	Returns the total count of elements in the Queue.

Method	Usage
Enqueue(T)	Adds an item into the queue.
Dequeue	Returns an item from the beginning of the queue and removes it from the queue.
Peek()	Returns an first item from the queue without removing it.
Contains(T)	Checks whether an item is in the queue or not
Clear()	Removes all the items from the queue.

Retrieve Elements from a Queue

The Dequeue() and the Peek() method is used to retrieve the first element in a queue collection. The Dequeue() removes and returns the first element from a queue because the queue stores elements in FIFO order. Calling the Dequeue() method on an empty queue will throw the InvalidOperation exception. So, always check that the total count of a queue is greater than zero before calling it.

Example:

```
Queue<string> strQ = new Queue<string>();
strQ.Enqueue("H");
strQ.Enqueue("e");
strQ.Enqueue("l");
strQ.Enqueue("l");
strQ.Enqueue("o");

Console.WriteLine("Total elements: {0}", strQ.Count); //prints 5

while (strQ.Count > 0)
```

```
    Console.WriteLine(strQ.Dequeue()); //prints Hello

Console.WriteLine("Total elements: {0}", strQ.Count); //prints 0
```