

# **SI-342(ENTERPRISE SYSTEM)**

## **.NET Framework in Enterprise Systems:**

The .NET Framework is like a toolbox for building software applications. It provides a set of tools and libraries that developers use to create various types of applications, from web services to desktop programs.

In enterprise systems, which are large-scale software applications used by businesses or organizations, the .NET Framework plays a crucial role. Here's why it's important:

1. **Consistency and Compatibility:** It ensures that different parts of a large system can work together smoothly. Developers can write code in different programming languages (like C#, VB.NET, etc.) that all run on the .NET Framework.
2. **Security:** The framework includes security features that help protect sensitive data and prevent unauthorized access.
3. **Performance:** It optimizes how applications run, making them faster and more efficient, which is critical for handling large amounts of data and many users simultaneously.
4. **Integration:** It allows enterprise systems to integrate with other software and systems, like databases or external services, making it easier to manage and share information.
5. **Scalability:** Enterprise systems often need to grow as businesses grow. The .NET Framework supports scalability, meaning it can handle increased workloads and users without major changes to the underlying codebase.

## **Working of .NET Framework:**

1. **Common Language Runtime (CLR):**
  - The CLR is the heart of the .NET Framework. It's like a virtual machine that manages the execution of .NET programs.
  - When a .NET application is run, the CLR compiles the intermediate language (IL) code into native code that the computer's processor can understand. This process is called Just-In-Time (JIT) compilation.
  - The CLR also handles memory management, exception handling, and security enforcement.
2. **Base Class Library (BCL):**
  - The BCL provides a rich set of functionalities that developers can use to build applications.
  - It includes classes for common tasks like file I/O, networking, database access, cryptography, and more.
  - This library saves developers from reinventing the wheel and ensures consistency across different applications.
3. **Language Independence:**

- One of the key strengths of the .NET Framework is its support for multiple programming languages.
  - Developers can write code in languages such as C#, Visual Basic .NET (VB.NET), F#, and more.
- These languages all compile into the same intermediate language (IL) that runs on the CLR, allowing components written in different languages to work together seamlessly.

## Components of .NET Framework

There are following components of .NET Framework:

1. CLR (Common Language Runtime)
2. CTS (Common Type System)
3. BCL (Base Class Library)
4. CLS (Common Language Specification)
5. FCL (Framework Class Library)
6. .NET Assemblies
7. XML Web Services
8. Window Services

### **CLR (common language runtime)**

It is an important part of a .NET framework that works like a virtual component of the .NET Framework to execute the different languages program like **c#**, Visual Basic, etc. A CLR also helps to convert a source code into the byte code, and this byte code is known as CIL (Common Intermediate Language) or MSIL (Microsoft Intermediate Language). After converting into a byte code, a CLR uses a JIT compiler at run time that helps to convert a CIL or MSIL code into the machine or native code.

### **CTS (Common Type System)**

It specifies a standard that represent what type of data and value can be defined and managed in computer memory at runtime. A CTS ensures that programming data defined in various languages should be interact with each other to share information. For example, in C# we define data type as int, while in VB.NET we define integer as a data type.

### **BCL (Base Class Library)**

The base class library has a rich collection of libraries features and functions that help to implement many programming languages in the .NET Framework, such as C #, F #, Visual C ++, and more. Furthermore, BCL divides into two parts:

1. **User defined class library** o **Assemblies** - It is the collection of small parts of deployment an application's part. It contains either the DLL (Dynamic Link Library) or exe (Executable) file.
  1. In LL, it uses code reusability, whereas in exe it contains only output file/ or application.
  2. DLL file can't be open, whereas exe file can be open.
  3. DLL file can't be run individually, whereas in exe, it can run individually.
  4. In DLL file, there is no main method, whereas exe file has main method.
2. **Predefined class library** o **Namespace** - It is the collection of predefined class and method that present in .Net. In other languages such as, C we used header files, in java we used package similarly we used "using system" in .NET, where using is a keyword and system is a namespace.

### **CLS (Common language Specification)**

It is a subset of common type system (CTS) that defines a set of rules and regulations which should be followed by every language that comes under the .net framework. In other words, a CLS language should be cross-language integration or interoperability. For example, in C# and VB.NET language, the C# language terminate each statement with semicolon, whereas in VB.NET it is not end with semicolon, and when these statements execute in .NET Framework, it provides a common platform to interact and share information with each other.

### **Microsoft .NET Assemblies**

A .NET assembly is the main building block of the .NET Framework. It is a small unit of code that contains a logical compiled code in the Common Language Infrastructure (CLI), which is used for deployment, security and versioning. It defines in two parts (process) DLL and library (exe) assemblies. When the .NET program is compiled, it generates a

metadata with Microsoft Intermediate Language, which is stored in a file called Assembly.

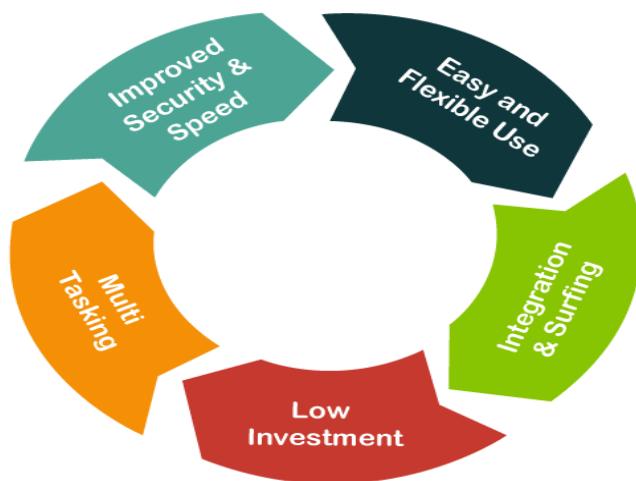
**FCL (Framework Class Library)**

It provides the various system functionality in the .NET Framework that includes classes, interfaces and data types, etc. to create multiple functions and different types of application such as desktop, web, mobile application, etc. In other words, it can be defined as, it provides a base on which various applications, controls and components are built in .NET Framework.

### Key Components of FCL

1. Object type
2. Implementation of data structure
3. Base data types
4. Garbage collection
5. Security and database connectivity
6. Creating common platform for window and web-based application

### Characteristics of .NET Framework



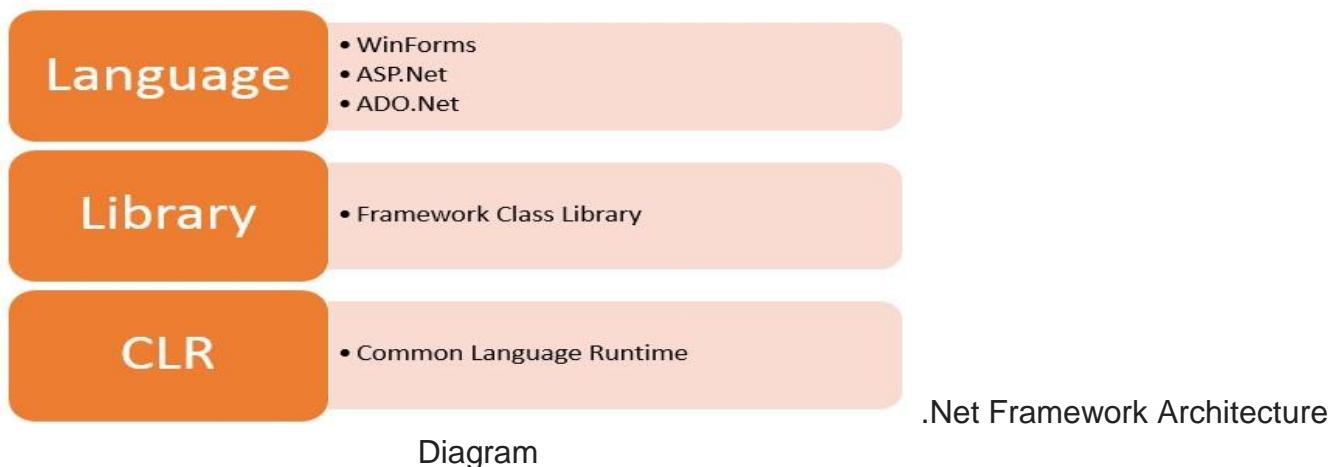
1. CLR (Common Language Runtime)
2. Namespace - Predefined class and function
3. Metadata and Assemblies
4. Application domains

5. It helps to configure and deploy the .net application
6. It provides form and web-based services
7. NET and ASP.NET AJAX
8. LINQ
9. Security and Portability
10. Interoperability
11. It provides multiple environments for developing an application

## .Net Framework Architecture

**.Net Framework Architecture** is a programming model for the .Net platform that provides an execution environment and integration with various programming languages for simple development and deployment of various Windows and desktop applications. It consists of class libraries and reusable components.

The basic architecture of the .Net framework is as shown below.



### .NET Components

The architecture of .Net framework is based on the following key components;

#### **1. Common Language Runtime**

The “Common Language Infrastructure” or CLI is a platform in .Net architecture on which the .Net programs are executed.

The CLI has the following key features:

**Exception Handling** – Exceptions are errors which occur when the application is executed.

Examples of exceptions are:

- If an application tries to open a file on the local machine, but the file is not present.
- If the application tries to fetch some records from a [database](#), but the connection to the database is not valid.

**Garbage Collection** – Garbage collection is the process of removing unwanted resources when they are no longer required.

Examples of garbage collection are

- A File handle which is no longer required. If the application has finished all operations on a file, then the file handle may no longer be required.
- The database connection is no longer required. If the application has finished all operations on a database, then the database connection may no longer be required.

### Working with Various programming languages –

As noted in an earlier section, a developer can develop an application in a variety of .Net programming languages.

1. **Language** – The first level is the programming language itself, the most common ones are VB.Net and C#.
2. **Compiler** – There is a compiler which will be separate for each programming language. So underlying the VB.Net language, there will be a separate VB.Net compiler. Similarly, for C#, you will have another compiler.
3. **Common Language Interpreter** – This is the final layer in .Net which would be used to run a .net program developed in any [programming language](#). So the subsequent compiler will send the program to the CLI layer to run the .Net application.



## 2. Class Library

The .NET Framework includes a set of standard class libraries. A class library is a collection of methods and functions that can be used for the core purpose.

For example, there is a class library with methods to handle all file-level operations. So there is a method which can be used to read the text from a file. Similarly, there is a method to write text to a file.

Most of the methods are split into either the System.\* or Microsoft.\* namespaces. (The asterisk \* just means a reference to all of the methods that fall under the System or Microsoft namespace)

A namespace is a logical separation of methods. We will learn these namespaces more in detail in the subsequent chapters.

## 3. Languages

The types of applications that can be built in the .Net framework is classified broadly into the following categories.

**WinForms** – This is used for developing Forms-based applications, which would run on an end user machine. Notepad is an example of a client-based application.

**ASP.Net** – This is used for developing web-based applications, which are made to run on any browser such as Internet Explorer, Chrome or Firefox.

- The Web application would be processed on a server, which would have Internet Information Services Installed.
- Internet Information Services or IIS is a Microsoft component which is used to execute an [Asp.Net](#) application.
- The result of the execution is then sent to the client machines, and the output is shown in the browser.

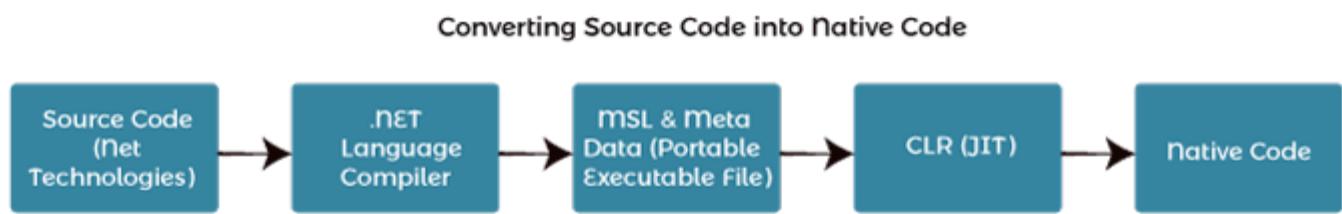
**ADO.Net** – This technology is used to develop applications to interact with Databases such as Oracle or Microsoft [SQL](#) Server.

Microsoft always ensures that .Net frameworks are in compliance with all the supported Windows operating systems.

## .NET Common Language Runtime (CLR)

.NET CLR is a runtime environment that manages and executes the code written in any .NET programming language. CLR is the virtual machine component of the .NET framework. That language's compiler compiles the source code of applications developed using .NET compliant languages into CLR's intermediate language called MSIL, i.e., Microsoft intermediate language code. This code is platform-independent. It is comparable to byte code in java. Metadata is also generated during compilation and MSIL code and stored in a file known as the Manifest file. This metadata is generally about members and types required by CLR to execute MSIL code. A just-in-time compiler component of CLR converts MSIL code into native code of the machine. This code is platform-dependent. CLR manages memory, threads, exceptions, code execution, code safety, verification, and compilation.

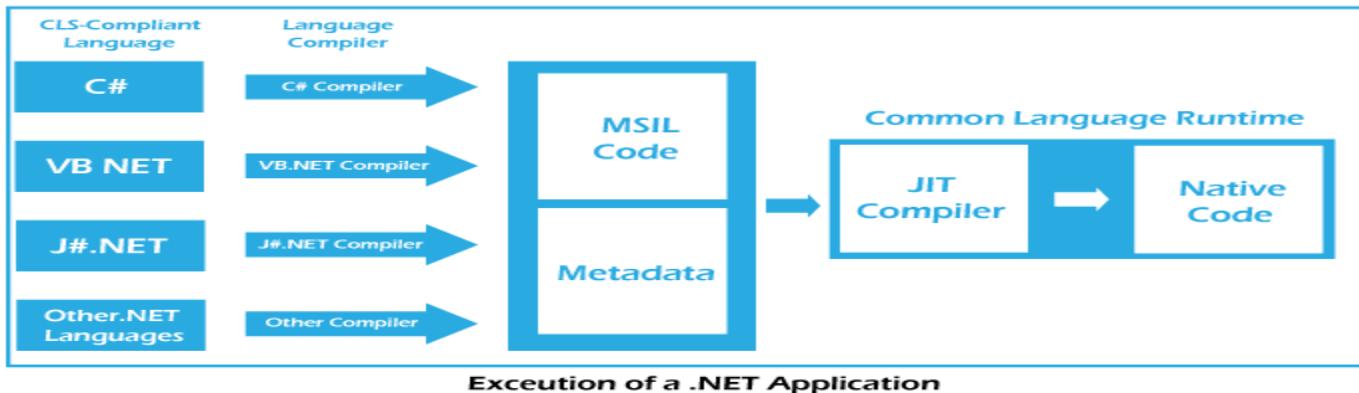
**The following figure shows the conversion of source code into native code.**



**The above figure** converts code into native code, which the CPU can execute.

The main components of CLR are:

- Common type system ○  
Common language  
speciation ○ Garbage  
Collector ○ Just in Time  
Compiler
- Metadata and Assemblies



### 1. Common type system:

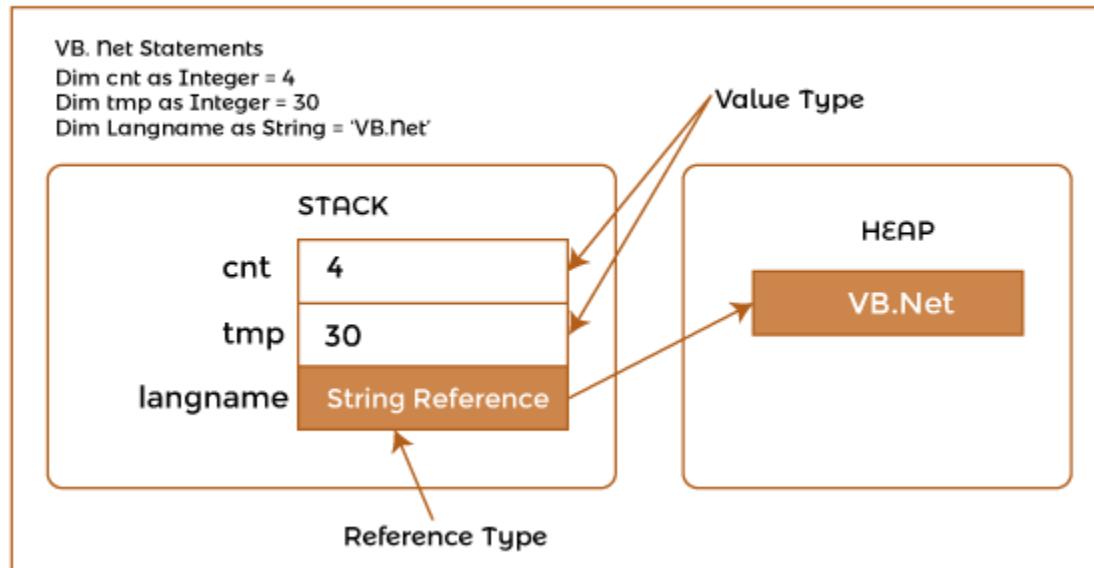
CTS provides guidelines for declaring, using, and managing data types at runtime. It offers cross-language communication. For example, VB.NET has an integer data type, and C# has an int data type for managing integers. After compilation, Int32 is used by both data types. So, CTS provides the data types using managed code. A common type system helps in writing language-independent code.

### **It provides two categories of Types.**

1. **Value Type:** A value type stores the data in memory allocated on the stack or inline in a structure. This category of Type holds the data directory. If one variable's value is copied to another, both the variables store data independently. It can be of inbuilt-in types, user-defined, or enumerations types. Built-in types are primitive data types like numeric, Boolean, char, and date. Users in the source code create user-defined types. An enumeration refers to a set of enumerated values

represented by labels but stored as a numeric type.

### Value Type and Reference Type



1. **Value Type:** A Value type stores data directly in memory. Examples include integers, floating-point numbers, and characters. When a variable of a value type is assigned to another variable, a copy of the data is made.
2. **Reference Type:** A Reference type stores a reference to the value of a memory address and is allocated on the heap. Heap memory is used for dynamic memory allocation. Reference Type does not hold actual data directly but holds the address of data. Whenever a reference type object is made, it copies the address and not actual data. Therefore two variables will refer to the same data. If data of one Reference Type object is changed, the same is reflected for the other object. Reference types can be self-describing types, pointer types, or interface types. The self-describing types may be string, array, and class types that store metadata about themselves.

## 2. Common Language Specification (CLS):

Common Language Specification (CLS) contains a set of rules to be followed by all .NET supported languages. The common rules make it easy to implement language integration and help in cross-language inheritance and debugging. Each language supported by .NET Framework has its own syntax rules. But CLS ensures interoperability among applications developed using .NET languages.

## 3. Garbage Collection:

Garbage Collector is a component of CLR that works as an automatic memory manager. It helps manage memory by automatically allocating memory according to the

requirement. It allocates heap memory to objects. When objects are not in use, it reclaims the memory allocated to them for future use. It also ensures the safety of objects by not allowing one object to use the content of another object.

#### 4. Just in Time (JIT) Compiler:

JIT Compiler is an important component of CLR. It converts the MSIL code into native code (i.e., machine-specific code). The .NET program is compiled either explicitly or implicitly. The developer or programmer calls a particular compiler to compile the program in the explicit compilation. In implicit compilation, the program is compiled twice. The source code is compiled into Microsoft Intermediate Language (MSIL) during the first compilation process. The MSIL code is converted into native code in the second compilation process. This process is called JIT compilation. There are three types of JIT compilers -Pre, Econo, and Normal. Pre JIT Compiler compiles entire MSIL code into native code before execution. Econo JIT Compiler compiles only those parts of MSIL code required during execution and removes those parts that are not required anymore. Normal JIT Compiler also compiles only those parts of MSIL code required during execution but places them in cache for future use. It does not require recompilations of already used parts as they have been placed in cache memory.

#### 5. Metadata:

A Metadata is a binary information about the program, either stored in a CLR Portable Executable file (PE) along with MSIL code or in the memory. During the execution of MSIL, metadata is also loaded into memory for proper interpretation of classes and related. Information used in code. So, metadata helps implement code in a language-neutral manner or achieve language interoperability.

#### 6. Assemblies:

An assembly is a fundamental unit of physical code grouping. It consists of the assembly manifest, metadata, MSIL code, and a set of resources like image files. It is also considered a basic deployment unit, version control, reuse, security permissions, etc.

## .NET CLR Functions

Following are the functions of the CLR.

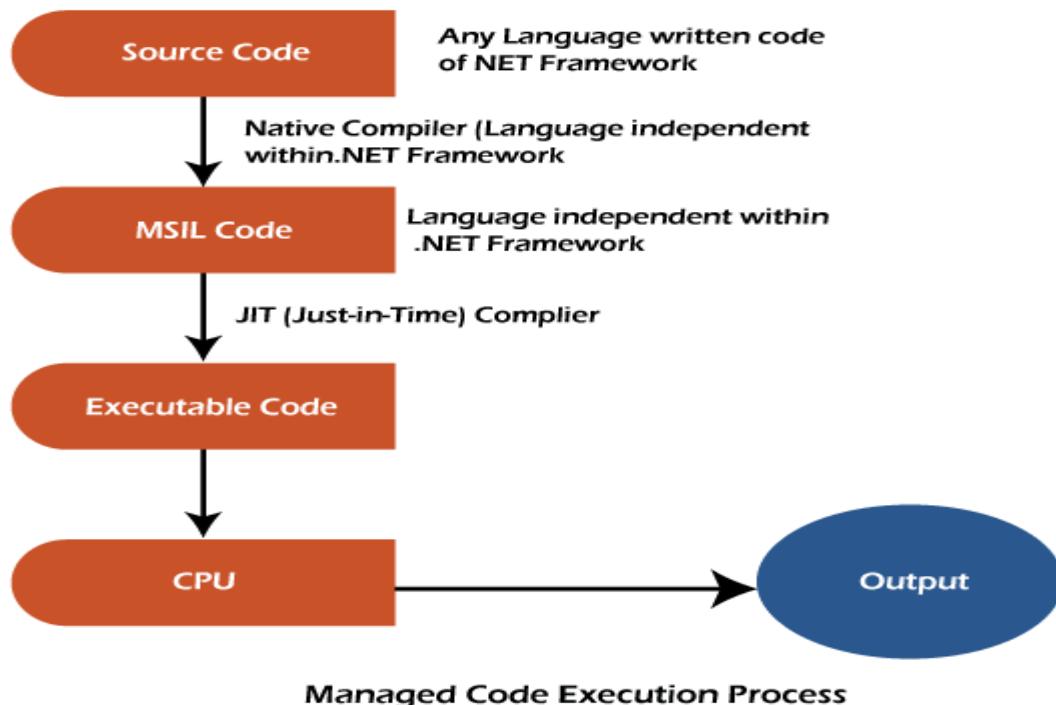
- It converts the program into native code.

- Handles Exceptions
- Provides type-safety ○ Memory management
- Provides security ○ Improved performance ○ Language independent ○ Platform independent ○ Garbage collection ○ Provides language features such as inheritance, interfaces, and overloading for object-oriented programs.

The code that runs with CLR is called managed code, whereas the code outside the CLR is called unmanaged code. The CLR also provides an Interoperability layer, which allows both the managed and unmanaged codes to interoperate.

## 1. Managed code:

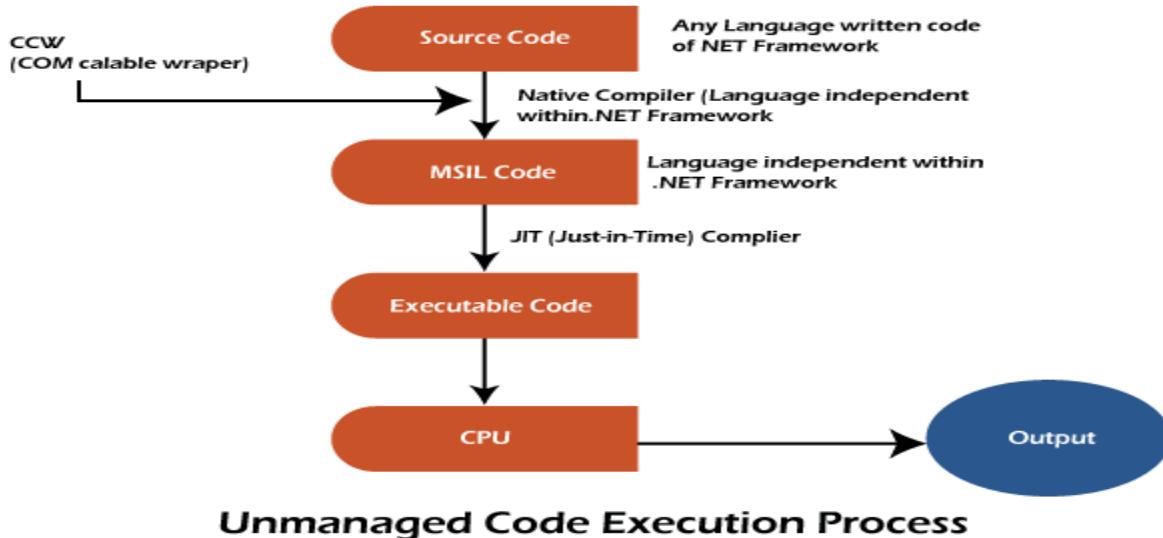
Any language that is written in the .NET framework is managed code. Managed code uses CLR, which looks after your applications by managing memory, handling security, allowing cross-language debugging, etc. The process of managed code is shown in the figure:



## 2. Unmanaged code:

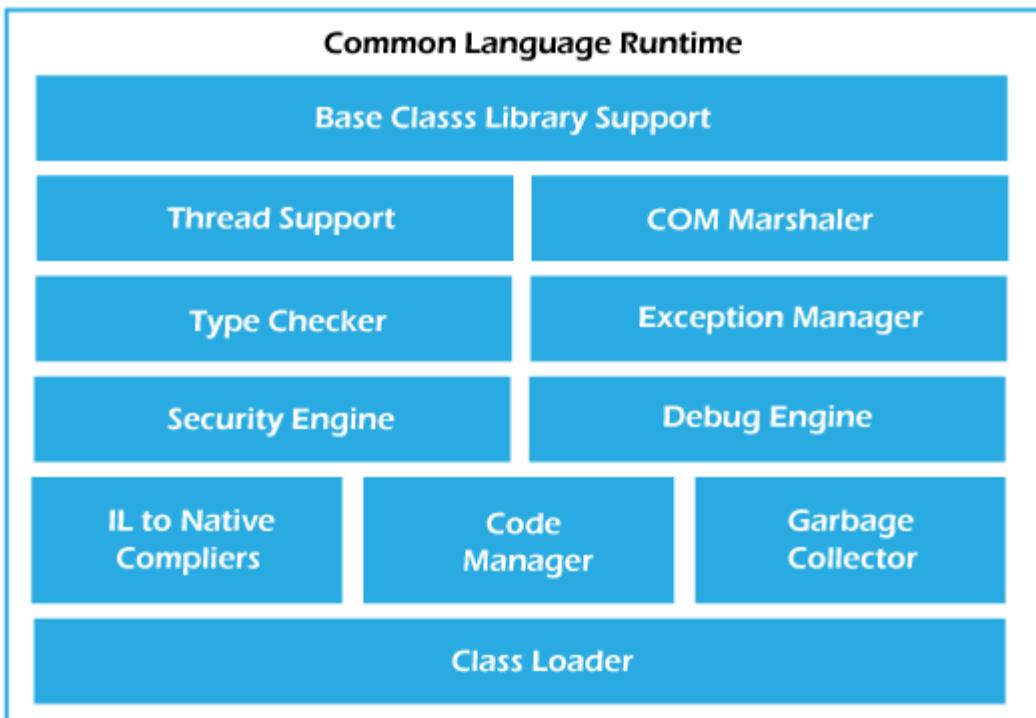
The code developed outside the .NET framework is known as unmanaged code. Applications that do not run under the control of the CLR are said to be unmanaged. Certain languages such as C++ can be used to write such applications, such as low-level access functions of the operating system. Background compatibility with VB, ASP, and

COM are examples of unmanaged code. This code is executed with the help of wrapper classes. The unmanaged code process is shown below:



## ..NET CLR Structure

Following is the component structure of Common Language Runtime.



Components of the Common Language runtime/Architecture of CLR

### **Base Class Library Support**

It is a class library that supports classes for the .NET application.

### **Thread Support**

It manages the parallel execution of the multi-threaded application.

### **COM Marshaler**

It provides communication between the COM objects and the application.

### **Security Engine**

It enforces security restrictions.

### **Debug Engine**

It allows you to debug different kinds of applications.

### **Type Checker**

It checks the types used in the application and verifies that they match the standards provided by the CLR.

### **Code Manager**

It manages code at execution runtime.

### **Garbage Collector**

It releases the unused memory and allocates it to a new application.

### **Exception Handler**

It handles the exception at runtime to avoid application failure.

### **ClassLoader**

It is used to load all classes at run time.

## **What is CTS (Common Type System)?**

1. **Definition:** CTS is a standard that specifies how types are declared, used, and managed in the runtime environment of the .NET Framework.

2. **Purpose:**

- **Interoperability:** Ensures that types defined in different .NET languages can interact seamlessly. For example, a class written in C# can inherit from a class written in VB.NET without issues.
- **Integration:** Facilitates integration between different parts of the .NET Framework, such as the CLR (Common Language Runtime) and the Base Class Library (BCL).
- **Language Neutrality:** Supports multiple programming languages (C#, VB.NET, F#, etc.) by providing a common set of rules and guidelines for type definitions.

3. **Key Features:**

- **Data Types:** Defines common data types such as integers, strings, arrays, and user-defined types (classes, structures, interfaces).
- **Type Safety:** Ensures that types are used correctly and prevents common programming errors related to type mismatches.
- **Member Accessibility:** Specifies rules for accessibility modifiers (public, private, protected) and member visibility across different languages.
- **Method Signatures:** Defines rules for method parameters, return types, and exception handling.

4. **Components:**

- **Value Types vs. Reference Types:** CTS distinguishes between value types (stored directly in memory) and reference types (stored in the heap with a reference to their location).
- **Metadata:** Each type defined in .NET includes metadata that describes its structure and behavior. This metadata is used by the CLR for type checking, memory management, and runtime behavior.

5. **Example:**

```
// Example of a class in C#
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public void Greet()
    {
        Console.WriteLine($"Hello, my name is {Name} and I am {Age} years old.");
    }
}
```

In this example:

- Person is a class defined in C#.
- It has properties (Name and Age) and a method (Greet).
- The CTS ensures that this class can be used across different .NET languages with consistent behavior and interoperability.

## **MSIL:**

MSIL (Microsoft Intermediate Language), also known as IL (Intermediate Language), is a key concept within the .NET Framework architecture. Here's an explanation of what MSIL is and its role in the .NET development process:

### **What is MSIL (Microsoft Intermediate Language)?**

#### **1. Definition:**

- MSIL is a low-level, platform-independent intermediate language used by the .NET Framework.
- When you write code in languages like C#, VB.NET, or F#, it is compiled into MSIL.

#### **2. Purpose:**

- **Intermediate Representation:** MSIL serves as an intermediate representation of your source code.
- **Execution Environment:** It is not directly executable by the CPU but is designed to be executed by the Common Language Runtime (CLR).
- **Portability:** MSIL allows .NET applications to be platform-independent. It can run on any system that has a compatible CLR implementation.

#### **3. Characteristics:**

- **Human-Readable:** While not intended for direct human modification, MSIL is readable and understandable compared to machine code.
- **Verifiability:** MSIL code undergoes verification by the CLR before execution, ensuring type safety and security.
- **Optimization:** MSIL code can be optimized by the CLR during JIT (Just-In-Time) compilation for improved runtime performance.

#### **4. Example:**

Consider a simple C# method like this:

```
public int Add(int a, int b)
{
    return a + b;
}
```

When compiled, this C# code is converted into MSIL, which might look something like this:

```
.method public hidebysig instance int32 Add(int32 a, int32 b) cil managed
{
    // Method implementation in MSIL
    .maxstack 2
    ldarg.1
    ldarg.2    add
    ret }
```

Here's a brief explanation of the MSIL code:

- ldarg.1 and ldarg.2 load the arguments a and b onto the evaluation stack.
- add adds the top two values on the stack.
- ret returns the result from the method.

This MSIL code represents the intermediate form of the Add method and is what the CLR executes during runtime.

## 5. Execution:

- When a .NET application is executed:
  - The source code is compiled into MSIL.
  - The CLR converts MSIL into native machine code (JIT compilation) specific to the executing platform.
  - The native code is executed by the CPU.

# Introduction to C#:

## What is C#?

C# is a programming language created by Microsoft. It's used to write software for computers, websites, and mobile devices. C# is known for being easy to read and learn. It helps developers build all sorts of programs, from simple tools to complex apps.

## What Can You Do with C#?

- **Desktop Apps:** Make programs that run on Windows computers, like games or business tools.
- **Web Apps:** Create websites and web services that can handle lots of users.
- **Mobile Apps:** Build apps for smartphones and tablets that work on different platforms.
- **Games:** Develop video games using popular game engines like Unity.

- **Cloud Services:** Write software that runs on the internet, using services like Microsoft Azure.

### Example of C# Code:

Here's a basic example of a C# program that prints a message:

```
using System;

class Program
{
    static void Main()
    {
        Console.WriteLine("Hello, World!");
    }
}
```

### Why Use C#?

C# is popular because it's flexible, reliable, and supported by Microsoft. It's used by developers worldwide to create software that's efficient, secure, and works well across different devices and platforms.

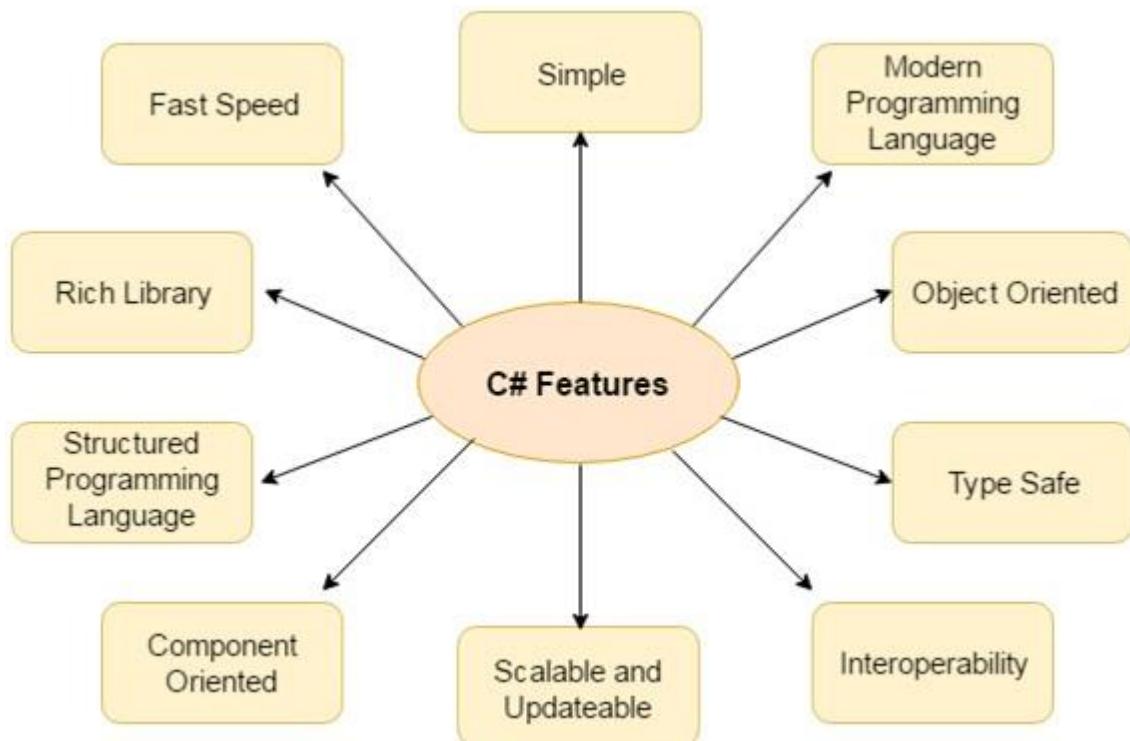
### Java vs C#

<b>The similarities between Java and C#</b>		
S. No	<b>Java</b>	<b>C#</b>
1.	Object-oriented programming languages	Object-oriented programming languages
2.	Java language is descended from C and C++	C# language also is descended from C and C++
3.	Java compiler generates Java byte code	C# compiler generates Microsoft Intermediate Language (MSIL)
4.	Java language includes advanced features like garbage collection	Like Java, it also includes these advanced features
5.	In a lot of areas, they are syntactically similar	In a lot of areas, they are syntactically similar
6.	Java programming gives up on multiple class inheritance in favor of a single inheritance model	Like Java, C# language also gives up on multiple class inheritance in favor of a single inheritance model

## 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# Example: Hello World

In C# programming language, a simple "hello world" program can be written by multiple ways. Let's see the top 4 ways to create a simple C# example:

- o Simple Example o

Using System o Using  
public modifier o Using  
namespace

#### C# Simple Example

```
1. class Program
2. {
3.     static void Main(string[] args)
4.     {
5.         System.Console.WriteLine("Hello World!");
6.     }
7. }
```

Output:

Hello World!

#### Description

**class:** is a keyword which is used to define class.

**Program:** is the class name. A class is a blueprint or template from which objects are created. It can have data members and methods. Here, it has only Main method.

**static:** is a keyword which means object is not required to access static members. So it saves memory.

**void:** is the return type of the method. It doesn't return any value. In such case, return statement is not required.

**Main:** is the method name. It is the entry point for any C# program. Whenever we run the C# program, Main() method is invoked first before any other method. It represents start up of the program.

**string[] args:** is used for command line arguments in C#. While running the C# program, we can pass values. These values are known as arguments which we can use in the program.

**System.Console.WriteLine("Hello World!"):** Here, System is the namespace. Console is the class defined in System namespace. The WriteLine() is the static method of Console class which is used to write the text on the console.

### C# Example: Using System

If we write *using System* before the class, it means we don't need to specify System namespace for accessing any class of this namespace. Here, we are using Console class without specifying System.Console.

```

1.      using System;
2.      class Program
3.      {
4.          static void Main(string[] args)
5.          {
6.              Console.WriteLine("Hello World!");
7.          }
8.      }
```

Output:

Hello World!

### C# Example: Using public modifier

We can also specify *public* modifier before class and Main() method. Now, it can be accessed from outside the class also.

```
1.      using System;
```

```
2.  public class Program  
3.  {  
4.      public static void Main(string[] args)  
5.      {  
6.          Console.WriteLine("Hello World!");  
7.      }  
8.  }
```

Output:

```
Hello World!
```

### C# Example: Using namespace

We can create classes inside the namespace. It is used to group related classes. It is used to categorize classes so that it can be easy to maintain.

```
1.  using System;  
2.  namespace ConsoleApplication1  
3.  {  
4.      public class Program  
5.      {  
6.          public static void Main(string[] args)  
7.          {  
8.              Console.WriteLine("Hello World!");  
9.          }  
10.     }  
11. }
```

Output:

```
Hello World!
```

### C# Variable

A variable is a name of memory location. It is used to store data. Its value can be changed and it can be reused many times.

It is a way to represent memory location through symbol so that it can be easily identified.

Let's see the syntax to declare a variable:

1. type variable\_list;

The example of declaring variable is given below:

1. **int** i, j;
2. **double** d;
3. **float** f;
4. **char** ch;

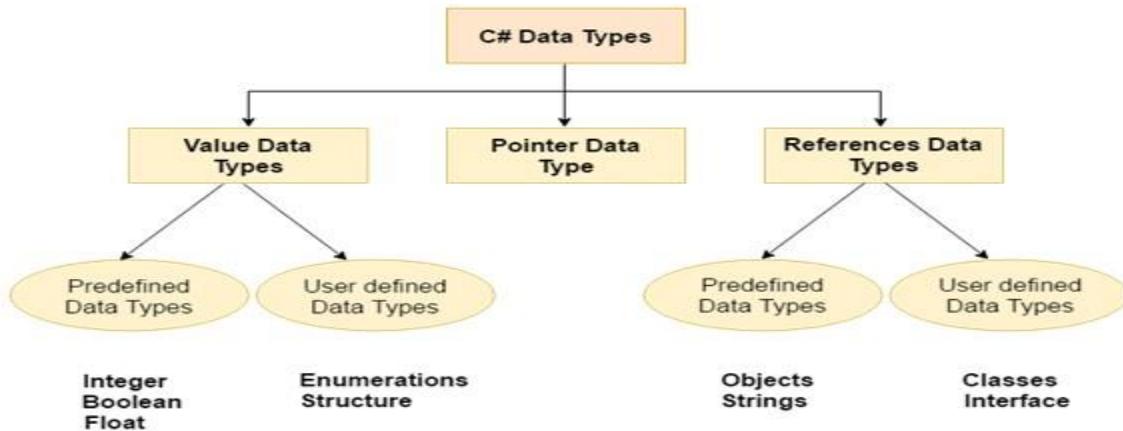
#### Rules for defining variables

- A variable can have alphabets, digits and underscore.
- A variable name can start with alphabet and underscore only. It can't start with digit.
- No white space is allowed within variable name.
- A variable name must not be any reserved word or keyword e.g. char, float etc.

## C# Data Types

A data type specifies the type of data that a variable can store such as integer, floating,

Types	Data Types
Value Data Type	short, int, char, float, double etc
Reference Data Type	String, Class, Object and Interface
Pointer Data Type character etc.	Pointers



There are 3 types of data types in C# language.

### **Value Data Type**

The value data types are integer-based and floating-point based. C# language supports both signed and unsigned literals.

There are 2 types of value data type in C# language.

**1) Predefined Data Types** - such as Integer, Boolean, Float, etc.

**2) User defined Data Types** - such as Structure, Enumerations, etc.

### **Reference Data Type**

The reference data types do not contain the actual data stored in a variable, but they contain a reference to the variables.

If the data is changed by one of the variables, the other variable automatically reflects this change in value.

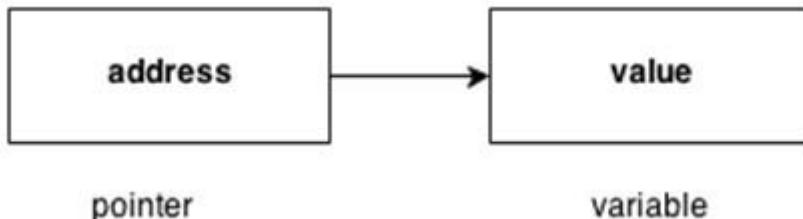
There are 2 types of reference data type in C# language.

**1) Predefined Types** - such as Objects, String.

**2) User defined Types** - such as Classes, Interface.

## Pointer Data Type

The pointer in C# language is a variable, it is also known as locator or indicator that points to an address of a value.



### Declaring a pointer

The pointer in C# language can be declared using \* (asterisk symbol).

1. `int * a; //pointer to int`
2. `char * c; //pointer to char`

## C# operators

An operator is simply a symbol that is used to perform operations. There can be many types of operations like arithmetic, logical, bitwise etc.

There are following types of operators to perform different types of operations in C# language.

- Arithmetic Operators ○
- Relational Operators ○
- Logical Operators ○
- Bitwise Operators ○
- Assignment Operators ○
- Unary Operators ○
- Ternary Operators ○
- Misc  
Operators

	<b>Operator</b>	<b>Type</b>
<b>Binary Operator</b>	+,-,*,/,%	<b>Arithmetic Operators</b>
	<,<=,>,>=,==,!=	<b>Relational Operators</b>
	&&,  ,!	<b>Logical Operators</b>
	&, ,<<,>>,∼,&	<b>Bitwise Operators</b>
	=,+=,-=,*=,/=%=	<b>Assignment Operators</b>
<b>Unary Operator</b>	++,--	<b>Unary Operator</b>
<b>Ternary Operator</b>	?:	<b>Ternary or Conditional Operator</b>

## C# Keywords

A keyword is a reserved word. You cannot use it as a variable name, constant name etc.

In C# keywords cannot be used as identifiers. However, if we want to use the keywords as identifiers, we may prefix the keyword with @ character.

## C# Control Statement

### C# if-else

In C# programming, the *if statement* is used to test the condition. There are various types of if statements in C#.

- o if statement
- o if-else statement
- o nested if statement
- o if-else-if ladder

### C# IF Statement

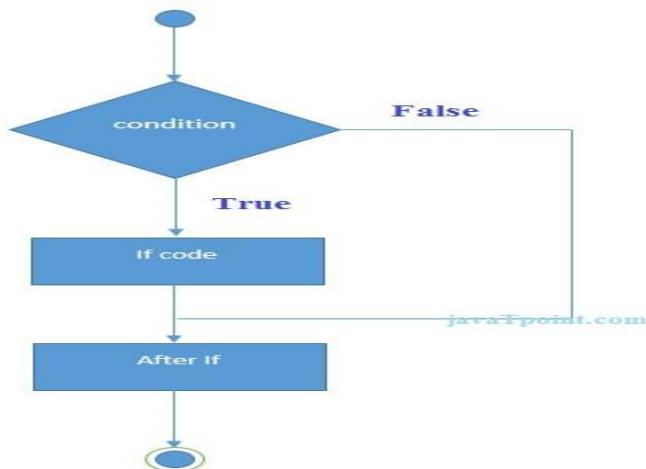
The C# if statement tests the condition. It is executed if condition is true.

#### Syntax:

1. **if**(condition){

2. //code to be executed

3. }



### C# If Example

```

1.     using System;
2. public class IfExample
3. {
4. public static void Main(string[] args)
5. {
6.     int num = 10;
7.     if (num % 2 == 0)
8.     {
9.         Console.WriteLine("It is even number");
10.    }
11.
12. }
13. }
```

Output:

It is even number

### C# IF-else Statement

The C# if-else statement also tests the condition. It executes the *if block* if condition is true otherwise *else block* is executed.

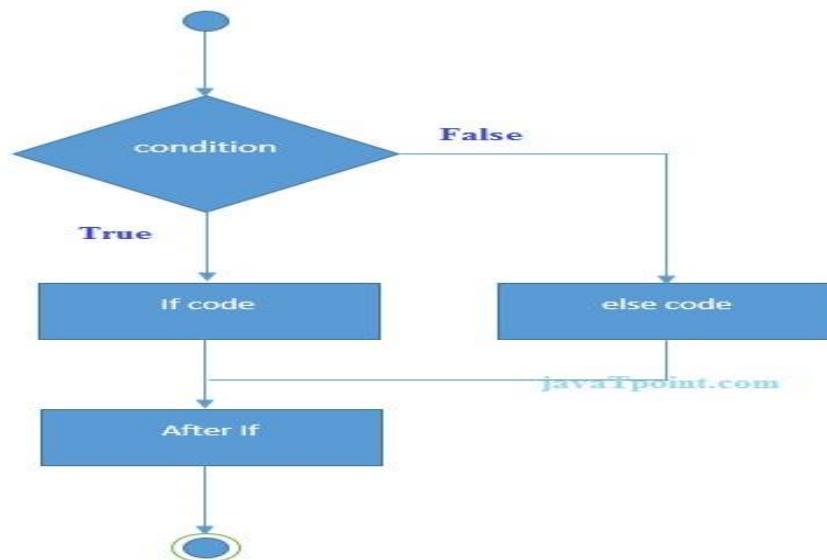
**Syntax:**

1. **if**(condition){
2. //code if condition is true
3. }**else**{
4. //code if condition is false
5. }

C#

If-else

Example



```

1. using System;
2. public class IfExample
3. {
4.     public static void Main(string[] args)
5.     {
6.         int num = 11;
7.         if (num % 2 == 0)
8.         {
9.             Console.WriteLine("It is even number");
10.        }
11.        else
    
```

```
12.    {
13.        Console.WriteLine("It is odd number");
14.    }
15.
16.    }
17. }
```

Output:

```
It is odd number
```

### C# If-else Example: with input from user

In this example, we are getting input from the user using **Console.ReadLine()** method. It returns string. For numeric value, you need to convert it into int using **Convert.ToInt32()** method.

```
1.    using System;
2.    public class IfExample
3.    {
4.        public static void Main(string[] args)
5.        {
6.            Console.WriteLine("Enter a number:");
7.            int num = Convert.ToInt32(Console.ReadLine());
8.
9.            if (num % 2 == 0)
10.            {
11.                Console.WriteLine("It is even number");
12.            }
13.            else
14.            {
15.                Console.WriteLine("It is odd number");
16.            }
17. }
```

```
18.    }
19. }
```

Output:

```
Enter a number:11
It is odd number
```

Output:

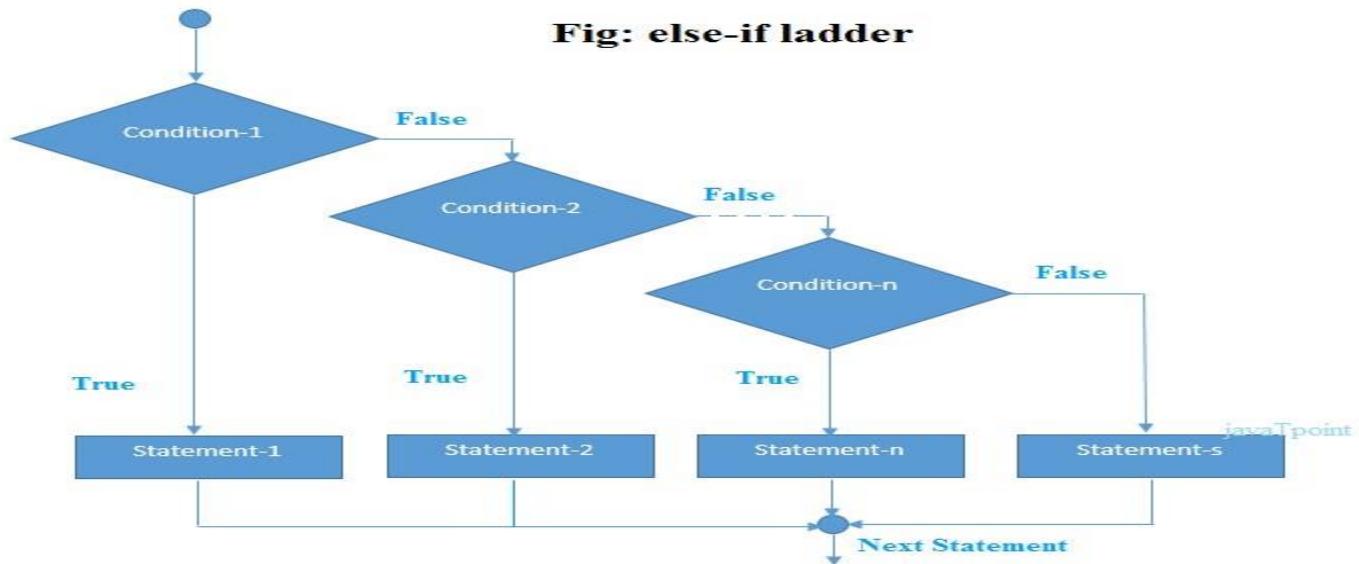
```
Enter a number:12
It is even number
```

### C# IF-else-if ladder Statement

The C# if-else-if ladder statement executes one condition from multiple statements.

#### Syntax:

```
1. if(condition1){
2.   //code to be executed if condition1 is true
3.   else if(condition2){
4.     //code to be executed if condition2 is true
5.   }
6.   else if(condition3){
7.     //code to be executed if condition3 is true
8.   }
9. ...
10. else{
11.   //code to be executed if all the conditions are false
12. }
```



### C# If else-if Example

```

1.  using System;
2.  public class IfExample
3.  {
4.  public static void Main(string[] args)
5.  {
6.  Console.WriteLine("Enter a number to check grade:");
7.  int num = Convert.ToInt32(Console.ReadLine());
8.
9.  if (num < 0 || num > 100)
10. {
11. Console.WriteLine("wrong number");
12. }
13. else if(num >= 0 && num < 50){
14. Console.WriteLine("Fail");
15. }
16. else if (num >= 50 && num < 60)
17. {
18. Console.WriteLine("D Grade");
  
```

```

19.        }
20.        else if (num >= 60 && num < 70)
21.        {
22.            Console.WriteLine("C Grade");
23.        }
24.        else if (num >= 70 && num < 80)
25.        {
26.            Console.WriteLine("B Grade");
27.        }
28.        else if (num >= 80 && num < 90)
29.        {
30.            Console.WriteLine("A Grade");
31.        }
32.        else if (num >= 90 && num <= 100)
33.        {
34.            Console.WriteLine("A+ Grade");
35.        }
36.    }
37. }
```

Output:

```
Enter a number to check grade:66
C Grade
```

Output:

```
Enter a number to check grade:-2
wrong number
```

## C# switch

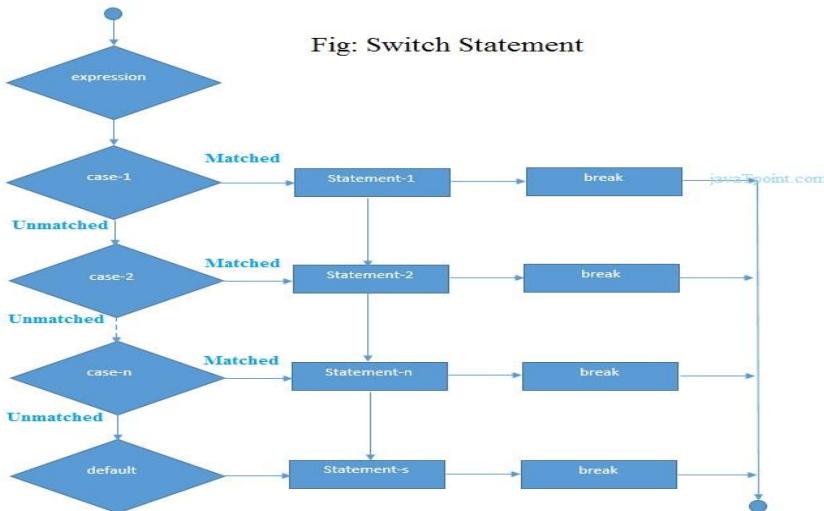
The C# *switch statement* executes one statement from multiple conditions. It is like if-elseif ladder statement in C#.

### Syntax:

```

1. switch(expression){
2.   case value1:
3.     //code to be executed;
4.   break;
5.   case value2:
6.     //code to be executed;
7.   break;
8.   ....
9.
10. default:
11.   //code to be executed if all cases are not matched;
12.   break;
13. }

```



### C# Switch Example

```

1. using System;
2. public class SwitchExample
3. {
4.   public static void Main(string[] args)
5.   {
6.     Console.WriteLine("Enter a number:");
7.     int num = Convert.ToInt32(Console.ReadLine());

```

```

8.
9.     switch (num)
10.    {
11.        case 10: Console.WriteLine("It is 10"); break;
12.        case 20: Console.WriteLine("It is 20"); break;
13.        case 30: Console.WriteLine("It is 30"); break;
14.        default: Console.WriteLine("Not 10, 20 or 30"); break;
15.    }
16. }
17. }
```

Output:

```
Enter a
number:
10 It is
10
```

Output:

```
Enter a number:
55
Not 10, 20 or 30
```

## C# For Loop

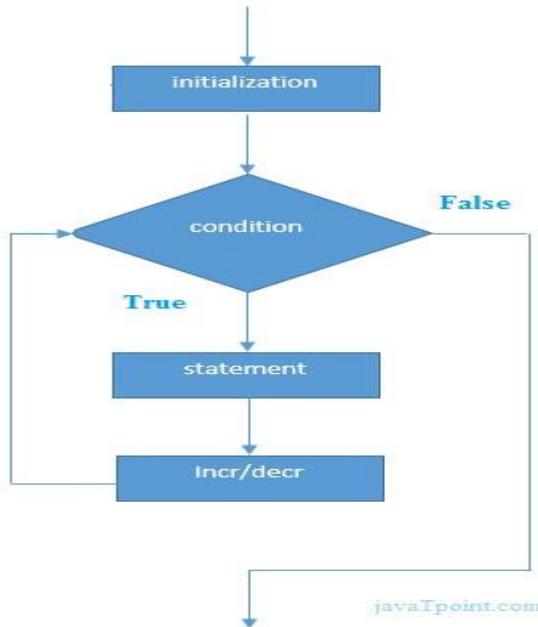
The C# *for loop* is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop than while or do-while loops.

The C# for loop is same as C/C++. We can initialize variable, check condition and increment/decrement value.

### Syntax:

1. **for**(initialization; condition; incr/decr){
2. //code to be executed
3. }

### Flowchart:



## C# For Loop Example

```
1.  using System;
2.  public class ForExample
3.  {
4.  public static void Main(string[] args)
5.  {
6.  for(int i=1;i<=10;i++){
7.  Console.WriteLine(i);
8.  }
9.  }
10. }
```

Output:

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

## C# Nested For Loop

In C#, we can use for loop inside another for loop, it is known as nested for loop. The inner loop is executed fully when outer loop is executed one time. So if outer loop and inner loop are executed 3 times, inner loop will be executed 3 times for each outer loop i.e. total 9 times.

Let's see a simple example of nested for loop in C#.

```
1.      using System;  
2.      public class ForExample  
3.      {  
4.          public static void Main(string[] args)  
5.          {  
6.              for(int i=1;i<=3;i++){  
7.                  for(int j=1;j<=3;j++){  
8.                      Console.WriteLine(i+ " "+j);  
9.                  }  
10.             }  
11.         }  
12.     }
```

Output:

```
1 1  
1 2  
1 3  
2 1  
2 2
```

```
2 3
3 1
3 2
3 3
```

## C# Infinite For Loop

If we use double semicolon in for loop, it will be executed infinite times. Let's see a simple example of infinite for loop in C#.

```
1.     using System;
2.     public class ForExample
3.     {
4.         public static void Main(string[] args)
5.         {
6.             for (; ;)
7.             {
8.                 Console.WriteLine("Infinitive For Loop");
9.             }
10.            }
11.        }
```

**Output:**

```
Infinitive For Loop
```

## C# While Loop

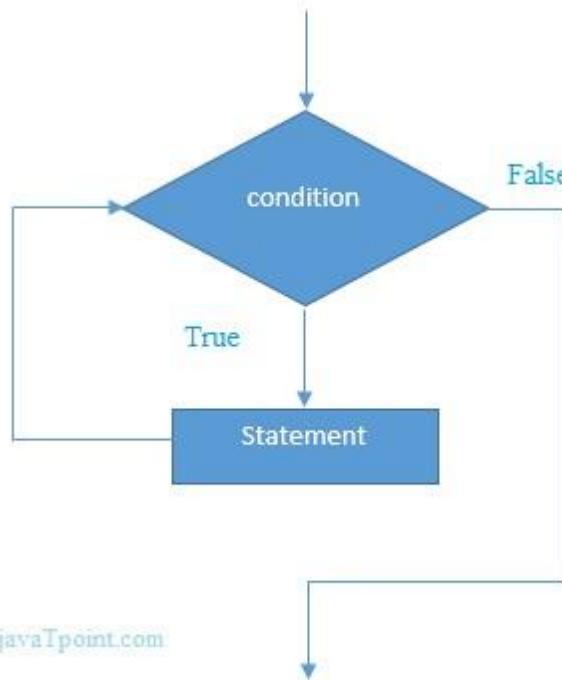
In C#, *while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop than for loop.

**Syntax:**

1. **while**(condition){
2. //code to be executed

3. }

### Flowchart:



### C# While Loop Example

Let's see a simple example of while loop to print table of 1.

T

```

1.     using System;
2. public class WhileExample
3. {
4.     public static void Main(string[] args)
5.     {
6.         int i=1;
7.         while(i<=10)
8.         {
9.             Console.WriteLine(i);
10.            i++;
}

```

```
11.    }
12.    }
13. }
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

### C# Nested While Loop Example:

In C#, we can use while loop inside another while loop, it is known as nested while loop. The nested while loop is executed fully when outer loop is executed once.

Let's see a simple example of nested while loop in C# programming language.

```
1.      using System;
2.      public class WhileExample
3.      {
4.          public static void Main(string[] args)
5.          {
6.              int i=1;
7.              while(i<=3)
8.              {
9.                  int j = 1;
10.                 while (j <= 3)
11.                 {
12.                     Console.WriteLine(i+ " " +j);
13.                     j++;
14.                 }
15.             }
16.         }
17.     }
```

```
14.        }
15.        i++;
16.    }
17.}
18.}
```

Output:

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

### C# Infinitive While Loop Example:

We can also create infinite while loop by passing *true* as the test condition.

```
1.  using System;
2.  public class WhileExample
3.  {
4.      public static void Main(string[] args)
5.      {
6.          while(true)
7.          {
8.              Console.WriteLine("Infinitive While Loop");
9.          }
10.     }
11. }
```

Output:

```
Infinitive While Loop
```

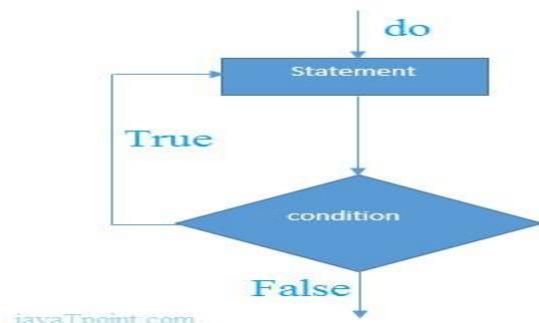
## C# Do-While Loop

The C# *do-while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.

The C# *do-while loop* is executed at least once because condition is checked after loop body.

### Syntax:

1. **do{**
2. **//code to be executed**
3. **}while(condition);**



### C# do-while Loop Example

Let's see a simple example of C# do-while loop to print the table of 1.

1. **using System;**
2. **public class DoWhileExample**
3. **{**
4. **public static void Main(string[] args)**
5. **{**
6. **int i = 1;**

```
7.  
8.     do{  
9.         Console.WriteLine(i);  
10.        i++;  
11.    } while (i <= 10);  
12.  
13. }  
14. }
```

Output:

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

### C# Nested do-while Loop

In C#, if you use do-while loop inside another do-while loop, it is known as nested dowhile loop. The nested do-while loop is executed fully for each outer do-while loop.

Let's see a simple example of nested do-while loop in C#.

```
1.     using System;  
2.     public class DoWhileExample  
3.     {  
4.         public static void Main(string[] args)  
5.         {  
6.             int i=1;  
7.  
8.             do{  
9.                 int j = 1;
```

```

10.
11.     do{
12.         Console.WriteLine(i+" "+j);
13.         j++;
14.     } while (j <= 3) ;
15.     i++;
16. } while (i <= 3) ;
17. }
18. }
```

Output:

```

1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

### C# Infinitive do-while Loop

In C#, if you pass **true** in the do-while loop, it will be infinitive do-while loop.

#### Syntax:

1. **do{**
2. **//code to be executed**
3. **}while(true);**

#### C# Infinitive do-while Loop Example

```

1. using System;
2. public class WhileExample
3. {
4.     public static void Main(string[] args)
```

```

5.    {
6.
7.    do{
8.        Console.WriteLine("Infinitive do-while Loop");
9.    } while(true);
10.   }
11. }
```

Output:

```

Infinitive do-while Loop
```

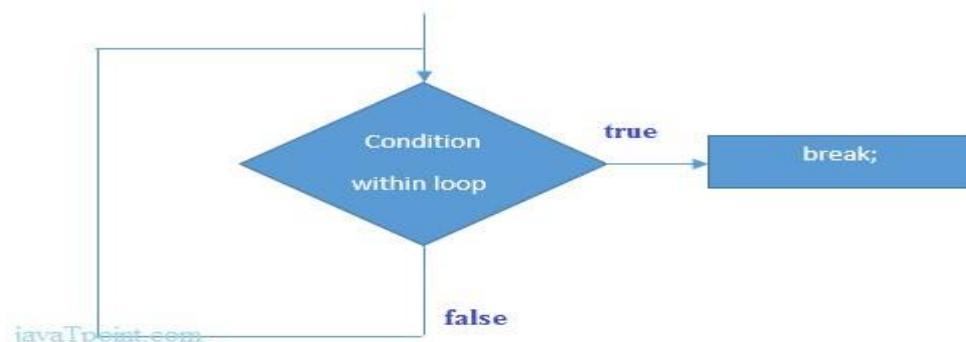
## C# Break Statement

The C# *break* is used to break loop or switch statement. It breaks the current flow of the program at the given condition. In case of inner loop, it breaks only inner loop.

### Syntax:

1. jump-statement;
2. **break;**

### Flowchart:



**Figure: Flowchart of break statement**

## C# Break Statement Example

Let's see a simple example of C# break statement which is used inside the loop.

```
1.     using System;
2.     public class BreakExample
3.     {
4.         public static void Main(string[] args)
5.         {
6.             for (int i = 1; i <= 10; i++)
7.             {
8.                 if (i == 5)
9.                 {
10.                     break;
11.                 }
12.                 Console.WriteLine(i);
13.             }
14.         }
15.     }
```

Output:

```
1
2
3
4
```

## C# Break Statement with Inner Loop

The C# break statement breaks inner loop only if you use break statement inside the inner loop. Let's see the example code:

```
1.     using System;
2.     public class BreakExample
3.     {
4.         public static void Main(string[] args)
5.         {
```

```

6.     for(int i=1;i<=3;i++){
7.         for(int j=1;j<=3;j++){
8.             if(i==2&&j==2){
9.                 break;
10.            }
11.            Console.WriteLine(i+ " "+j);
12.        }
13.    }
14. }
15. }
```

Output:

```

1 1
1 2
1 3
2 1
3 1
3 2
3 3
```

## C# Continue Statement

The C# *continue statement* is used to continue loop. It continues the current flow of the program and skips the remaining code at specified condition. In case of inner loop, it continues only inner loop.

### Syntax:

1. jump-statement;
2. **continue**;

### C# Continue Statement Example

```

1.      using System;
2.      public class ContinueExample
3.      {
4.          public static void Main(string[] args)
```

```
5.      {
6.      for(int i=1;i<=10;i++){
7.      if(i==5){
8.      continue;
9.      }
10.     Console.WriteLine(i);
11.    }
12.   }
13. }
```

Output:

```
1
2
3
4
6
7
8
9
10
```

### C# Continue Statement with Inner Loop

C# Continue Statement continues inner loop only if you use continue statement inside the inner loop.

```
1.      using System;
2.      public class ContinueExample
3.      {
4.      public static void Main(string[] args)
5.      {
6.      for(int i=1;i<=3;i++){
7.      for(int j=1;j<=3;j++){
8.      if(i==2&&j==2){
9.      continue;
10.     }
11.     Console.WriteLine(i+ " "+j);
```

```

12.      }
13.      }
14.      }
15.      }

```

Output:

```

1 1
1 2
1 3
2 1
2 3
3 1
3 2
3 3

```

## C# Goto Statement

The C# goto statement is also known jump statement. It is used to transfer control to the other part of the program. It unconditionally jumps to the specified label.

It can be used to transfer control from deeply nested loop or switch case label.

Currently, it is avoided to use goto statement in C# because it makes the program complex.

### C# Goto Statement Example

Let's see the simple example of goto statement in C#.

```

1.      using System;
2.      public class GotoExample
3.      {
4.          public static void Main(string[] args)
5.          {
6.              ineligible:
7.              Console.WriteLine("You are not eligible to vote!");
8.
9.              Console.WriteLine("Enter your age:\n");

```

```

10.     int age = Convert.ToInt32(Console.ReadLine());
11.     if (age < 18){
12.         goto ineligible;
13.     }
14.     else
15.     {
16.         Console.WriteLine("You are eligible to vote!");
17.     }
18. }
19. }
```

Output:

```

You are not eligible to vote!
Enter your age:
11
You are not eligible to vote!
Enter your age:
5
You are not eligible to vote!
Enter your age:
26
You are eligible to vote!
```

## C# Comments

The C# comments are statements that are not executed by the compiler. The comments in C# programming can be used to provide explanation of the code, variable, method or class. By the help of comments, you can hide the program code also.

There are two types of comments in C#.

- Single Line comment
- Multi Line comment

### C# Single Line Comment

The single line comment starts with // (double slash). Let's see an example of single line comment in C#.

```
1.  using System;
2.  public class CommentExample
3.  {
4.  public static void Main(string[] args)
5.  {
6.  int x = 10;//Here, x is a variable
7.  Console.WriteLine(x);
8.  }
9. }
```

Output:

```
10
```

### C# Multi Line Comment

The C# multi line comment is used to comment multiple lines of code. It is surrounded by slash and asterisk /\* ..... \*/. Let's see an example of multi line comment in C#.

```
1.  using System;
2.  public class CommentExample
3.  {
4.  public static void Main(string[] args)
5.  {
6.  /* Let's declare and
7.  print variable in C#. */
8.  int x=20;
9.  Console.WriteLine(x);
10. }
11. }
```

Output:

20

## C# Function

### C# Function

Function is a block of code that has a signature. Function is used to execute statements specified in the code block. A function consists of the following components:

**Function name:** It is a unique name that is used to make Function call.

**Return type:** It is used to specify the data type of function return value.

**Body:** It is a block that contains executable statements.

**Access specifier:** It is used to specify function accessibility in the application.

**Parameters:** It is a list of arguments that we can pass to the function during call.

### C# Function Syntax

1. <access-specifier><**return**-type>FunctionName(<parameters>)
2. {
3. // function body
4. // return statement
5. }

Access-specifier, parameters and return statement are optional.

Let's see an example in which we have created a function that returns a string value and takes a string parameter.

### C# Function: using no parameter and return type

A function that does not return any value specifies **void** type as a return type. In the following example, a function is created without return type.

1. **using** System;
2. **namespace** FunctionExample
3. {

```
4.     class Program
5.     {
6.         // User defined function without return type
7.         public void Show() // No Parameter
8.         {
9.             Console.WriteLine("This is non parameterized function");
10.            // No return statement
11.        }
12.        // Main function, execution entry point of the program
13.        static void Main(string[] args)
14.        {
15.            Program program = new Program(); // Creating Object
16.            program.Show(); // Calling Function
17.        }
18.    }
19. }
```

**Output:**

```
This is non parameterized function
```

**C# Function: using parameter but no return type**

```
1.     using System;
2.     namespace FunctionExample
3.     {
4.         class Program
5.         {
6.             // User defined function without return type
7.             public void Show(string message)
8.             {
9.                 Console.WriteLine("Hello " + message);
10.                // No return statement
11.            }
12.        }
13.    }
```

```
11.    }
12.    // Main function, execution entry point of the program
13.    static void Main(string[] args)
14.    {
15.        Program program = new Program(); // Creating Object
16.        program.Show("Rahul Kumar"); // Calling Function
17.    }
18. }
19. }
```

### Output:

```
Hello Rahul Kumar
```

A function can have zero or any number of parameters to get data. In the following example, a function is created without parameters. A function without parameter is also known as **non-parameterized** function.

### C# Function: using parameter and return type

```
1.    using System;
2.    namespace FunctionExample
3.    {
4.        class Program
5.        {
6.            // User defined function
7.            public string Show(string message)
8.            {
9.                Console.WriteLine("Inside Show Function");
10.               return message;
11.            }
12.            // Main function, execution entry point of the program
```

```

    Program();

13. static void Main(string[] args)
14. {
15.     Program program = new
16.     string message = program.Show("Rahul Kumar");
17.     Console.WriteLine("Hello " + message);
18. }
19. }
20. }

```

### Output:

Inside Show Function  
Hello Rahul Kumar

## C# Call By Value

In C#, value-type parameters are that pass a copy of original value to the function rather than reference. It does not modify the original value. A change made in passed value does not alter the actual value. In the following example, we have pass value during function call.

### C# Call By Value Example

```

1. using System;
2. namespace CallByValue
3. {
4.     class Program
5.     {
6.         // User defined function
7.         public void Show(int val)
8.     {

```

```

15.    {
16.        int val = 50;
17.        Program program = new Program(); // Creating Object
18.        Console.WriteLine("Value before");
19.        val *= val; // Manipulating value
20.        Console.WriteLine("Value inside the show function "+val);
21.        // No return statement
22.    }
23.    // Main function, execution entry point of the program
24.    static void Main(string[] args)
25.    {
26.        calling the function "+val);
27.        program.Show(val); // Calling Function by passing value
28.        Console.WriteLine("Value after calling the function " + val);
29.    }
30.}

```

### Output:

```

Value before calling the function 50
Value inside the show function 2500
Value after calling the function 50

```

## C# Call By Reference

C# provides a **ref** keyword to pass argument as reference-type. It passes reference of arguments to the function rather than copy of original value. The changes in passed values are permanent and **modify** the original variable value.

### C# Call By Reference Example

```

1.    using System;
2.    namespace CallByReference
3.    {
4.        class Program

```

```

15.    {
16.        int val = 50;
17.        Program program = new Program(); // Creating Object
18.        Console.WriteLine("Value before");
19.        {
20.            // User defined function
21.            public void Show(ref int val)
22.            {
23.                val *= val; // Manipulating value
24.                Console.WriteLine("Value inside the show function " + val);
25.                // No return statement
26.            }
27.            // Main function, execution entry point of the program
28.            static void Main(string[] args)
29.            {
30.                calling the function " + val);
31.                program.Show(ref val); // Calling Function by passing reference
32.                Console.WriteLine("Value after calling the function " + val);
33.            }
34.        }
35.    }

```

#### **Output:**

```

Value before calling the function 50
Value inside the show function 2500
Value after calling the function 2500

```

## **C# Out Parameter**

C# provides **out** keyword to pass arguments as out-type. It is like reference-type, except that it does not require variable to initialize before passing. We must use **out** keyword to pass argument as out-type. It is useful when we want a function to return multiple values.

```

15.    {
16.        int val = 50;
17.        Program program = new Program(); // Creating Object
18.        Console.WriteLine("Value before")

```

### C# Out Parameter Example 1

```

1.     using System;
2.     namespace OutParameter
3.     {
4.         class Program
5.         {
6.             // User defined function
7.             public void Show(out int val) // Out parameter
8.             {
9.                 int square = 5;
10.                val = square;
11.                val *= val; // Manipulating value
12.            }
13.            // Main function, execution entry point of the program
14.            static void Main(string[] args)
15.                passing out variable " + val);
16.                program.Show(out val); // Passing out argument
17.                Console.WriteLine("Value after receiving the out variable " + val);
18.            }
19.        }
20.    }
21. }

```

### Output:

```

Value before passing out variable 50
Value after receiving the out variable 25

```

The following example demonstrates that how a function can return multiple values.

```
15.    {
16.        int val = 50;
17.        Program program = new Program(); // Creating Object
18.        Console.WriteLine("Value before")
```

### C# Out Parameter Example 2

```
1.    using System;
2.    namespace OutParameter
3.    {
4.        class Program
5.        {
6.            // User defined function
7.            public void Show(out int a, out int b) // Out parameter
8.            {
9.                int square = 5;
10.               a = square;
11.               b = square;
12.               // Manipulating value
13.               a *= a;
14.               b *= b;
15.            }
16.            // Main function, execution entry point of the program
17.            static void Main(string[] args)
```

```

{
    int
    Program() // Creating Object
    before
18.
19.        val1 = 50, val2 = 100;
20.        program = new
21.        Console.WriteLine("Value passing \n val1 = " + val1+ " \n val2 = " +val2);
22.        program.Show(out val1, out val2); // Passing out argument
23.        Console.WriteLine("Value after passing \n val1 = " + val1 + " \n val2 = " +
24.        val2);
25.    }
26.}

```

**Output:**

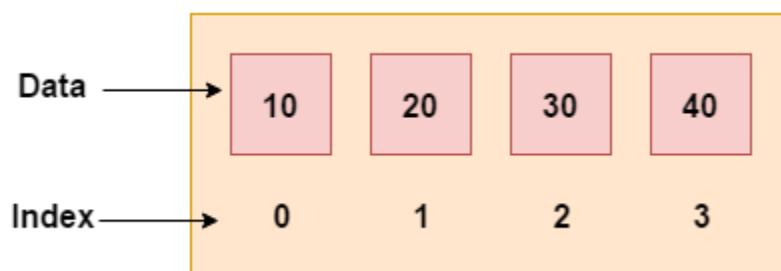
```

Value before passing
val1 = 50
val2 = 100
Value after passing
val1 = 25
val2 = 25

```

**C# Arravs****C# Arrays**

Like other programming languages, 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**. In C#, array index starts from 0. We can store only fixed set of elements in C# array.



## 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:

1. Single Dimensional Array
2. Multidimensional Array
3. Jagged Array

### C# Single Dimensional Array

To create single dimensional array, you need to use square brackets [] after the type.

1. **int[] arr = new int[5];//creating array**

You cannot place square brackets after the identifier.

1. **int arr[] = new int[5];//compile time error**

Let's see a simple example of C# array, where we are going to declare, initialize and traverse array.

1. **using System;**
2. **public class** ArrayExample
3. {

```
4. public static void Main(string[] args)
5. {
6.     int[] arr = new int[5];//creating array
7.     arr[0] = 10;//initializing array
8.     arr[2] = 20;
9.     arr[4] = 30;
10.
11.    //traversing array
12.    for (int i = 0; i < arr.Length; i++)
13.    {
14.        Console.WriteLine(arr[i]);
15.    }
16. }
17. }
```

Output:

```
10
0
20
0
30
```

### C# Array Example: Declaration and Initialization at same time

There are 3 ways to initialize array at the time of declaration.

```
1. int[] arr = new int[5]{ 10, 20, 30, 40, 50 };
```

We can omit the size of array.

```
1. int[] arr = new int[]{ 10, 20, 30, 40, 50 };
```

We can omit the new operator also.

```
1. int[] arr = { 10, 20, 30, 40, 50 };
```

Let's see the example of array where we are declaring and initializing array at the same time.

```
1.  using System;
2.  public class ArrayExample
3.  {
4.    public static void Main(string[] args)
5.    {
6.      int[] arr = { 10, 20, 30, 40, 50 };//Declaration and Initialization of array
7.
8.      //traversing array
9.      for (int i = 0; i < arr.Length; i++)
10.     {
11.       Console.WriteLine(arr[i]);
12.     }
13.   }
14. }
```

Output:

```
10
20
30
40
50
```

### C# Array Example: Traversal using foreach loop

We can also traverse the array elements using foreach loop. It returns array element one by one.

```
1.  using System;
2.  public class ArrayExample
3.  {
4.    public static void Main(string[] args)
5.    {
```

```

6. int[] arr = { 10, 20, 30, 40, 50 }; //creating and initializing array
7.
8. //traversing array
9. foreach (int i in arr)
10. {
11.     Console.WriteLine(i);
12. }
13. }
14. }
```

Output:

```

10
20
30
40
50
```

## C# Passing Array to Function

In C#, to reuse the array logic, we can create function. To pass array to function in C#, we need to provide only array name.

1. functionname(arrayname); //passing array

### C# Passing Array to Function Example: print array elements

Let's see an example of C# function which prints the array elements.

```

1.      using System;
2.      public class ArrayExample
3.      {
4.          static void printArray(int[] arr)
5.          {
6.              Console.WriteLine("Printing array elements:");
7.              for (int i = 0; i < arr.Length; i++)
8.              {
9.                  Console.WriteLine(arr[i]);
10.             }
```

```

10.    }
11.    }
12.    public static void Main(string[] args)
13.    {
14.        int[] arr1 = { 25, 10, 20, 15, 40, 50 };
15.        int[] arr2 = { 12, 23, 44, 11, 54 };
16.        printArray(arr1);//passing array to function
17.        printArray(arr2);
18.    }
19. }
```

Output:

```

Printing array elements:
25
10
20
15
40
50
Printing array elements:
12
23
44
11
54
```

### C# Passing Array to Function Example: Print minimum number

Let's see an example of C# array which prints minimum number in an array using function.

```

1.    using System;
2.    public class ArrayExample
3.    {
4.        static void printMin(int[] arr)
5.        {
6.            int min = arr[0];
7.            for (int i = 1; i < arr.Length; i++)
8.            {
```

```
9.     if (min > arr[i])
10.    {
11.        min = arr[i];
12.    }
13.    }
14.    Console.WriteLine("Minimum element is: " + min);
15. }
16. public static void Main(string[] args)
17. {
18.     int[] arr1 = { 25, 10, 20, 15, 40, 50 };
19.     int[] arr2 = { 12, 23, 44, 11, 54 };
20.
21.     printMin(arr1);//passing array to function
22.     printMin(arr2);
23. }
24. }
```

Output:

```
Minimum element is: 10
Minimum element is: 11
```

## C# Multidimensional Arrays

The multidimensional array is also known as rectangular arrays in C#. It can be two dimensional or three dimensional. The data is stored in tabular form (row \* column) which is also known as matrix.

To create multidimensional array, we need to use comma inside the square brackets. For example:

1. **int[,] arr=new int[3,3];//declaration of 2D array**
2. **int[, ,] arr=new int[3,3,3];//declaration of 3D array**

## C# Multidimensional Array Example

Let's see a simple example of multidimensional array in C# which declares, initializes and traverse two dimensional array.

```

1.  using System;
2.  public class MultiArrayExample
3.  {
4.      public static void Main(string[] args)
5.      {
6.          int[,] arr=new int[3,3];//declaration of 2D array
7.          arr[0,1]=10;//initialization
8.          arr[1,2]=20;
9.          arr[2,0]=30;
10.
11.         //traversal
12.         for(int i=0;i<3;i++){
13.             for(int j=0;j<3;j++){
14.                 Console.Write(arr[i,j]+" ");
15.             }
16.             Console.WriteLine();//new line at each row
17.         }
18.     }
19. }
```

Output:

```
0 10 0
0 0 20
30 0 0
```

**C# Multidimensional Array Example: Declaration and initialization at same time** There are 3 ways to initialize multidimensional array in C# while declaration.

1. **int[]** arr = **new int[3,3]**= { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };

We can omit the array size.

1. **int[]** arr = **new int[]**{ { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };

We can omit the new operator also.

1. **int[]** arr = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };

Let's see a simple example of multidimensional array which initializes array at the time of declaration.

```

1.   using System;
2.   public class MultiArrayExample
3.   {
4.     public static void Main(string[] args)
5.     {
6.       int[] arr = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };//declaration and initialization
7.
8.       //traversal
9.       for(int i=0;i<3;i++){
10.         for(int j=0;j<3;j++){
11.           Console.Write(arr[i,j]+" ");
12.         }
13.         Console.WriteLine();//new line at each row
14.       }
15.     }
16.   }
```

Output:

```

1 2 3
4 5 6
7 8 9

```

## C# Object Class

### C# Object and Class

Since C# is an object-oriented language, program is designed using objects and classes in C#.

#### C# Object

In C#, Object is a real world entity, for example, chair, car, pen, mobile, laptop etc.

In other words, object is an entity that has state and behavior. Here, state means data and behavior means functionality.

Object is a runtime entity, it is created at runtime.

Object is an instance of a class. All the members of the class can be accessed through object.

Let's see an example to create object using new keyword.

#### 1. Student s1 = **new** Student();//creating an object of Student

In this example, Student is the type and s1 is the reference variable that refers to the instance of Student class. The new keyword allocates memory at runtime.

#### C# Class

In C#, class is a group of similar objects. It is a template from which objects are created. It can have fields, methods, constructors etc.

Let's see an example of C# class that has two fields only.

1. **public class** Student
2. {
3.   **int** id;//field or data member
4.   String name;//field or data member

5. }

### C# Object and Class Example

Let's see an example of class that has two fields: id and name. It creates instance of the class, initializes the object and prints the object value.

```

1. using System;
2. public class Student
3. {
4.     int id;//data member (also instance variable)    5.     String name;//data member(also
   instance variable)
6.
7.     public static void Main(string[] args)
8.     {
9.         Student s1 = new Student();//creating an object of Student
10.        s1.id = 101;
11.        s1.name = "Sonoo Jaiswal";
12.        Console.WriteLine(s1.id); 13.        Console.WriteLine(s1.name);
14.
15.    }
16. }
```

Output:

```
101
Sonoo Jaiswal
```

### C# Class Example 2: Having Main() in another class

Let's see another example of class where we are having Main() method in another class. In such case, class must be public.

```

1. using System;
2. public class Student
3. {
4.     public int id;
```

```
5.     public String name;
6. }
7. class TestStudent{
8.     public static void Main(string[] args)
9. {
10.     Student s1 = new Student();
11.     s1.id = 101;
12.     s1.name = "Sonoo Jaiswal";
13.     Console.WriteLine(s1.id); 14.     Console.WriteLine(s1.name);
15.
16. }
17. }
```

Output:

```
101
Sonoo Jaiswal
```

### C# Class Example 3: Initialize and Display data through method

Let's see another example of C# class where we are initializing and displaying object through method.

```
1.     using System;
2.     public class Student
3. {
4.     public int id;
5.     public String name;
6.     public void insert(int i, String n)
7. {
8.     id = i;
9.     name = n;
10. }
11.     public void display()
```

```
12.    {
13.        Console.WriteLine(id + " " + name);
14.    }
15. }
16. class TestStudent{
17.     public static void Main(string[] args)
18.     {
19.         Student s1 = new Student();
20.         Student s2 = new Student();
21.         s1.insert(101, "Ajeet");
22.         s2.insert(102, "Tom");
23.         s1.display();
24.         s2.display();
25.
26.     }
27. }
```

Output:

```
101 Ajeet
102 Tom
```

#### C# Class Example 4: Store and Display Employee Information

```
1.      using System;
2.      public class Employee
3.      {
4.          public int id;
5.          public String name;
6.          public float salary;
7.          public void insert(int i, String n, float s)
8.          {
9.              id = i;
```

```

10.     name = n;
11.     salary = s;
12. }
13. public void display()
14. {
15.     Console.WriteLine(id + " " + name+ " " +salary);
16. }
17. }
18. class TestEmployee{
19.     public static void Main(string[] args)
20.     {
21.         Employee e1 = new Employee();
22.         Employee e2 = new Employee();
23.         e1.insert(101, "Sonoo",890000f);
24.         e2.insert(102, "Mahesh", 490000f);
25.         e1.display();
26.         e2.display();
27.
28.     }
29. }
```

Output:

```
101 Sonoo 890000
102 Mahesh 490000
```

## C# Constructor

In C#, constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new object generally. The constructor in C# has the same name as class or struct.

There can be two types of constructors in C#.

- Default constructor
- Parameterized constructor

## C# Default Constructor

A constructor which has no argument is known as default constructor. It is invoked at the time of creating object.

### C# Default Constructor Example: Having Main() within class

```

1.      using System;
2.      public class Employee
3.      {
4.          public Employee()
5.          {
6.              Console.WriteLine("Default Constructor Invoked");
7.          }
8.          public static void Main(string[] args)
9.          {
10.             Employee e1 = new Employee();
11.             Employee e2 = new Employee();
12.         }
13.     }
```

Output:

```
Default Constructor Invoked
Default Constructor Invoked
```

### C# Default Constructor Example: Having Main() in another class

Let's see another example of default constructor where we are having Main() method in another class.

```

1.      using System;
2.      public class Employee
3.      {
4.          public Employee()
5.          {
```

```
6.     Console.WriteLine("Default Constructor Invoked");
7. }
8. }
9. class TestEmployee{
10.    public static void Main(string[] args)
11.    {
12.        Employee e1 = new Employee();
13.        Employee e2 = new Employee();
14.    }
15. }
```

Output:

```
Default Constructor Invoked
Default Constructor Invoked
```

### C# Parameterized Constructor

A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects.

```
1.    using System;
2.    public class Employee
3.    {
4.        public int id;
5.        public String name;
6.        public float salary;
7.        public Employee(int i, String n, float s)
8.        {
9.            id = i;
10.           name = n;
11.           salary = s;
12.        }
13.        public void display()
14.        {
```

```

15.     Console.WriteLine(id + " " + name+" "+salary);
16. }
17. }
18. class TestEmployee{
19.     public static void Main(string[] args)
20.     {
21.         Employee e1 = new Employee(101, "Sonoo", 890000f);
22.         Employee e2 = new Employee(102, "Mahesh", 490000f);
23.         e1.display();
24.         e2.display();
25.
26.     }
27. }
```

Output:

```

101 Sonoo 890000
102 Mahesh 490000
```

## C# Destructor

A destructor works opposite to constructor, It destructs the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically.

**Note: C# destructor cannot have parameters. Moreover, modifiers can't be applied on destructors.**

### C# Constructor and Destructor Example

Let's see an example of constructor and destructor in C# which is called automatically.

```

1.     using System;
2.     public class Employee
3.     {
4.         public Employee()
5.         {
6.             Console.WriteLine("Constructor Invoked");
```

```

7.      }
8.      ~Employee()
9.      {
10.     Console.WriteLine("Destructor Invoked");
11.    }
12.   }
13.   class TestEmployee{
14.     public static void Main(string[] args)
15.     {
16.       Employee e1 = new Employee();
17.       Employee e2 = new Employee();
18.     }
19.   }

```

Output:

```

Constructor Invoked
Constructor Invoked
Destructor Invoked
Destructor Invoked

```

## C# this

In c# programming, this is a keyword that refers to the current instance of the class. There can be 3 main usage of this keyword in C#.

- It can be used **to refer current class instance variable**. It is used if field names (instance variables) and parameter names are same, that is why both can be distinguish easily.
- It can be used **to pass current object as a parameter to another method**.
- It can be used **to declare indexers**.

### C# this example

Let's see the example of this keyword in C# that refers to the fields of current class.

1.     **using** System;

```
2.  public class Employee
3.  {
4.      public int id;
5.      public String name;
6.      public float salary;
7.      public Employee(int id, String name, float salary)
8.      {
9.          this.id = id;
10.         this.name = name;
11.         this.salary = salary;
12.     }
13.     public void display()
14.     {
15.         Console.WriteLine(id + " " + name + " " + salary);
16.     }
17. }
18. class TestEmployee{
19.     public static void Main(string[] args)
20.     {
21.         Employee e1 = new Employee(101, "Sonoo", 890000f);
22.         Employee e2 = new Employee(102, "Mahesh", 490000f);
23.         e1.display();
24.         e2.display();
25.
26.     }
27. }
```

Output:

```
101 Sonoo 890000
102 Mahesh 490000
```

## C# static

In C#, static is a keyword or modifier that belongs to the type not instance. So instance is not required to access the static members. In C#, static can be field, method, constructor, class, properties, operator and event.

**Note: Indexers and destructors cannot be static.**

**Advantage of C# static keyword**

**Memory efficient:** Now we don't need to create instance for accessing the static members, so it saves memory. Moreover, it belongs to the type, so it will not get memory each time when instance is created.

### C# Static Field

A field which is declared as static, is called static field. Unlike instance field which gets memory each time whenever you create object, there is only one copy of static field created in the memory. It is shared to all the objects.

It is used to refer the common property of all objects such as rateOfInterest in case of Account, companyName in case of Employee etc.

#### C# static field example

Let's see the simple example of static field in C#.

```

1.    using System;
2.    public class Account
3.    {
4.        public int accno;
5.        public String name;
6.        public static float rateOfInterest=8.8f;
7.        public Account(int accno, String name)
8.        {
9.            this.accno = accno;
10.           this.name = name;
11.        }
12.
13.        public void display()

```

```
14.    {
15.        Console.WriteLine(accno + " " + name + " " + rateOfInterest);
16.    }
17. }
18. class TestAccount{
19.     public static void Main(string[] args)
20.     {
21.         Account a1 = new Account(101, "Sonoo");
22.         Account a2 = new Account(102, "Mahesh");
23.         a1.display();
24.         a2.display();
25.
26.     }
27. }
```

Output:

```
101 Sonoo 8.8
102 Mahesh 8.8
```

### C# static field example 2: changing static field

If you change the value of static field, it will be applied to all the objects.

```
1.      using System;
2.      public class Account
3.      {
4.          public int accno;
5.          public String name;
6.          public static float rateOfInterest=8.8f;
7.          public Account(int accno, String name)
8.          {
9.              this.accno = accno;
10.             this.name = name;
11.         }
```

```
12.  
13.    public void display()  
14.    {  
15.        Console.WriteLine(accno + " " + name + " " + rateOfInterest);  
16.    }  
17.}  
18. class TestAccount{  
19.    public static void Main(string[] args)  
20.    {  
21.        Account.rateOfInterest = 10.5f;//changing value  
22.        Account a1 = new Account(101, "Sonoo");  
23.        Account a2 = new Account(102, "Mahesh");  
24.        a1.display();  
25.        a2.display();  
26.  
27.    }  
28.}
```

Output:

```
101 Sonoo 10.5  
102 Mahesh 10.5
```

### C# static field example 3: Counting Objects

Let's see another example of static keyword in C# which counts the objects.

```
1.  using System;  
2.  public class Account  
3.  {  
4.      public int accno;  
5.      public String name;  
6.      public static int count=0;  
7.      public Account(int accno, String name)  
8.      {
```

```
9.     this.accno = accno;
10.    this.name = name;
11.    count++;
12. }
13.
14. public void display()
15. {
16.     Console.WriteLine(accno + " " + name);
17. }
18. }
19. class TestAccount{
20.     public static void Main(string[] args)
21.     {
22.         Account a1 = new Account(101, "Sonoo");
23.         Account a2 = new Account(102, "Mahesh");
24.         Account a3 = new Account(103, "Ajeet");
25.         a1.display();
26.         a2.display();
27.         a3.display();
28.         Console.WriteLine("Total Objects are: "+Account.count);
29.     }
30. }
```

Output:

```
101 Sonoo
102 Mahesh
103 Ajeet
Total Objects are: 3
```

## C# static class

The C# static class is like the normal class but it cannot be instantiated. It can have only static members. The advantage of static class is that it provides you guarantee that instance of static class cannot be created.

## Points to remember for C# static class

- C# static class contains only static members.
- C# static class cannot be instantiated.
- C# static class is sealed.
- C# static class cannot contain instance constructors.

## C# static class example

Let's see the example of static class that contains static field and static method.

```

1.     using System;
2.     public static class MyMath
3.     {
4.         public static float PI=3.14f;
5.         public static int cube(int n){return n*n*n;}
6.     }
7.     class TestMyMath{
8.         public static void Main(string[] args)
9.         {
10.             Console.WriteLine("Value of PI is: "+MyMath.PI);
11.             Console.WriteLine("Cube of 3 is: " + MyMath(cube(3)));
12.         }
13.     }

```

Output:

```
Value of PI is: 3.14
Cube of 3 is: 27
```

## C# static constructor

C# static constructor is used to initialize static fields. It can also be used to perform any action that is to be performed only once. It is invoked automatically before first instance is created or any static member is referenced.

## Points to remember for C# Static Constructor

- C# static constructor cannot have any modifier or parameter.
- C# static constructor is invoked implicitly. It can't be called explicitly.

## C# Static Constructor example

Let's see the example of static constructor which initializes the static field rateOfInterest in Account class.

```
1.  using System;
2.  public class Account
3.  {
4.      public int id;
5.      public String name;
6.      public static float rateOfInterest;
7.      public Account(int id, String name)
8.      {
9.          this.id = id;
10.         this.name = name;
11.     }
12.     static Account()
13.     {
14.         rateOfInterest = 9.5f;
15.     }
16.     public void display()
17.     {
18.         Console.WriteLine(id + " " + name + " " + rateOfInterest);
19.     }
20. }
21. class TestEmployee{
22.     public static void Main(string[] args)
23.     {
```

```

24.     Account a1 = new Account(101, "Sonoo");
25.     Account a2 = new Account(102, "Mahesh");
26.     a1.display();
27.     a2.display();
28.
29. }
30. }
```

Output:

```
101 Sonoo 9.5
102 Mahesh 9.5
```

## C# Structs

In C#, classes and structs are blueprints that are used to create instance of a class. Structs are used for lightweight objects such as Color, Rectangle, Point etc.

Unlike class, structs in C# are value type than reference type. It is useful if you have data that is not intended to be modified after creation of struct.

### C# Struct Example

Let's see a simple example of struct Rectangle which has two data members width and height.

```

1. using System;
2. public struct Rectangle
3. {
4.     public int width, height;
5.
6. }
7. public class TestStructs
8. {
9.     public static void Main()
10.    {
11.        Rectangle r = new Rectangle();
12.        r.width = 4;
```

```
13.     r.height = 5;
14.     Console.WriteLine("Area of Rectangle is: " + (r.width * r.height));
15. }
16. }
```

Output:

```
Area of Rectangle is: 20
```

### C# Struct Example: Using Constructor and Method

Let's see another example of struct where we are using constructor to initialize data and method to calculate area of rectangle.

```
1. using System;
2. public struct Rectangle
3. {
4.     public int width, height;
5.
6.     public Rectangle(int w, int h)
7.     {
8.         width = w;
9.         height = h;
10.    }
11.    public void areaOfRectangle() {
12.        Console.WriteLine("Area of Rectangle is: "+(width*height)); }
13.    }
14.    public class TestStructs
15.    {
16.        public static void Main()
17.        {
18.            Rectangle r = new Rectangle(5, 6);
19.            r.areaOfRectangle();
20.        }
21.    }
22. }
```

21. }

Output:

Area of Rectanale is: 30

## C# Enum

Enum in C# is also known as enumeration. It is used to store a set of named constants such as season, days, month, size etc. The enum constants are also known as enumerators. Enum in C# can be declared within or outside class and structs.

Enum constants has default values which starts from 0 and incremented to one by one. But we can change the default value.

### Points to remember

- enum has fixed set of constants ○
- enum improves type safety ○
- enum can be traversed

### C# Enum Example

Let's see a simple example of C# enum.

```

1. using System;
2. public class EnumExample
3. {
4.     public enum Season { WINTER, SPRING, SUMMER, FALL }
5.
6.     public static void Main()
7.     {
8.         int x = (int)Season.WINTER;
9.         int y = (int)Season.SUMMER;
10.        Console.WriteLine("WINTER = {0}", x);
11.        Console.WriteLine("SUMMER = {0}", y);
12.    }

```

13. }

Output:

```
WINTER = 0
SUMMER = 2
```

### C# enum example changing start index

```
1. using System;
2. public class EnumExample
3. {
4.     public enum Season { WINTER=10, SPRING, SUMMER, FALL }
5.
6.     public static void Main()
7.     {
8.         int x = (int)Season.WINTER;
9.         int y = (int)Season.SUMMER;
10.        Console.WriteLine("WINTER = {0}", x);
11.        Console.WriteLine("SUMMER = {0}", y);
12.    }
13. }
```

Output:

```
WINTER = 10
SUMMER = 12
```

### C# enum example for Days

```
1. using System;
2. public class EnumExample
3. {
4.     public enum Days { Sun, Mon, Tue, Wed, Thu, Fri, Sat };
5.
6.     public static void Main()
7.     {
```

```
8. int x = (int)Days.Sun;
9. int y = (int)Days.Mon;
10. int z = (int)Days.Sat;
11. Console.WriteLine("Sun = {0}", x);
12. Console.WriteLine("Mon = {0}", y);
13. Console.WriteLine("Sat = {0}", z);
14. }
15. }
```

Output:

```
Sun = 0
Mon = 1
Sat = 6
```

### C# enum example: traversing all values using getNames()

```
1. using System;
2. public class EnumExample
3. {
4.     public enum Days { Sun, Mon, Tue, Wed, Thu, Fri, Sat };
5.
6.     public static void Main()
7.     {
8.         foreach (string s in Enum.GetNames(typeof(Days)))
9.         {
10.             Console.WriteLine(s);
11.         }
12.     }
13. }
```

Output:

```
Sun  
Mon  
Tue  
Wed  
Thu  
Fri  
Sat
```

C# enum example: traversing all values using getValues()

```
1. using System;  
2. public class EnumExample  
3. {  
4.     public enum Days { Sun, Mon, Tue, Wed, Thu, Fri, Sat };  
5.  
6.     public static void Main()  
7.     {  
8.         foreach (Days d in Enum.GetValues(typeof(Days)))  
9.         {  
10.             Console.WriteLine(d);  
11.         }  
12.     }  
13. }
```

Output:

```
Sun  
Mon  
Tue  
Wed  
Thu  
Fri  
Sat
```

## C# Inheritance

### C# Inheritance

In C#, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which is defined in other class.

In C#, the class which inherits the members of another class is called **derived class** and the class whose members are inherited is called **base** class. The derived class is the specialized class for the base class.

### Advantage of C# Inheritance

**Code reusability:** Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

### C# Single Level Inheritance Example: Inheriting Fields

When one class inherits another class, it is known as single level inheritance. Let's see the example of single level inheritance which inherits the fields only.

```

1.      using System;
2.      public class Employee
3.      {
4.          public float salary = 40000;
5.      }
6.      public class Programmer: Employee
7.      {
8.          public float bonus = 10000;
9.      }
10.     class TestInheritance{
11.         public static void Main(string[] args)
12.         {
13.             Programmer p1 = new Programmer();
14.
15.             Console.WriteLine("Salary: " + p1.salary); 16.
Console.WriteLine("Bonus: " + p1.bonus);
17.

```

```
18.    }
19. }
```

Output:

```
Salary: 40000
Bonus: 10000
```

In the above example, Employee is the **base** class and Programmer is the **derived** class.

### C# Single Level Inheritance Example: Inheriting Methods

Let's see another example of inheritance in C# which inherits methods only.

```
1.  using System;
2.  public class Animal
3.  {
4.      public void eat() { Console.WriteLine("Eating..."); }
5.  }
6.  public class Dog: Animal
7.  {
8.      public void bark() { Console.WriteLine("Barking..."); } 9.  }
10. class TestInheritance2{
11.     public static void Main(string[] args)
12.     {
13.         Dog d1 = new Dog();
14.         d1.eat();
15.         d1.bark();
16.     }
17. }
```

Output:

```
Eating...
Barking...
```

### C# Multi Level Inheritance Example

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C#. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

Let's see the example of multi level inheritance in C#.

```
1.  using System;
2.  public class Animal
3.  {
4.      public void eat() { Console.WriteLine("Eating..."); }
5.  }
6.  public class Dog: Animal
7.  {
8.      public void bark() { Console.WriteLine("Barking..."); } 9.  }
10.     public class BabyDog : Dog
11.     {
12.         public void weep() { Console.WriteLine("Weeping..."); }
13.     }
14.     class TestInheritance2{
15.         public static void Main(string[] args)
16.         {
17.             BabyDog d1 = new BabyDog();
18.             d1.eat();
19.             d1.bark();
20.             d1.weep();
21.         }
22.     }
```

Output:

```
Eating...
Barking...
Weeping...
```

## C# Aggregation (HAS-A Relationship)

In C#, aggregation is a process in which one class defines another class as any entity reference. It is another way to reuse the class. It is a form of association that represents HAS-A relationship.

### C# Aggregation Example

Let's see an example of aggregation where Employee class has the reference of Address class as data member. In such way, it can reuse the members of Address class.

```
1. using System;
2. public class Address
3. {
4.     public string addressLine, city, state;
5.     public Address(string addressLine, string city, string state) 6.    {
7.         this.addressLine = addressLine;
8.         this.city = city;
9.         this.state = state;
10.    }
11. }
12. public class Employee
13. {
14.     public int id;
15.     public string name;
16.     public Address address;//Employee HAS-A Address
17.     public Employee(int id, string name, Address address)
18.    {
19.        this.id = id;
20.        this.name = name;
21.        this.address = address;
22.    }
23.     public void display()
```

```

24.    {
25.        Console.WriteLine(id + " " + name + " " +
26.            address.addressLine + " " + address.city + " " + address.state);
27.    }
28. }
29. public class TestAggregation
30. {
31.     public static void Main(string[] args)
32.     {
33.         Address a1=new Address("G-13, Sec-3","Noida","UP");
34.         Employee e1 = new Employee(1,"Sonoo",a1);
35.         e1.display();
36.     }
37. }
```

Output:

```
1 Sonoo G-13 Sec-3 Noida UP
```

## **C# Polymorphism**

### **C# Member Overloading**

If we create two or more members having same name but different in number or type of parameter, it is known as member overloading. In C#, we can overload:

- o methods,
- o constructors, and
- o indexed properties

It is because these members have parameters only.

### **C# Method Overloading**

Having two or more methods with same name but different in parameters, is known as method overloading in C#.

The **advantage** of method overloading is that it increases the readability of the program because you don't need to use different names for same action.

You can perform method overloading in C# by two ways:

1. By changing number of arguments
2. By changing data type of the arguments

### C# Method Overloading Example: By changing no. of arguments

Let's see the simple example of method overloading where we are changing number of arguments of add() method.

```
1. using System;
2. public class Cal{
3.     public static int add(int a,int b){
4.         return a + b;
5.     }
6.     public static int add(int a, int b, int c)
7.     {
8.         return a + b + c;
9.     }
10.    }
11.    public class TestMemberOverloading
12.    {
13.        public static void Main()
14.        {
15.            Console.WriteLine(Cal.add(12, 23));
16.            Console.WriteLine(Cal.add(12, 23, 25));
17.        }
18.    }
```

Output:

35  
60

### C# Member Overloading Example: By changing data type of arguments

Let's see the another example of method overloading where we are changing data type of arguments.

```
1.  using System;
2.  public class Cal{
3.      public static int add(int a, int b){
4.          return a + b;
5.      }
6.      public static float add(float a, float b)
7.      {
8.          return a + b;
9.      }
10. }
11. public class TestMemberOverloading
12. {
13.     public static void Main()
14.     {
15.         Console.WriteLine(Cal.add(12, 23));
16.         Console.WriteLine(Cal.add(12.4f, 21.3f));
17.     }
18. }
```

Output:

35  
33.7

## C# Method Overriding

If derived class defines same method as defined in its base class, it is known as method overriding in C#. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the method which is already provided by its base class.

To perform method overriding in C#, you need to use **virtual** keyword with base class method and **override** keyword with derived class method.

### C# Method Overriding Example

Let's see a simple example of method overriding in C#. In this example, we are overriding the eat() method by the help of override keyword.

```
1.  using System;
2.  public class Animal{
3.      public virtual void eat(){
4.          Console.WriteLine("Eating... ");
5.      }
6.  }
7.  public class Dog: Animal
8.  {
9.      public override void eat()
10.     {
11.         Console.WriteLine("Eating bread... ");
12.     }
13. }
14. public class TestOverriding
15. {
16.     public static void Main()
17.     {
18.         Dog d = new Dog();
19.         d.eat();
20.     }
21. }
```

Output:

```
Eating bread...
```

## C# Polymorphism

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word. In object-oriented programming, we use 3 main concepts: inheritance, encapsulation and polymorphism.

There are two types of polymorphism in C#: compile time polymorphism and runtime polymorphism. Compile time polymorphism is achieved by method overloading and operator overloading in C#. It is also known as static binding or early binding. Runtime polymorphism is achieved by method overriding which is also known as dynamic binding or late binding.

### C# Runtime Polymorphism Example

Let's see a simple example of runtime polymorphism in C#.

```
1.  using System;
2.  public class Animal{
3.      public virtual void eat(){
4.          Console.WriteLine("eating...");
5.      }
6.  }
7.  public class Dog: Animal
8.  {
9.      public override void eat()
10.     {
11.         Console.WriteLine("eating bread...");
12.     }
13. }
14. }
15. public class TestPolymorphism
16. {
17.     public static void Main()
```

```
18.    {
19.        Animal a= new Dog();
20.        a.eat();
21.    }
22. }
```

Output:

```
eating bread...
```

## C# Runtime Polymorphism Example 2

Let's see another example of runtime polymorphism in C# where we are having two derived classes.

```
1.  using System;
2.  public class Shape{
3.      public virtual void draw(){
4.          Console.WriteLine("drawing...");
5.      }
6.  }
7.  public class Rectangle: Shape
8.  {
9.      public override void draw()
10.     {
11.         Console.WriteLine("drawing rectangle...");
12.     }
13.
14. }
15. public class Circle : Shape
16. {
17.     public override void draw()
18.     {
19.         Console.WriteLine("drawing circle...");
```

```
20.    }
21.
22.    }
23.    public class TestPolymorphism
24.    {
25.        public static void Main()
26.        {
27.            Shape s;
28.            s = new Shape();
29.            s.draw();
30.            s = new Rectangle();
31.            s.draw();
32.            s = new Circle();
33.            s.draw();
34.
35.        }
36.    }
```

Output:

```
drawing...
drawing rectanale...
drawing circle...
```

### Runtime Polymorphism with Data Members

Runtime Polymorphism can't be achieved by data members in C#. Let's see an example where we are accessing the field by reference variable which refers to the instance of derived class.

```
1.    using System;
2.    public class Animal{
3.        public string color = "white";
4.
5.    }
6.    public class Dog: Animal
```

```

7.  {
8.  public string color = "black";
9.  }
10. public class TestSealed
11. {
12. public static void Main()
13. {
14. Animal d = new Dog(); 15.     Console.WriteLine(d.color);
16.
17. }
18. }
```

Output:

white

## C# Abstraction

## C# Abstract

Abstract classes are the way to achieve abstraction in C#. Abstraction in C# is the process to hide the internal details and showing functionality only. Abstraction can be achieved by two ways:

1. Abstract class
2. Interface

Abstract class and interface both can have abstract methods which are necessary for abstraction.

### Abstract Method

A method which is declared abstract and has no body is called abstract method. It can be declared inside the abstract class only. Its implementation must be provided by derived classes. For example:

1. **public abstract void** draw();

**An abstract method in C# is internally a virtual method so it can be overridden by the derived class.**

You can't use static and virtual modifiers in abstract method declaration.

## C# Abstract class

In C#, abstract class is a class which is declared abstract. It can have abstract and nonabstract methods. It cannot be instantiated. Its implementation must be provided by derived classes. Here, derived class is forced to provide the implementation of all the abstract methods.

Let's see an example of abstract class in C# which has one abstract method draw(). Its implementation is provided by derived classes: Rectangle and Circle. Both classes have different implementation.

```
1.  using System;
2.  public abstract class Shape
3.  {
4.      public abstract void draw();
5.  }
6.  public class Rectangle : Shape
7.  {
8.      public override void draw()
9.      {
10.         Console.WriteLine("drawing rectangle...");
11.     }
12. }
13. public class Circle : Shape
14. {
15.     public override void draw()
16.     {
17.         Console.WriteLine("drawing circle...");
18.     }
19. }
```

```
20. public class TestAbstract  
21. {  
22.     public static void Main()  
23.     {  
24.         Shape s;  
25.         s = new Rectangle();  
26.         s.draw();  
27.         s = new Circle();  
28.         s.draw();  
29.     }  
30. }
```

Output:

```
drawing ractangle... drawing  
circle...
```

## C# Interface

Interface in C# is a blueprint of a class. It is like abstract class because all the methods which are declared inside the interface are abstract methods. It cannot have method body and cannot be instantiated.

It is used *to achieve multiple inheritance* which can't be achieved by class. It is used *to achieve fully abstraction* because it cannot have method body.

Its implementation must be provided by class or struct. The class or struct which implements the interface, must provide the implementation of all the methods declared inside the interface.

### C# interface example

Let's see the example of interface in C# which has draw() method. Its implementation is provided by two classes: Rectangle and Circle.

ADVERTISEMENT

```

1.  using System;
2.  public interface Drawable
3.  {
4.      void draw();
5.  }
6.  public class Rectangle : Drawable
7.  {
8.      public void draw()
9.      {
10.         Console.WriteLine("drawing rectangle...");
11.     }
12. }
13. public class Circle : Drawable
14. {
15.     public void draw()
16.     {

```

```

17.     Console.WriteLine("drawing circle...");
18. }
19. }
20. public class TestInterface
21. {
22.     public static void Main()
23.     {
24.         Drawable d;
25.         d = new Rectangle();
26.         d.draw();
27.         d = new Circle();
28.         d.draw();
29.     }
30. }
```

Output:

```

drawing ractangle...
drawing circle...
```

**Note:** Interface methods are public and abstract by default. You cannot explicitly use public and abstract keywords for an interface method.

```

1. using System;
2. public interface Drawable
3. {
4.     public abstract void draw();//Compile Time Error
5. }
```

## C# Namespace

### C# Namespaces

Namespaces in C# are used to organize too many classes so that it can be easy to handle the application.

In a simple C# program, we use System.Console where System is the namespace and Console is the class. To access the class of a namespace, we need to use

namespacename.classname. We can use **using** keyword so that we don't have to use complete name all the time.

In C#, global namespace is the root namespace. The global::System will always refer to the namespace "System" of .Net Framework.

### C# namespace example

Let's see a simple example of namespace which contains one class "Program".

```

1.   using System;
2.   namespace ConsoleApplication1
3.   {
4.       class Program
5.       {
6.           static void Main(string[] args)
7.           {
8.               Console.WriteLine("Hello Namespace!");
9.           }
10.      }
11.  }
```

Output:

```
Hello Namespace!
```

### C# namespace example: by fully qualified name

Let's see another example of namespace in C# where one namespace program accesses another namespace program.

```

1.   using System;
2.   namespace First {
3.       public class Hello
4.       {
5.           public void sayHello() { Console.WriteLine("Hello First Namespace"); }
6.       }
```

```

7. }
8. namespace Second
9. {
10. public class Hello
11. {
12.     public void sayHello() { Console.WriteLine("Hello Second Namespace"); }
13. }
14. }
15. public class TestNamespace
16. {
17.     public static void Main()
18. {
19.     First.Hello h1 = new First.Hello();
20.     Second.Hello h2 = new Second.Hello();
21.     h1.sayHello();
22.     h2.sayHello();
23.
24. }
25. }
```

Output:

```
Hello First Namespace
Hello Second Namespace
```

### C# namespace example: by using keyword

Let's see another example of namespace where we are using "using" keyword so that we don't have to use complete name for accessing a namespace program.

1. **using** System;
2. **using** First;
3. **using** Second;
4. **namespace** First {
5. **public class** Hello

```
6.    {
7.        public void sayHello() { Console.WriteLine("Hello Namespace"); }
8.    }
9. }
10. namespace Second
11. {
12.     public class Welcome
13.     {
14.         public void sayWelcome() { Console.WriteLine("Welcome Namespace"); }
15.     }
16. }
17. public class TestNamespace
18. {
19.     public static void Main()
20.     {
21.         Hello h1 = new Hello();
22.         Welcome w1 = new Welcome();
23.         h1.sayHello();
24.         w1.sayWelcome();
25.     }
26. }
```

Output:

```
Hello Namespace Welcome
Namespace
```

## C# Access Modifiers / Specifiers

C# Access modifiers or specifiers are the keywords that are used to specify accessibility or scope of variables and functions in the C# application.

C# provides five types of access specifiers.

1. Public

Access Specifier	Description
Public	It specifies that access is not restricted.
Protected	It specifies that access is limited to the containing class or in derived class.
Internal	It specifies that access is limited to the current assembly.
protected internal	It specifies that access is limited to the current assembly or types derived from
Private	It specifies that access is limited to the containing type.

2. Protected
3. Internal
4. Protected internal
5. Private

We can choose any of these to protect our data. Public is not restricted and Private is most restricted. The following table describes about the accessibility of each.

Now, let's create examples to check accessibility of each access specifier.

### 1) C# Public Access Specifier

It makes data accessible publicly. It does not restrict data to the declared block.

#### Example

1. **using** System;
2. **namespace** AccessSpecifiers
3. {

```

4.     class PublicTest
5.     {
6.         public string name = "Shantosh Kumar";
7.         public void Msg(string msg)
8.         {
9.             Console.WriteLine("Hello " + msg);
10.        }
11.    }
12.    class Program
13.    {
14.        static void Main(string[] args)
15.        {
16.            PublicTest publicTest = new PublicTest();
17.            // Accessing public variable
18.            Console.WriteLine("Hello " + publicTest.name);
19.            // Accessing public function
20.            publicTest.Msg("Peter Decosta");
21.        }
22.    }
23. }
```

**Output:**

```
Hello Shantosh Kumar
Hello Peter Decosta
```

**2) C# Protected Access Specifier**

It is accessible within the class and has limited scope. It is also accessible within sub class or child class, in case of inheritance.

**Example**

```

1.     using System;
2.     namespace AccessSpecifiers
3.     {
```

```

4.     class ProtectedTest
5.     {
6.         protected string name = "Shashikant";
7.         protected void Msg(string msg)
8.     {
9.         Console.WriteLine("Hello " + msg);
10.    }
11. }
12. class Program
13. {
14.     static void Main(string[] args)
15.     {
16.         ProtectedTest protectedTest = new ProtectedTest();
17.         // Accessing protected variable
18.         Console.WriteLine("Hello " + protectedTest.name);
19.         // Accessing protected function
20.         protectedTest.Msg("Swami Ayyer");
21.     }
22. }
23. }
```

**Output:**

Compile time error

'ProtectedTest.name' is inaccessible due to its protection level. **Example2**

Here, we are accessing protected members within child class by inheritance.

```

1.     using System;
2.     namespace AccessSpecifiers
3.     {
4.         class ProtectedTest
5.         {
6.             protected string name = "Shashikant";
```

```

7.     protected void Msg(string msg)
8.     {
9.         Console.WriteLine("Hello " + msg);
10.    }
11.   }
12.   class Program : ProtectedTest
13.   {
14.       static void Main(string[] args)
15.       {
16.           Program program = new Program();
17.           // Accessing protected variable
18.           Console.WriteLine("Hello " + program.name);
19.           // Accessing protected function
20.           program.Msg("Swami Ayyer");
21.       }
22.   }
23. }
```

**Output:**

```
Hello Shashikant
Hello Swami Ayyer
```

**3) C# Internal Access Specifier**

The internal keyword is used to specify the internal access specifier for the variables and functions. This specifier is accessible only within files in the same assembly.

**Example**

```

1.     using System;
2.     namespace AccessSpecifiers
3.     {
4.         class InternalTest
5.         {
6.             internal string name = "Shantosh Kumar";
```

```

7.     internal void Msg(string msg)
8.     {
9.         Console.WriteLine("Hello " + msg);
10.    }
11.   }
12.   class Program
13.   {
14.       static void Main(string[] args)
15.       {
16.           InternalTest internalTest = new InternalTest();
17.           // Accessing internal variable
18.           Console.WriteLine("Hello " + internalTest.name);
19.           // Accessing internal function
20.           internalTest.Msg("Peter Decosta");
21.       }
22.   }
23. }
```

**Output:**

```
Hello Shantosh Kumar
Hello Peter Decosta
```

**4) C# Protected Internal Access Specifier**

Variable or function declared **protected internal** can be accessed in the assembly in which it is declared. It can also be accessed within a derived class in another assembly.

**Example**

```

1.     using System;
2.     namespace AccessSpecifiers
3.     {
4.         class InternalTest
5.         {
6.             protected internal string name = "Shantosh Kumar";
```

```

7.     protected internal void Msg(string msg)
8.     {
9.         Console.WriteLine("Hello " + msg);
10.    }
11.   }
12.   class Program
13.   {
14.       static void Main(string[] args)
15.       {
16.           InternalTest internalTest = new InternalTest();
17.           // Accessing protected internal variable
18.           Console.WriteLine("Hello " + internalTest.name);
19.           // Accessing protected internal function
20.           internalTest.Msg("Peter Decosta");
21.       }
22.   }
23. }
```

**Output:**

Hello Shantosh Kumar  
Hello Peter Decosta

**5) C# Private Access Specifier**

Private Access Specifier is used to specify private accessibility to the variable or function. It is most restrictive and accessible only within the body of class in which it is declared.

**Example**

```

1.     using System;
2.     namespace AccessSpecifiers
3.     {
4.         class PrivateTest
5.         {
6.             private string name = "Shantosh Kumar";
```

```

7.     private void Msg(string msg)
8.     {
9.         Console.WriteLine("Hello " + msg);
10.    }
11.   }
12.   class Program
13.   {
14.       static void Main(string[] args)
15.       {
16.           PrivateTest privateTest = new PrivateTest();
17.           // Accessing private variable
18.           Console.WriteLine("Hello " + privateTest.name);
19.           // Accessing private function
20.           privateTest.Msg("Peter Decosta");
21.       }
22.   }
23. }
```

**Output:**

```
Compile time error
'PrivateTest.name' is inaccessible due to its protection level.
```

**C# Private Specifier Example 2**

```

1.     using System;
2.     namespace AccessSpecifiers
3.     {
4.         class Program
5.         {
6.             private string name = "Shantosh Kumar";
7.             private void Msg(string msg)
8.             {
9.                 Console.WriteLine("Hello " + msg);
```

```

10.    }
11.    static void Main(string[] args)
12.    {
13.        Program program = new Program();
14.        // Accessing private variable
15.        Console.WriteLine("Hello " + program.name);
16.        // Accessing private function
17.        program.Msg("Peter Decosta");
18.    }
19. }
20. }
```

**Output:**

```
Hello Shantosh Kumar
Hello Peter Decosta
```

## C# Encapsulation

Encapsulation is the concept of wrapping data into a single unit. It collects data members and member functions into a single unit called class. The purpose of encapsulation is to prevent alteration of data from outside. This data can only be accessed by getter functions of the class.

A fully encapsulated class has getter and setter functions that are used to read and write data. This class does not allow data access directly.

Here, we are creating an example in which we have a class that encapsulates properties and provides getter and setter functions.

### Example

```

1.    namespace AccessSpecifiers
2.    {
3.        class Student
4.        {
5.            // Creating setter and getter for each property
6.            public string ID { get; set; }
```

```

7. public string Name { get; set; }
8. public string Email { get; set; }
9. }
10. }

1. using System;
2. namespace AccessSpecifiers
3. {
4. class Program
5. {
6. static void Main(string[] args)
7. {
8. Student student = new Student();
9. // Setting values to the properties
10. student.ID = "101";
11. student.Name = "Mohan Ram";
12. student.Email = "mohan@example.com";
13. // getting values
14. Console.WriteLine("ID = "+student.ID);
15. Console.WriteLine("Name = "+student.Name);
16. Console.WriteLine("Email = "+student.Email);
17. }
18. }
19. }

```

**Output:**

```

ID = 101
Name = Mohan Ram
Email = mohan@example.com

```

## C# Strings

In C#, string is an object of **System.String** class that represent sequence of characters. We can perform many operations on strings such as concatenation, comparision, getting substring, search, trim, replacement etc.

## String vs String

In C#, *string* is keyword which is an alias for *System.String* class. That is why string and String are equivalent. We are free to use any naming convention.

1. **string** s1 = "hello";//creating string using string keyword
2. String s2 = "welcome";//creating string using String class **C#**

### String Example

```

1.   using System;
2.   public class StringExample
3.   {
4.     public static void Main(string[] args)
5.     {
6.       string s1 = "hello";
7.
8.       char[] ch = { 'c', 's', 'h', 'a', 'r', 'p' };
9.       string s2 = new string(ch);
10.
11.      Console.WriteLine(s1);
12.      Console.WriteLine(s2);
13.    }
14.  }

```

Output:

```
hello
csharp
```

## C# Exception Handling

Exception Handling in C# 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.

In C#, exception is an event or object which is thrown at runtime. All exceptions are 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.

## Advantage

It *maintains the normal flow* of the application. In such case, rest of the code is executed even after exception.

## C# Exception Handling Keywords

In C#, we use 4 keywords to perform exception handling:

- try
- catch
- finally, and
- throw

### C# try/catch

In C# programming, exception handling is performed by try/catch statement. The **try block** in C# is used to place the code that may throw exception. The **catch block** is used to handle the exception. The catch block must be preceded by try block.

### C# example without try/catch

```

1.  using System;
2.  public class ExExample
3.  {
4.      public static void Main(string[] args)
5.      {
6.          int a = 10;
7.          int b = 0;
8.          int x = a/b;
9.          Console.WriteLine("Rest of the code");
10.     }
11. }
```

Output:

```
Unhandled Exception: System.DivideByZeroException: Attempted to divide
by zero.
```

## C# try/catch example

```

1.  using System;
2.  public class ExExample
3.  {
4.      public static void Main(string[] args)
5.      {
6.          try
7.          {
8.              int a = 10;
9.              int b = 0;
10.             int x = a / b;
11.         }
12.         catch (Exception e) { Console.WriteLine(e); }
13.
14.         Console.WriteLine("Rest of the code");
15.     }
16. }
```

Output:

```
System.DivideByZeroException: Attempted to divide by zero.
Rest of the code
```

## C# finally

C# 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.

### C# finally example if exception is handled

```

1.  using System;
2.  public class ExExample
3.  {
4.      public static void Main(string[] args)
5.      {
```

```
6.     try
7.     {
8.         int a = 10;
9.         int b = 0;
10.        int x = a / b;
11.    }
12.    catch (Exception e) { Console.WriteLine(e); }
13.    finally { Console.WriteLine("Finally block is executed"); }
14.    Console.WriteLine("Rest of the code");
15. }
16. }
```

Output:

```
System.DivideByZeroException: Attempted to divide by zero.
Finally block is executed
Rest of the code
```

### C# finally example if exception is not handled

```
1.     using System;
2.     public class ExExample
3.     {
4.         public static void Main(string[] args)
5.         {
6.             try
7.             {
8.                 int a = 10;
9.                 int b = 0;
10.                int x = a / b;
11.            }
12.            catch (NullReferenceException e) { Console.WriteLine(e); }
13.            finally { Console.WriteLine("Finally block is executed"); }
14.            Console.WriteLine("Rest of the code");
```

```
15. }
16. }
```

Output:

```
Unhandled Exception: System.DivideBy
```

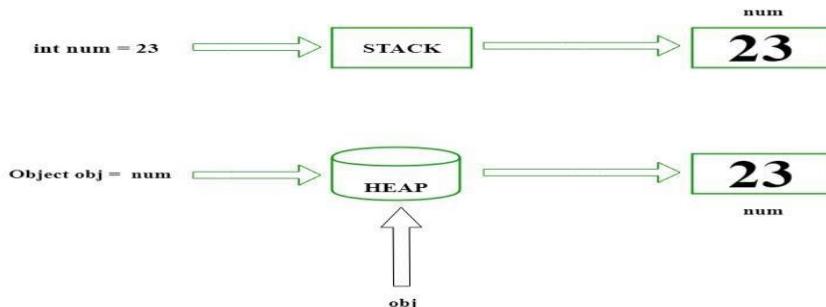
## C# | Boxing and Unboxing

### Boxing in C#:

- The process of converting a **Value Type** variable (**char, int etc.**) to a **Reference Type variable (object)** is called **Boxing**.
- Boxing is an implicit conversion process in which object type (super type) is used.
- Value Type variables are always stored in Stack memory, while Reference Type variables are stored in Heap memory.
- **Example :** `int num = 23; // 23 will assigned to num`

```
Object Obj = num; // Boxing
```

- **Description :** First, we declare a Value Type variable *num* of the type **int** and initialise it with value 23. Now, we create a Reference Type variable *obj* of the type **Object** and assign *num* to it. This assignment implicitly results in the Value Type variable *num* to be copied and stored in Reference Type variable *obj* as shown in below figure :



- Let's understand **Boxing** with a C# programming code :

□ Csharp

```
// C# implementation to demonstrate

// the Boxing

using System;

class GFG //


Main Method

    static public void Main()

    {

// assigned int value

// 2020 to num

int num = 2020;

// boxing      object

obj = num;

// value of num to be change

num = 100;

System.Console.WriteLine

("Value - type value of num is : {0}", num);
```

```

System.Console.WriteLine

("Object - type value of obj is : {0}", obj);

}

}

```

**Output:**

Value - type value of num is : 100

Object - type value of obj is : 2020

**Unboxing In C#**

- The process of converting a [Reference Type](#) variable into a [Value Type](#) variable is known as **Unboxing**.
- It is an explicit conversion process.

□ **Example :**

```

int num = 23;      // value type is int and assigned value 23

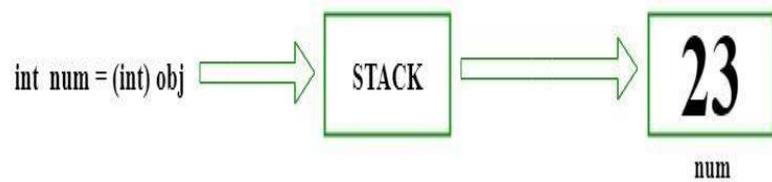
Object Obj = num; // Boxing

int i = (int)Obj; // Unboxing

```

- 
- 
- 

- **Description :** We declare a Value Type variable *num*, which is of the type **int** and assign it with integer value 23. Now, we create a Reference Type variable *obj* of the type **Object**, in which we box the variable *num*. Now, we create a Value Type integer *i* to unbox the value from *obj*. This is done using the casting method, in which we explicitly specify that *obj* must be cast as an **int** value. Thus, the Reference Type variable residing in the heap memory is copied to stack as shown in below figure :



- Let's understand **Unboxing** with a C# programming code :

Csharp

```
// C# implementation to demonstrate  
// the Unboxing  
  
using System; class  
  
GFG {  
  
    // Main Method  
  
    static public void Main()  
  
    {  
  
        // assigned int value  
  
        // 23 to num      int  
  
        num = 23;  
  
        // boxing      object  
  
        obj = num;  
  
        // unboxing  
  
        int i = (int)obj;  
  
        // Display result
```

```

Console.WriteLine("Value of ob object is : " + obj);

Console.WriteLine("Value of i is : " + i);

}
}

```

**Output:**

Value of ob object is : 23

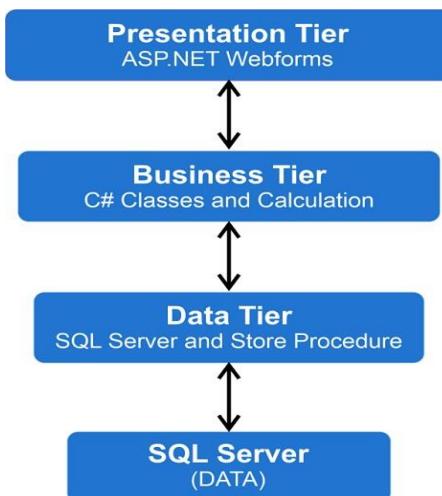
Value of i is : 23

- In cases such as unboxing of a **null** object or casting the object as an incompatible data type, the program throws exceptions.

## **3-tier architecture in c#**

This article briefly discusses the three-tier application in c#. A three-tier architecture divides the project into three layers: the user interfaces layer, the business layer, and the data (database) layer, and separates the UI, logic, and data into three layers.

If a user wants to change her UI from Windows to Phone, she can change the UI layer, and the other layers will not be affected by this change. Otherwise, everything remains the same.



### **Why do we use 3 tier application on any project?**

We use a three-tier structure to control large projects. For instance, let us say you wrote 2000 entity codes and coded all the code in a single layer. Say you want to interchange between home

windows software and cellular or web applications, and you must rewrite the code for all 2000 entities.

You must trade the database connection for all 2000 entities if you change the database provider. Increase. So, writing a lot of code wastes money and time. Additionally, writing numerous codes in layers makes it difficult for new humans to apprehend them.

Therefore, a project should use a three-tier architecture to provide maintenance, flexibility, update, and clean and understandable code without affecting other parts.

So, we use 3-tier architecture because of these points -

1. **Accelerated development:** Different teams can improve each layer simultaneously, optimizing product time-to-market and allowing developers to use the latest tools and the best languages for each layer.
2. **Improved scalability:** Each tier can always be scaled independently by deploying applications in different tiers.
3. **Improved Reliability:** With different tiers, you can also run other parts of your application on different servers and use cached results to enhance reliability and availability.
4. **Enhanced Security:** Leverages a well-designed application layer as an internal firewall that helps prevent SQL injection and other malicious exploits.

## What is the difference between layer and tier?

### Layers:

1. **Definition:** Layers refer to the logical separation of components within the same application or system.
2. **Purpose:** Layers are used to organize and structure different functionalities of an application, such as presentation (UI), business logic, and data access.
3. **Example:** In a typical 3-layer architecture (presentation layer, business logic layer, data access layer), each layer serves a distinct purpose and communicates with adjacent layers to fulfill application tasks.
4. **Interaction:** Layers interact directly with adjacent layers within the same application or system.

### Tiers:

1. **Definition:** Tiers refer to the physical separation of components into distinct parts that may be deployed on separate machines or servers.
2. **Purpose:** Tiers are used for scalability, distribution of workload, and sometimes to enforce security boundaries.
3. **Example:** In a 3-tier architecture, you might have the presentation tier (client-side interface), the application tier (server-side logic), and the data tier (database or external data source).
4. **Interaction:** Tiers interact over a network or some form of communication protocol (e.g., HTTP, TCP/IP), which allows them to communicate across different physical locations or machines.

**Another difference arises from the responsibility of the lowest-level component of each architecture.**

The data access layer does not provide data, so most of the time, the data resides in the application layer (some designs are separated into one layer).

In this view, one level can contain multiple levels. For example, a mobile phone camera application is n-tier. It is also called a single-tier application because all the processes run on the phone.

**Presentation Layer:** The user interacts with the app to capture images.

**Data Access Level:** Apps access device memory to store processed information.

## What are the three parts of the three-tier architecture?

The three-tier architecture includes three tiers, these are -

### 1. Presentation Tier:

- This is where the user interacts with the application.
- In C#, it typically involves creating the user interface (UI) using technologies like Windows Forms, WPF (Windows Presentation Foundation), or ASP.NET for web applications.
- The main goal here is to present data to the user and capture user inputs (like button clicks, text inputs, etc.).

### 3. Business Tier:

- This layer contains the logic that drives the application and processes data based on the user input from the presentation layer.

- In C#, this layer consists of classes and methods that implement business rules, calculations, and validations.
- It acts as a bridge between the presentation layer and the data access layer, ensuring that data is processed correctly according to the business requirements.

#### 4. Data Tier:

- This layer is responsible for interacting with the data storage systems such as databases or external services.
- In C#, it often involves using technologies like ADO.NET, Entity Framework, or other ORM (Object-Relational Mapping) frameworks.
- Its main tasks include querying data, inserting or updating data, and handling transactions with the underlying data storage.

### **Application of 3 tiers using C#:**

This is the heart of the 3-tier architecture and the most complex and difficult implementation layer. So, you need an excellent design to manage and organize your code. Therefore, we used a 3-tier architecture for this tier.

This level does not interact with the user. Interact with other layers/applications. In other words, the presentation layer (application layer) is not the user interface.

### **What are the advantages and drawbacks of 3-tier architecture?**

#### **Advantages:**

1. Data Provider queries can be easily updated, making applying OOP concepts to your projects easy.
2. Updating to the new graphical environment is now easier and faster.

#### **Drawbacks:**

1. Complex and time-consuming to build.
2. This is more complicated than a simple client-server architecture.
3. Users should be familiar with object-oriented concepts.

#### **Example:**

### 1. Presentation Layer (User Interface):

- **Role:** Imagine you have a restaurant with a menu displayed on a digital tablet at each table.
- **Functionality:** Customers use the tablet to browse the menu, select dishes, and place their orders.
- **Technology:** The tablet interface (like a touch screen) is designed using user-friendly buttons and displays to make ordering easy for customers.

### 2. Business Logic Layer:

- **Role:** In the kitchen, the chef receives orders and prepares dishes according to the restaurant's recipes and standards.
- **Implementation:** This layer ensures that orders are processed correctly, ingredients are available, and cooking times are managed efficiently.
- **Example:** If a customer orders a steak, the business logic layer ensures the steak is cooked to the desired doneness and served with appropriate sides.

### 3. Data Access Layer:

- **Role:** Behind the scenes, the restaurant's inventory system tracks ingredient levels and manages stock.
- **Implementation:** This layer communicates with the inventory database to check ingredient availability when an order is placed.
- **Example:** When an order is received for a dish, the data access layer deducts the necessary ingredients from the inventory database to ensure accurate stock levels.

## ADO.NET Introduction

### What is ADO.NET?

ADO.NET (ActiveX Data Objects for .NET) is a technology in the .NET framework that allows developers to interact with data sources such as databases and XML files. It provides a set of classes and libraries for data access, manipulation, and management in applications built using .NET languages like C# and VB.NET.

ADO.NET is powerful for managing data in .NET applications, providing essential tools and flexibility for interacting with various data sources efficiently.

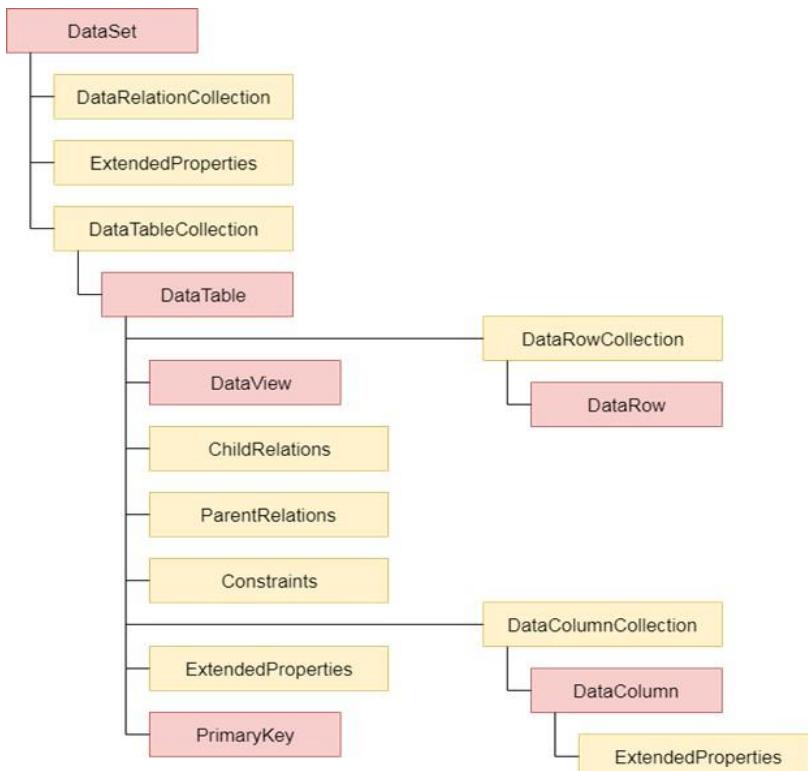
### .NET Framework Data Providers

These are the components that are designed for data manipulation and fast access to data. It provides various objects such as **Connection**, **Command**, **DataReader** and **DataAdapter** that are

used to perform database operations. We will have a detailed discussion about **Data Providers** in new topic.

### The DataSet

It is used to access data independently from any data resource. DataSet contains a collection of one or more DataTable objects of data. The following diagram shows the relationship between .NET Framework data provider and DataSet.



**Fig:** ADO.NET Architecture

Let's break down the architecture and components of ADO.NET in easy wording:

#### 1. Data Providers:

- **Definition:** Data providers in ADO.NET are components that interact with different types of data sources, such as databases (SQL Server, Oracle), XML files, and more.
- **Example:** For SQL Server, you use the `System.Data.SqlClient` namespace; for Oracle, it's `System.Data.OracleClient`.

#### 2. Connection:

- **Definition:** The connection establishes a connection to the data source, enabling communication between the application and the database.

- **Functionality:** It includes details like server address, database name, and authentication credentials.
- **Example:** SqlConnection for SQL Server, OracleConnection for Oracle databases.

### 3. Command:

- **Definition:** Commands (like SqlCommand or OracleCommand) are used to execute SQL queries, stored procedures, or commands against the database.
- **Functionality:** They can retrieve data (SELECT), modify data (INSERT, UPDATE, DELETE), or execute stored procedures.
- **Example:** SqlCommand for SQL Server to execute SQL queries.

### 4. DataReader:

- **Definition:** The DataReader provides a fast, forward-only, read-only stream of data retrieved from the database.
- **Functionality:** It's used for retrieving large volumes of data efficiently, row by row.
- **Example:** SqlDataReader for SQL Server.

### 5. DataAdapter:

- **Definition:** The DataAdapter acts as a bridge between a dataset (in-memory representation of data) and the data source.
- **Functionality:** It fills a dataset with data from the database using commands (SelectCommand, InsertCommand, UpdateCommand, DeleteCommand) and updates changes made to the dataset back to the database. □ **Example:** SqlDataAdapter for SQL Server.

### 6. DataSet:

- **Definition:** A DataSet is an in-memory representation of data retrieved from the data source.
- **Functionality:** It can hold multiple DataTables (representing tabular data) and their relationships.
- **Example:** DataSet containing multiple DataTable objects.

### Workflow:

- **Establish Connection:** Open a connection to the database using a connection string.
- **Execute Commands:** Create and execute commands to retrieve or manipulate data.
- **Retrieve Data:** Use DataReader to fetch data row by row for read-only operations.
- **Manipulate Data (optional):** Modify data using DataSet, DataTables, and DataRow.

- **Update Database:** Use DataAdapter to update changes made to the DataSet back to the database.
- **Close Connection:** Close the connection to release resources.

**Example Scenario:** Imagine building a simple C# application to manage employee information:

- **Connection:** Establish a connection to a SQL Server database where employee details are stored.
- **Command:** Execute SQL queries to retrieve employee information or update employee records.
- **DataReader:** Use a DataReader to display employee details one at a time as users browse through employee profiles.
- **DataAdapter and DataSet:** Use a DataAdapter to fill a DataSet with employee data. Modify the DataSet locally (like updating an employee's contact information) and use the DataAdapter to update these changes back to the database.

## ADO.NET Framework Data Providers

Data provider is used to connect to the database, execute commands and retrieve the record. It is lightweight component with better performance. It also allows us to place the data into DataSet to use it further in our application. [.NET Framework Data Providers Objects](#)

### 1. Connection Object:

- **Role:** Establishes a connection to a database or data source.
- **Example:** SqlConnection for SQL Server, OracleConnection for Oracle databases.

### 2. Command Object:

- **Role:** Executes SQL queries or stored procedures against the data source.
- **Example:** SqlCommand for SQL Server, OracleCommand for Oracle databases.

### 3. DataReader Object:

- **Role:** Retrieves data from the database in a read-only, forward-only manner.
- **Example:** SqlDataReader for SQL Server, OracleDataReader for Oracle databases.

### 4. DataAdapter and DataSet Objects:

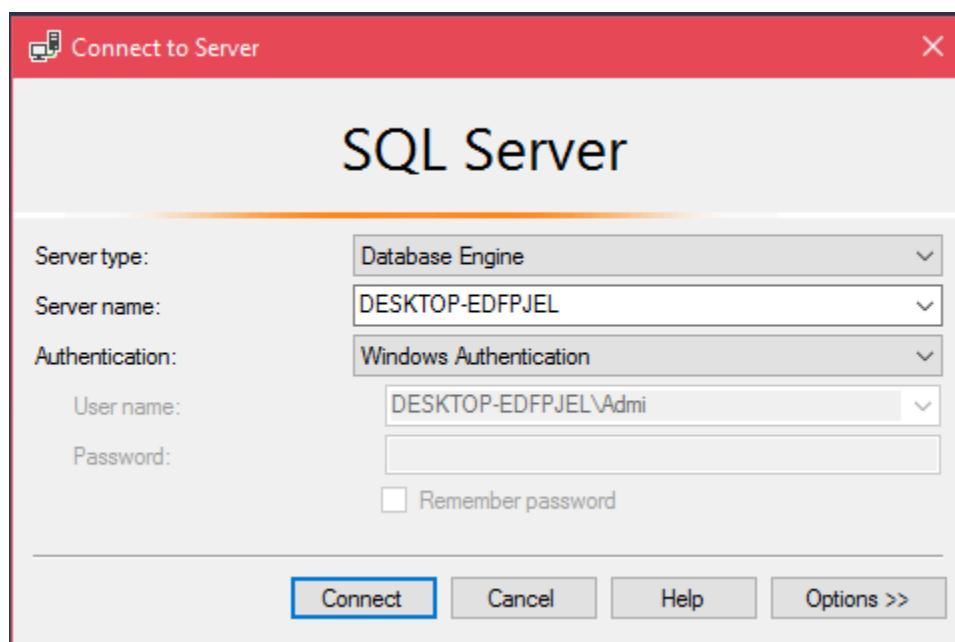
- **Role:** Manages the retrieval and manipulation of data between the application and the database.
- **Example:** SqlDataAdapter for SQL Server, DataSet for storing data in-memory.

## ADO.NET SQL Server Connection

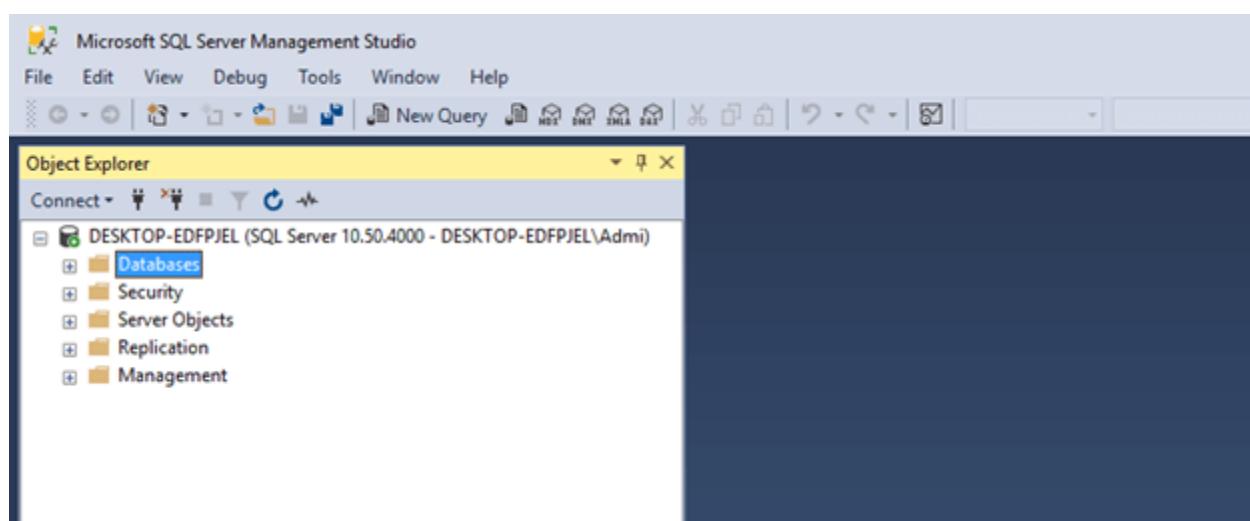
To connect with SQL Server, we must have it installed in our system. We are using Microsoft SQL Server Management Tool to connect with the SQL Server. We can use this tool to handle database. Now, follow the following steps to connect with SQL Server.

### 1. Open Microsoft SQL Server Management Tool

It will prompt for database connection. Provide the server name and authentication.

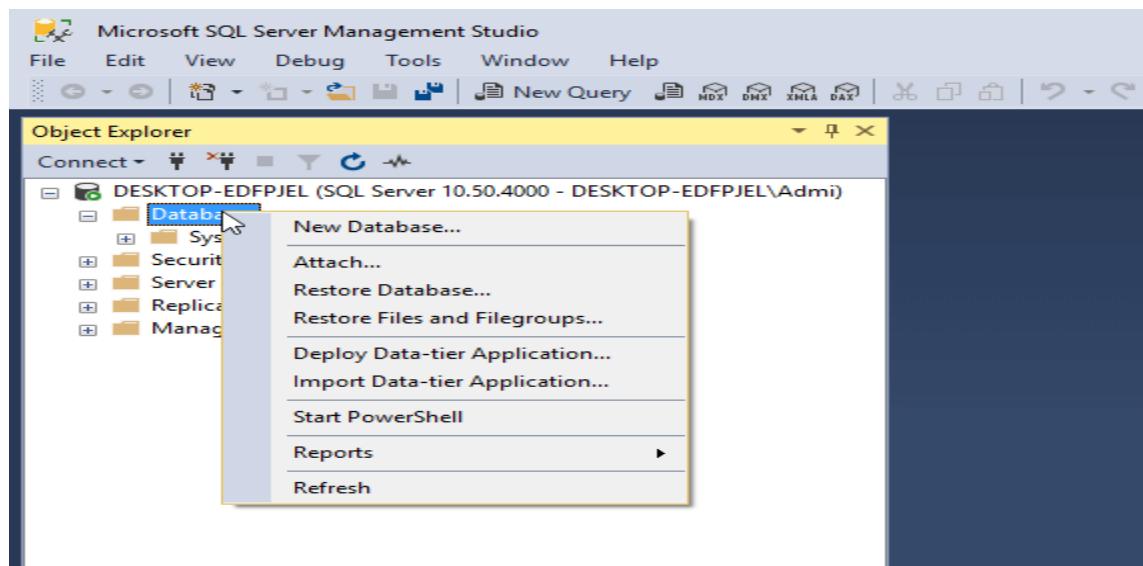


After successful connection, it displays the following window.

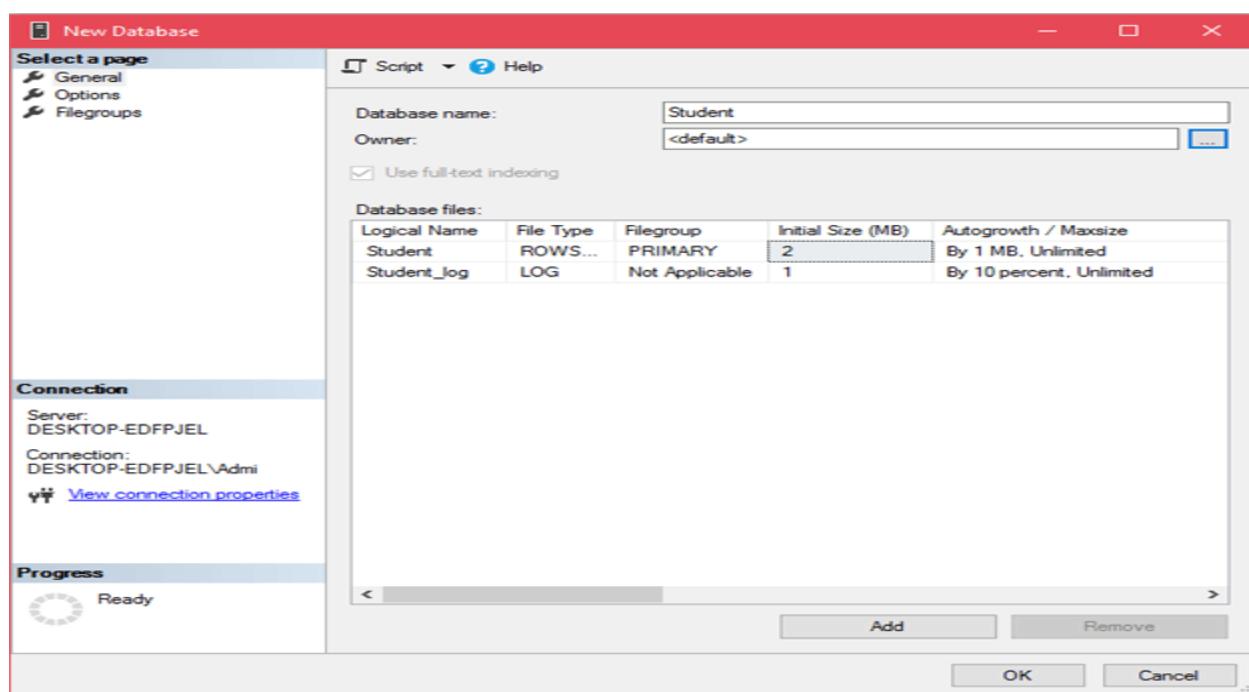


## 2. Creating Database

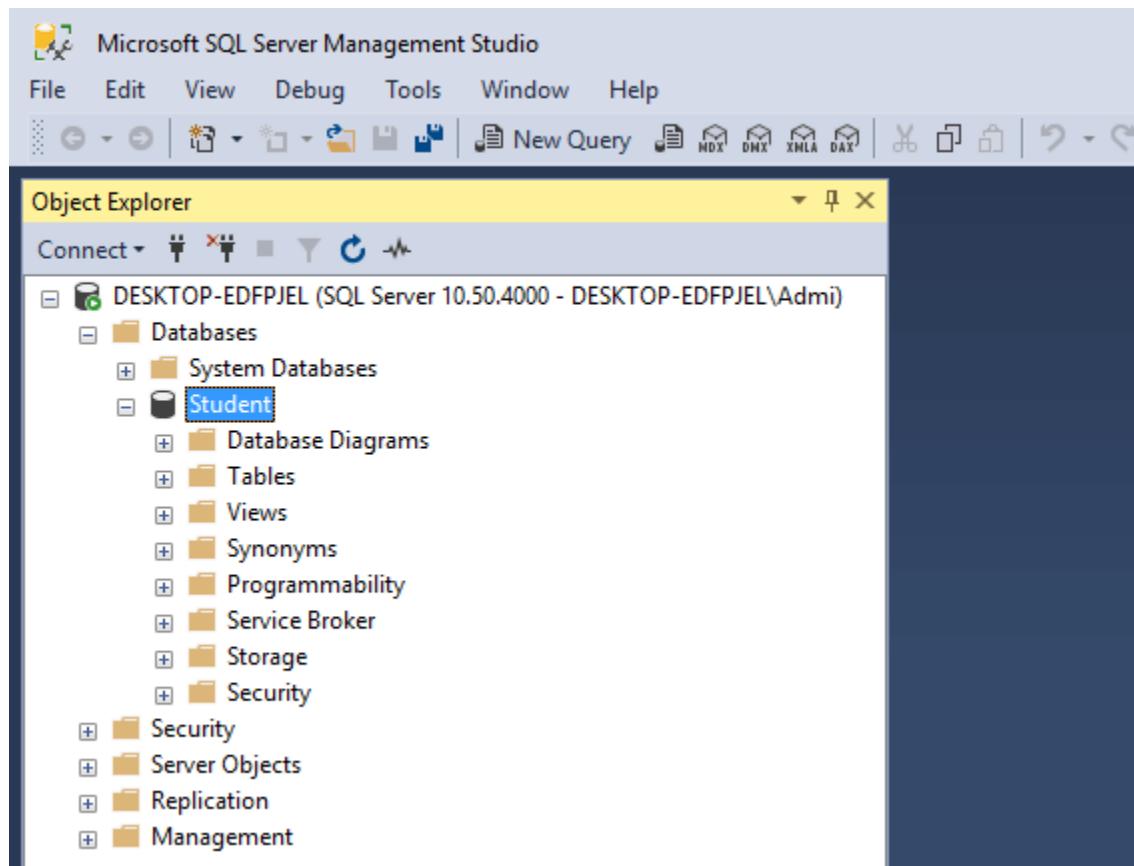
Now, create database by selecting database option then right click on it. It pops up an option menu and provides couple of options.



Click on the **New Database** then it will ask for the database name. Here, we have created a **Student** database.



Click on the Ok button then it will create a database that we can see in the left window of the below screenshot.



### 3. Establish connection and create a table

After creating database, now, let's create a table by using the following C# code. In this source code, we are using created **student** database to connect.

In visual studio 2017, we created a .NET console application project that contains the following C# code.

#### // Program.cs

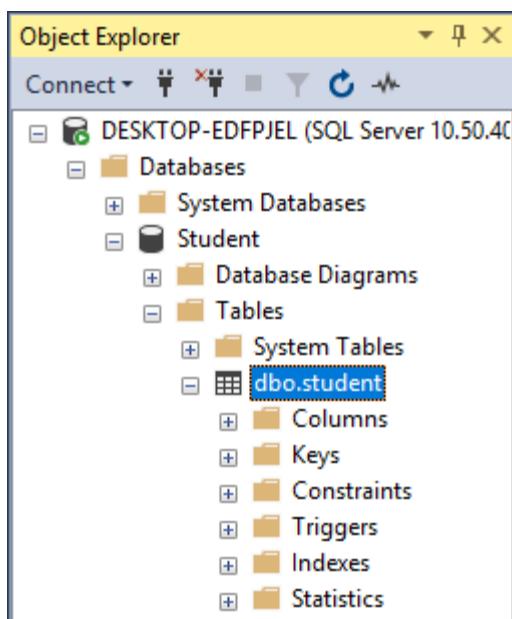
1. **using** System;
2. **using** System.Data.SqlClient;
3. **namespace** AdoNetConsoleApplication
4. {
5.     **class** Program
6.     {

```
7. static void Main(string[] args)
8. {
9.     new Program().CreateTable();
10. }
11. public void CreateTable()
12. {
13.     SqlConnection con = null;
14.     try
15.     {
16.         // Creating Connection
17.         con = new SqlConnection("data source=.; database=student; integrated security=SSPI");
18.         // writing sql query
19.         SqlCommand cm = new SqlCommand("create table student(id int not null,
20.             name varchar(100), email varchar(50), join_date date)", con);
21.         // Opening Connection
22.         con.Open();
23.         // Executing the SQL query
24.         cm.ExecuteNonQuery();
25.         // Displaying a message
26.         Console.WriteLine("Table created Successfully");
27.     }
28.     catch (Exception e)
29.     {
30.         Console.WriteLine("OOPS, something went wrong." + e);
31.     }
32.     // Closing the connection
33.     finally
34.     {
35.         con.Close();
36.     }
37. }
38. }
39. }
```

Execute this code using **Ctrl+F5**. After executing, it displays a message to the console as below.

```
C:\Windows\system32\cmd.exe
Table created Successfully
Press any key to continue . . .
```

We can see the created table in Microsoft SQL Server Management Studio also. It shows the created table as shown below.



See, we have a table here. Initially, this table is empty so we need to insert data into it.

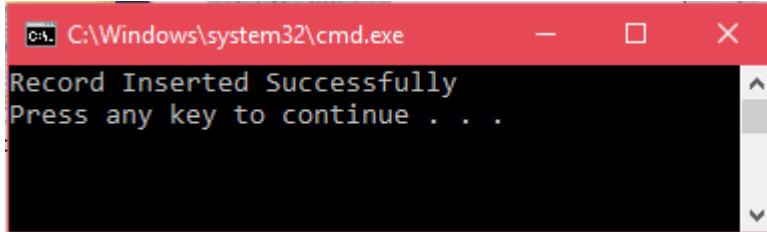
#### 4. Insert Data into the Table

**// Program.cs**

```
1. using System;
2. using System.Data.SqlClient;
3. namespace AdoNetConsoleApplication
4. {
5.     class Program
6.     {
7.         static void Main(string[] args)
8.         {
9.             new Program().CreateTable();
```

```
10.    }
11.    public void CreateTable()
12.    {
13.        SqlConnection con = null;
14.        try
15.        {
16.            // Creating Connection
17.            con = new SqlConnection("data source=.; database=student; integrated security=SSPI");
18.            // writing sql query
19.            SqlCommand cm = new SqlCommand("insert into student
20.                (id, name, email, join_date)values('101','Ronald Trump','ronald@example.com','1/12/2017')", co
n);
21.            // Opening Connection
22.            con.Open();
23.            // Executing the SQL query
24.            cm.ExecuteNonQuery();
25.            // Displaying a message
26.            Console.WriteLine("Record Inserted Successfully");
27.        }
28.        catch (Exception e)
29.        {
30.            Console.WriteLine("OOPS, something went wrong." +e);
31.        }
32.        // Closing the connection
33.        finally
34.        {
35.            con.Close();
36.        }
37.    }
38. }
39. }
```

Execute this code by using **Ctrl+F5** and it will display the following output.



## 5. Retrieve Record

Here, we will retrieve the inserted data. Look at the following C# code.

### // Program.cs

```
1. using System;
2. using System.Data.SqlClient;
3. namespace AdoNetConsoleApplication
4. {
5.     class Program
6.     {
7.         static void Main(string[] args)
8.         {
9.             new Program().CreateTable();
10.        }
11.        public void CreateTable()
12.        {
13.            SqlConnection con = null;
14.            try
15.            {
16.                // Creating Connection
17.                con = new SqlConnection("data source=.; database=student; integrated security=SSPI");
18.                // writing sql query
19.                SqlCommand cm = new SqlCommand("Select * from student", con);
20.                // Opening Connection
21.                con.Open();
22.                // Executing the SQL query
23.                SqlDataReader sdr = cm.ExecuteReader();
24.                // Iterating Data
```

```

25.     while (sdr.Read())
26.     {
27.         Console.WriteLine(sdr["id"] + " " + sdr["name"] + " " + sdr["email"]); // Displaying Record
28.     }
29. }
30. catch (Exception e)
31. {
32.     Console.WriteLine("OOPS, something went wrong.\n" + e);
33. }
34. // Closing the connection
35. finally
36. {
37.     con.Close();
38. }
39. }
40. }
41. }
```

Execute this code by **Ctrl+F5** and it will produce the following result. This displays two records, one we inserted manually.

Output:

```
C:\Windows\system32\cmd.exe
101 Ronald Trump ronald@example.com
102 Somya Bansal somya@example.com
Press any key to continue . . .
```

## 6. Deleting Record

This time **student** table contains two records. The following C# code delete one row from the table.

**// Program.cs**

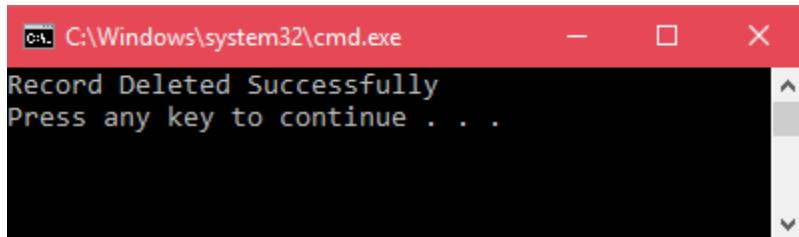
1. **using** System;
2. **using** System.Data.SqlClient;
3. **namespace** AdoNetConsoleApplication

```
4. {
5.     class Program
6.     {
7.         static void Main(string[] args)
8.         {
9.             new Program().CreateTable();
10.        }
11.        public void CreateTable()
12.        {
13.            SqlConnection con = null;
14.            try
15.            {
16.                // Creating Connection
17.                con = new SqlConnection("data source=.; database=student; integrated security=SSPI");
18.                // writing sql query
19.                SqlCommand cm = new SqlCommand("delete from student where id = '101'", con);
20.                // Opening Connection
21.                con.Open();
22.                // Executing the SQL query
23.                cm.ExecuteNonQuery();
24.                Console.WriteLine("Record Deleted Successfully");
25.            }
26.            catch (Exception e)
27.            {
28.                Console.WriteLine("OOPS, something went wrong.\n"+e);
29.            }
30.            // Closing the connection
31.            finally
32.            {
33.                con.Close();
34.            }
35.        }
36.    }
37. }
```

Output:

ADVERTISEMENT

It displays the following output.



```
C:\Windows\system32\cmd.exe
Record Deleted Successfully
Press any key to continue . . .
```

We can verify it by retrieving data back by using SqlDataReader.

## .NET Framework Data Provider for Oracle

It is used to connect with Oracle database through Oracle client. The data provider supports Oracle client software version 8.1.7 or a later version. This data provider supports both local and distributed transactions.

Oracle Data Provider classes are located into **System.Data.OracleClient** namespace. We must use both **System.Data.OracleClient** and **System.data** to connect our application with the Oracle database.

1. using System.Data;
2. using System.Data.OracleClient;

### Which .NET Framework Data Provider is better

Choosing the "better" .NET Framework Data Provider depends on your specific requirements and the data source you are interacting with. Here's a comparison of common .NET Framework Data Providers to help you decide:

#### 1. SqlConnection (for SQL Server):

- **Best For:** Interacting specifically with Microsoft SQL Server databases.
- **Advantages:** Optimized for SQL Server, provides direct integration and efficient data access.
- **Usage:** Ideal when your application primarily interacts with SQL Server databases and requires high performance and reliability.

#### 2. OracleClient (for Oracle databases):

- **Best For:** Interacting with Oracle databases.
- **Advantages:** Provides direct support for Oracle databases, efficient data access capabilities.
- **Usage:** Suitable when your application needs to connect to and work extensively with Oracle databases, maintaining compatibility and performance.

### **3. OleDb (for OLE DB data sources):**

- **Best For:** Interacting with a variety of data sources that support OLE DB.
- **Advantages:** Supports multiple data sources beyond just databases (e.g., Excel files, Access databases), providing versatility.
- **Usage:** Useful when your application needs to interact with diverse data sources that are accessible via OLE DB, though it may not be as optimized as provider-specific options.

### **4. Odbc (for ODBC data sources):**

- **Best For:** Interacting with databases and other data sources that support ODBC.
- **Advantages:** Offers broad compatibility with various data sources through ODBC drivers.
- **Usage:** Helpful when working with legacy systems or databases that only support ODBC connections, but may not offer the same level of performance as provider-specific options.

#### **Choosing the Best Data Provider:**

- **Performance:** Consider the performance requirements specific to your application and data source. Provider-specific options like SqlConnection for SQL Server or OracleClient for Oracle databases are often optimized for best performance.
- **Compatibility:** Ensure compatibility with your target data source. Each provider has specific capabilities and optimizations tailored to certain databases or data sources.

## **ADO.NET SQL Server Connection**

It is used to establish an open connection to the SQL Server database. It is a sealed class so that cannot be inherited. SqlConnection class uses SqlDataAdapter and SqlCommand classes together to increase performance when connecting to a Microsoft SQL Server database.

Connection does not close explicitly even it goes out of scope. Therefore, you must explicitly close the connection by calling Close() method.

## SqlConnection Signature

1. public sealed class SqlConnection : System.Data.Common.DbConnection, ICloneable, IDisposable
- ### SqlConnection Constructors

Constructors	Description
SqlConnection()	It is used to initializes a new instance of the SqlConnection class.
SqlConnection(String)0	It is used to initialize a new instance of the SqlConnection class and takes connection string as an argument.
SqlConnection(String, SqlCredential)	It is used to initialize a new instance of the SqlConnection class that takes two parameters. First is connection string and second is sql credentials.

### SqlConnection Methods

Method	Description
BeginTransaction()	It is used to start a database transaction.
ChangeDatabase(String)	It is used to change the current database for an open SqlConnection.
ChangePassword(String, String)	It changes the SQL Server password for the user indicated in the connection string.
Close()	It is used to close the connection to the database.
CreateCommand()	It enlists in the specified transaction as a distributed transaction.
GetSchema()	It returns schema information for the data source of this SqlConnection.
Open()	It is used to open a database connection.
ResetStatistics()	It resets all values if statistics gathering is enabled.

## SqlConnection Example

Now, let's create an example that establishes a connection to the SQL Server. We have created a **Student** database and will use it to connect. Look at the following C# code.

```
1. using (SqlConnection connection = new SqlConnection(connectionString))
2. {
3.     connection.Open();
4. }
```

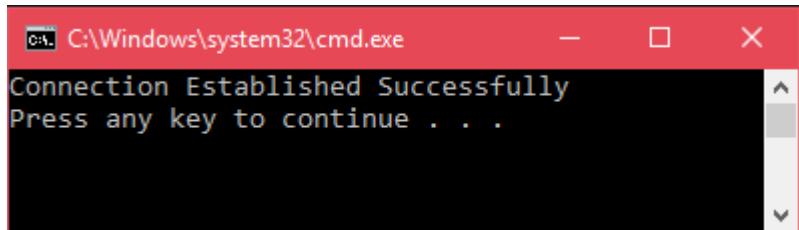
**Using** block is used to close the connection automatically. We don't need to call close () method explicitly, **using** block do this for ours implicitly when the code exits the block.

### // Program.cs

```
1. using System;
2. using System.Data.SqlClient;
3. namespace AdoNetConsoleApplication
4. {
5.     class Program
6.     {
7.         static void Main(string[] args)
8.         {
9.             new Program().Connecting();
10.        }
11.        public void Connecting()
12.        {
13.            using (
14.                // Creating Connection
15.                SqlConnection con = new SqlConnection("data source=.; database=student; integrated security=SS
PI")
16.            )
17.            {
18.                con.Open();
19.                Console.WriteLine("Connection Established Successfully");
}
```

```
20.      }
21.    }
22.  }
23. }
```

Output:



A screenshot of a Windows command prompt window titled 'C:\Windows\system32\cmd.exe'. The window contains the text 'Connection Established Successfully' and 'Press any key to continue . . .'. The window has standard red title bar and black background.

What, if we don't use **using** block.

If we don't use using block to create connection, we have to close connection explicitly. In the following example, we are using try-block instead of using block.

### // Program.cs

```
1. using System;
2. using System.Data.SqlClient;
3. namespace AdoNetConsoleApplication
4. {
5.   class Program
6.   {
7.     static void Main(string[] args)
8.     {
9.       new Program().Connecting();
10.    }
11.    public void Connecting()
12.    {
13.      SqlConnection con = null;
14.      try
15.      {
16.        // Creating Connection
17.        con = new SqlConnection("data source=.; database=student; integrated security=SSPI");
```

```

18.         con.Open();
19.         Console.WriteLine("Connection Established Successfully");
20.     }
21.     catch (Exception e)
22.     {
23.         Console.WriteLine("OOPS, something went wrong.\n"+e);
24.     }
25.     finally
26.     { // Closing the connection
27.         con.Close();
28.     }
29. }
30. }
31. }
```

Output:

The screenshot shows a standard Windows command prompt window. The title bar says 'cmd.exe'. The main area of the window displays the text 'Connection Established Successfully' followed by 'Press any key to continue . . .'. The window has a red header bar and a black body.

## ADO.NET SqlCommand Class

The SqlCommand class in ADO.NET is used to execute SQL queries or stored procedures against a SQL Server database. It allows your C# (or other .NET language) application to interact with the database by sending commands and retrieving results.

### Example: Using SqlCommand to Execute a Query

Let's walk through a simple example to illustrate how to use SqlCommand:

#### 1. Setting up the SqlConnection:

First, you need to set up the SqlConnection object to connect to your SQL Server database. Here's a basic example:

```

using System;
using System.Data.SqlClient;

class Program
{
    static void Main(string[] args)
    {
        string connectionString = "Server=myServerAddress;Database=myDataBase;User
Id=myUsername;Password=myPassword;";

        SqlConnection connection = new SqlConnection(connectionString);

        try
        {
            connection.Open();
            Console.WriteLine("Connection to database successful!");

            // Your SqlCommand operations go here

        }
        catch (Exception ex)
        {
            Console.WriteLine("Error: " + ex.Message);
        }
        finally
        {
            connection.Close();
        }
    }
}

```

Replace myServerAddress, myDataBase, myUsername, and myPassword with your actual database server details.

## 2. Executing a SELECT query using SqlCommand:

Now, let's use SqlCommand to execute a simple SELECT query and retrieve data from a Users table:

```

try
{
    string query = "SELECT * FROM Users";
    SqlCommand command = new SqlCommand(query, connection);

```

```

SqlDataReader reader = command.ExecuteReader();

while (reader.Read())
{
    Console.WriteLine($"User ID: {reader["UserID"]}, Name: {reader["Name"]}, Email: {reader["Email"]}");
}

reader.Close();
}
catch (Exception ex)
{
    Console.WriteLine("Error executing query: " + ex.Message);
}

```

- **SqlCommand** is created with the SQL query (SELECT \* FROM Users) and the SqlConnection.
- **ExecuteReader()** executes the query and returns a SqlDataReader object.
- **SqlDataReader** allows you to iterate over the results row by row (while (reader.Read())), accessing each column by its name (reader["ColumnName"]).

### 3. Executing other types of queries:

SqlCommand can be used for other types of SQL commands as well, such as INSERT, UPDATE, DELETE, or executing stored procedures. Here's an example of executing an INSERT query:

```

try
{
    string insertQuery = "INSERT INTO Users (Name, Email) VALUES ('John Doe', 'john.doe@example.com')";
    SqlCommand insertCommand = new SqlCommand(insertQuery, connection);

    int rowsAffected = insertCommand.ExecuteNonQuery();
    Console.WriteLine($"Rows affected: {rowsAffected}");
}

catch (Exception ex)
{
    Console.WriteLine("Error executing insert query: " + ex.Message);
}

```

- **ExecuteNonQuery()** is used for commands that don't return data (like INSERT, UPDATE, DELETE). It returns the number of rows affected.

## ADO.NET SqlDataReader Class

The SqlDataReader class in ADO.NET allows your C# (or other .NET language) application to efficiently retrieve data from a SQL Server database after executing a SQL query using SqlCommand. It provides a forward-only stream of rows from the database and is especially useful when dealing with large result sets.

### How SqlDataReader Works

#### 1. Executing a Query with SqlCommand:

First, you use SqlCommand to execute a SQL query against your database. Here's a quick recap from the previous example:

```
string connectionString = "Server=myServerAddress;Database=myDataBase;User
Id=myUsername;Password=myPassword;";
SqlConnection connection = new SqlConnection(connectionString);

try
{
    connection.Open();
    string query = "SELECT * FROM Users";
    SqlCommand command = new SqlCommand(query, connection);

    SqlDataReader reader = command.ExecuteReader();

    // Use SqlDataReader to retrieve data
    while (reader.Read())
    {
        // Access columns by their names or indexes
        Console.WriteLine($"User ID: {reader["UserID"]}, Name: {reader["Name"]}, Email:
{reader["Email"]}");
    }

    reader.Close();
}
catch (Exception ex)
{
    Console.WriteLine("Error: " + ex.Message);
```

```

} finally
{
    connection.Close();
}

```

## 2. Iterating Through Results:

- **ExecuteReader()**: This method of SqlCommand returns a SqlDataReader object.
- **Reading Data**: Use the Read() method of SqlDataReader to advance to the next row in the result set. It returns true if there are more rows to read; otherwise, false when all rows have been read.

## 3. Accessing Data:

- **Accessing Columns**: Use SqlDataReader to access column values either by their column name (reader["ColumnName"]) or by their ordinal position (reader[index]).
- **Data Types**: SqlDataReader automatically converts database column types to appropriate .NET data types (int, string, DateTime, etc.).

## 4. Closing the Reader:

- **Closing**: Always call Close() on the SqlDataReader object once you've finished reading data to release resources and close the underlying connection.

## Example Scenario

Imagine you have a Users table in your database with columns UserID, Name, and Email. Using SqlDataReader, you can fetch all users and display their details in the console or perform further processing in your application.

# ADO.NET DataSet

## What is a DataSet?

### 1. Storage for Data:

- A DataSet is like a container that holds data retrieved from a database. It keeps this data in memory, making it easy for your program to work with.

### 2. Disconnected from Database:

- Unlike directly reading data with something like SqlDataReader, which needs a live connection to the database, DataSet fetches data once and then disconnects. This means you can work with the data offline without being connected to the database all the time.

### 3. Structure:

- Inside a DataSet, you can have multiple tables (DataTable objects), each resembling a table from your database. Each table has rows and columns, just like in a database.

#### 4. Operations:

- **Fetching Data:** Use SqlDataAdapter to fill a DataSet with data from a database table using SQL queries or stored procedures.
- **Manipulating Data:** Once data is in a DataSet, you can modify, add, or delete rows and columns as needed.
- **Updating Database:** When you're ready, changes made in the DataSet can be sent back to the database using SqlDataAdapter.

#### Example Usage:

Let's illustrate with a simple scenario:

```
// Imagine you have a database with a table 'Users' having columns 'UserID', 'Name', 'Email'

// Setting up connection and DataSet
string connectionString = "Server=myServerAddress;Database=myDataBase;User
Id=myUsername;Password=myPassword;";
SqlConnection connection = new SqlConnection(connectionString);
DataSet dataSet = new DataSet();

try
{
    // Open connection
    connection.Open();

    // SQL query to fetch data
    string query = "SELECT * FROM Users";

    // Create SqlDataAdapter to fill DataSet with data
    SqlDataAdapter adapter = new SqlDataAdapter(query, connection);
    adapter.Fill(dataSet, "Users"); // 'Users' is the name given to the DataTable in the DataSet

    // Work with data in the DataSet
    DataTable usersTable = dataSet.Tables["Users"];
    foreach (DataRow row in usersTable.Rows)
    {
        Console.WriteLine($"User ID: {row["UserID"]}, Name: {row["Name"]}, Email:
{row["Email"]}");
    }

    // Example of adding a new row
    DataRow newRow = usersTable.NewRow();
    newRow["Name"] = "New User";
}
```

```

newRow["Email"] = "newuser@example.com";
usersTable.Rows.Add(newRow);

    // Update changes back to the database
    SqlCommandBuilder commandBuilder = new SqlCommandBuilder(adapter);
    adapter.Update(dataSet, "Users");

    Console.WriteLine("Changes updated to the database.");
}
catch (Exception ex)
{
    Console.WriteLine("Error: " + ex.Message);
} finally
{
    // Close connection
    connection.Close();
}

```

## ADO.NET DataAdapter

### What is a DataAdapter?

#### 1. Data Bridge:

- Think of DataAdapter as a translator or messenger between your program and the database.
- It manages communication for retrieving data (filling DataSet or DataTable) and sending updates (updating the database).

#### 2. Key Functions:

- **Fetching Data:** It uses SQL commands or stored procedures to fetch data from the database and stores it in a DataSet or DataTable.
- **Updating Data:** It helps in applying changes made in the DataSet or DataTable back to the database. This includes inserting, updating, or deleting rows.

#### 3. Components:

- **SelectCommand:** Specifies the SQL query or stored procedure to fetch data from the database.
- **InsertCommand, UpdateCommand, DeleteCommand:** Specify the SQL commands or stored procedures to apply changes (insert, update, delete) from DataSet back to the database.

### Example Usage:

Here's how you typically use DataAdapter to fetch data into a DataSet and update it back to the database:

```
// Setting up connection and DataAdapter
string connectionString = "Server=myServerAddress;Database=myDataBase;User
Id=myUsername;Password=myPassword;";
SqlConnection connection = new SqlConnection(connectionString);
DataSet dataSet = new DataSet();

try
{
    // Open connection
    connection.Open();

    // Define SQL query to fetch data
    string query = "SELECT * FROM Users";

    // Create DataAdapter with SelectCommand
    SqlDataAdapter adapter = new SqlDataAdapter(query, connection);

    // Fill DataSet with data from database
    adapter.Fill(dataSet, "Users"); // 'Users' is the name given to the DataTable in the DataSet

    // Work with data in the DataSet
    DataTable usersTable = dataSet.Tables["Users"];
    foreach (DataRow row in usersTable.Rows)
    {
        Console.WriteLine($"User ID: {row["UserID"]}, Name: {row["Name"]}, Email:
{row["Email"]}");
    }

    // Example of adding a new row
    DataRow newRow = usersTable.NewRow();
    newRow["Name"] = "New User";
    newRow["Email"] = "newuser@example.com";
    usersTable.Rows.Add(newRow);

    // Update changes back to the database
    SqlCommandBuilder commandBuilder = new SqlCommandBuilder(adapter);
    adapter.Update(dataSet, "Users");

    Console.WriteLine("Changes updated to the database.");
}

catch (Exception ex)
{
```

```

        Console.WriteLine("Error: " + ex.Message);
    } finally
    {
        // Close connection
        connection.Close();
    }
}

```

## **ADO.NET Architecture: CONNECTED VS DISCONNECTED**

### **Connected Architecture:**

**Explanation:** In connected architecture, your application stays directly connected to the database while it's interacting with data. This means:

- **Continuous Connection:** Your program opens a connection (`SqlConnection`) to the database and keeps it open as long as it needs to fetch or update data.
- **Real-Time Interaction:** When you use `SqlConnection` and `SqlCommand`, you're directly communicating with the database. For example, using `SqlDataReader` allows you to read data row by row from the database.
- **Immediate Updates:** Any changes you make, like inserting new data or updating existing records, are immediately reflected in the database. This architecture is ideal for applications that need real-time access to data and immediate synchronization with the database.

**Example:** Imagine a dashboard application that needs to display live stock prices from a database. It uses a connected approach to continuously fetch and display the latest data in realtime.

### **Disconnected Architecture:**

**Explanation:** In disconnected architecture, your application fetches data from the database, but then works with it locally without keeping a continuous connection open:

- **Data Stored Locally:** Instead of keeping a connection open, data is retrieved into a `DataSet` or `DataTable` using `DataAdapter`.
- **Offline Manipulation:** Once data is in memory (`DataSet`), your application can work with it offline. This means you can add, edit, or delete rows locally without affecting the actual database immediately.
- **Delayed Synchronization:** Changes made to the `DataSet` are synchronized back to the database only when you explicitly call `DataAdapter.Update()`. This architecture is useful when you need to work with data offline or in batches, optimizing performance and reducing load on the database server.

**Example:** Consider a mobile app for a delivery service. It downloads a list of orders into a DataSet when online, allowing delivery drivers to view and update order statuses offline. When they reconnect, the app syncs the local changes back to the central database.

### Choosing Between Architectures:

- **Connected:** Use when real-time interaction with data and immediate updates to the database are necessary, like in financial trading systems or live monitoring applications.
- **Disconnected:** Choose when working offline, batch processing, or reducing database load are priorities, such as in mobile apps, reporting systems, or applications with intermittent network connectivity.

## SQL Injection

SQL injection is a code injection technique that might destroy your database.

SQL injection is one of the most common web hacking techniques.

SQL injection is the placement of malicious code in SQL statements, via web page input.

## SQL in Web Pages

SQL injection usually occurs when you ask a user for input, like their username/userid, and instead of a name/id, the user gives you an SQL statement that you will **unknowingly** run on your database.

Look at the following example which creates a **SELECT** statement by adding a variable (txtUserId) to a select string. The variable is fetched from user input (getRequestString):

### Example [Get your own SQL Server](#)

```
txtUserId = getRequestString("UserId"); txtSQL = "SELECT *  
FROM Users WHERE UserId = " + txtUserId;
```

## SQL Injection Based on 1=1 is Always True

Look at the example above again. The original purpose of the code was to create an SQL statement to select a user, with a given user id.

If there is nothing to prevent a user from entering "wrong" input, the user can enter some "smart" input like this:

UserId:

Then, the SQL statement will look like this:

```
SELECT * FROM Users WHERE UserId = 105 OR 1=1;
```

The SQL above is valid and will return ALL rows from the "Users" table, since **OR 1=1** is always TRUE.

The SQL statement above is much the same as this:

```
SELECT UserId, Name, Password FROM Users WHERE UserId = 105 or 1=1;
```

A hacker might get access to all the user names and passwords in a database, by simply inserting 105 OR 1=1 into the input field.

### **SQL Injection Based on ""="" is Always True Here**

is an example of a user login on a web site:

Username:

Password:

### **Example**

```
uName = getRequestString("username");
uPass = getRequestString("userpassword");
```

```
sql = 'SELECT * FROM Users WHERE Name =' + uName + ' AND Pass =' + uPass + '''
```

### **Result**

```
SELECT * FROM Users WHERE Name ="John Doe" AND Pass ="myPass"
```

A hacker might get access to user names and passwords in a database by simply inserting " OR ""=" into the user name or password text box:

User Name:

Password:

The code at the server will create a valid SQL statement like this:

Result

```
SELECT * FROM Users WHERE Name ="" or ""="" AND Pass ="" or ""=""
```

The SQL above is valid and will return all rows from the "Users" table, since **OR** ""="" is always TRUE.

## SQL Injection Based on Batched SQL Statements

Most databases support batched SQL statement.

A batch of SQL statements is a group of two or more SQL statements, separated by semicolons.

The SQL statement below will return all rows from the "Users" table, then delete the "Suppliers" table.

Example

```
SELECT * FROM Users; DROP TABLE Suppliers Look
```

at the following example:

Example

txtUserId = getRequestString("UserId"); txtSQL = "SELECT \* FROM Users WHERE UserId = " +  
txtUserId; And the following input:

User id:

The valid SQL statement would look like this:

Result

```
SELECT * FROM Users WHERE UserId = 105; DROP TABLE Suppliers;
```

## Use SQL Parameters for Protection

To protect a web site from SQL injection, you can use SQL parameters.

SQL parameters are values that are added to an SQL query at execution time, in a controlled manner.

### ASP.NET Razor Example

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = @0"; db.Execute(txtSQL,txtUserId);
```

Note that parameters are represented in the SQL statement by a @ marker.

The SQL engine checks each parameter to ensure that it is correct for its column and are treated literally, and not as part of the SQL to be executed.

### Another Example

```
txtNam =
getRequestString("CustomerName"); txtAdd =
getRequestString("Address"); txtCit =
getRequestString("City");
txtSQL = "INSERT INTO Customers (CustomerName,Address,City) Values(@0,@1,@2)";
db.Execute(txtSQL,txtNam,txtAdd,txtCit);
```

## Examples

The following examples shows how to build parameterized queries in some common web languages.

SELECT STATEMENT IN ASP.NET:

```
txtUserId = getRequestString("UserId");
sql = "SELECT * FROM Customers WHERE CustomerId = @0"; command
= new SqlCommand(sql);
command.Parameters.AddWithValue("@0",txtUserId);
command.ExecuteReader();
```

INSERT INTO STATEMENT IN ASP.NET:

```

txtNam =
getRequestParam("CustomerName"); txtAdd =
getRequestParam("Address"); txtCit =
getRequestParam("City");
txtSQL = "INSERT INTO Customers (CustomerName,Address,City)
Values(@0,@1,@2)"; command = new SqlCommand(txtSQL);
command.Parameters.AddWithValue("@0",txtNam);
command.Parameters.AddWithValue("@1",txtAdd);
command.Parameters.AddWithValue("@2",txtCit); command.ExecuteNonQuery();

```

INSERT INTO STATEMENT IN PHP:

```

$stmt = $dbh->prepare("INSERT INTO Customers (CustomerName,Address,City)
VALUES (:nam, :add, :cit)");
$stmt->bindParam(':nam', $txtNam);
$stmt->bindParam(':add', $txtAdd);
$stmt->bindParam(':cit', $txtCit); $stmt->execute();

```

## **Parametrized queries in SQL:**

### **What are Parameterized Queries?**

#### **1. Definition:**

- Parameterized queries are SQL queries that use placeholders (parameters) for input values, rather than directly inserting user-supplied data into the query string.

#### **2. Why Use Them?:**

- They help prevent SQL injection attacks by separating SQL code from data inputs.
- Instead of concatenating strings (which can be manipulated by attackers), parameters ensure that input values are treated as data, not executable code.

#### **3. How They Work:**

- In a parameterized query, placeholders (often denoted by question marks ? or named parameters like @ParameterName) are used in the SQL statement where values will be inserted.
- The actual values are provided separately when executing the query.

#### **4. Example:**

- Suppose you have a login query that checks if a username and password match in a database:

```

SELECT * FROM Users WHERE username = @Username AND password =
@Password

```

- @Username and @Password are parameters here.

- When executing the query, you provide actual values for @Username and @Password instead of embedding them directly into the SQL string.

## 5. Benefits:

- **Security:** Protects against SQL injection attacks by ensuring input values are treated as data, not executable SQL code.
- **Performance:** Improves query performance because the database can optimize execution plans for parameterized queries.
- **Maintainability:** Easier to read and maintain code since SQL logic and data values are clearly separated.

### Implementation in SQL Server (Example):

Here's how you might implement a parameterized query in SQL Server using SqlCommand in C#:

```
string connectionString = "Server=myServerAddress;Database=myDataBase;User
Id=myUsername;Password=myPassword;";
string query = "SELECT * FROM Users WHERE username = @Username AND password =
@Password";

using (SqlConnection connection = new SqlConnection(connectionString))
{
    SqlCommand command = new SqlCommand(query, connection);
    command.Parameters.AddWithValue("@Username", usernameTextBox.Text);
    command.Parameters.AddWithValue("@Password", passwordTextBox.Text);

    connection.Open();

    // Execute the query and process the results
    SqlDataReader reader = command.ExecuteReader();
    // Process the results as needed

    connection.Close();
}
```

- **SqlCommand** is used with parameters @Username and @Password.
- **Parameters.AddWithValue** sets values for @Username and @Password from user input (e.g., text boxes).

By using parameterized queries, developers can create more secure and efficient applications that protect against SQL injection vulnerabilities while maintaining clear and manageable SQL code.

## Sql Command Builder Class in disconnect method:

### Definition

It's a tool provided by .NET for automating the generation of SQL commands that reflect changes made to a DataSet or DataTable, ensuring data integrity between your application and the database.

### Step-by-Step Procedure

1. **Fetch Data:** Start by retrieving data from the database into a DataTable using a SqlDataAdapter. This data can be manipulated locally without being continuously connected to the database.

```
SqlDataAdapter dataAdapter = new SqlDataAdapter("SELECT * FROM YourTable",
connectionString);
DataTable dataTable = new DataTable();
dataAdapter.Fill(dataTable);
```

2. **Modify Data Locally:** Make changes to the data in the DataTable within your application. These changes could involve updating existing records, adding new ones, or deleting existing records.

```
// Example: Changing a value
foreach (DataRow row in dataTable.Rows)
{
    row["ColumnName"] = "NewValue";
}
```

3. **Create SqlCommandBuilder:** Instantiate a SqlCommandBuilder using the SqlDataAdapter. This step prepares the SqlDataAdapter to automatically generate SQL commands based on changes detected in the DataTable.

```
SqlCommandBuilder commandBuilder = new SqlCommandBuilder(dataAdapter);
```

4. **Update Database:** Use the Update method of SqlDataAdapter to apply the changes from the DataTable back to the database. The SqlCommandBuilder helps by generating the necessary SQL commands (UPDATE, INSERT, DELETE) based on the changes made in the DataTable.

```
dataAdapter.Update(dataTable);
```

5. **Dispose Resources:** Properly dispose of resources such as DataTable and SqlDataAdapter to release database connections and free up memory.

```
dataTable.Dispose(); dataAdapter.Dispose();
```

## Example

Suppose you have a Windows Forms application that displays employee information from a database. You want to allow users to modify this information locally within the application and then synchronize these changes back to the database using SqlCommandBuilder.

### Step-by-Step Example

#### 1. Setup

Assume you have a Windows Forms application with a DataGridView (dataGridViewEmployees) bound to a DataTable (dataTableEmployees).

```
// Assume these are class-level variables
private DataTable dataTableEmployees; private SqlDataAdapter
dataAdapter; private SqlCommandBuilder
commandBuilder; private string connectionString =
"YourConnectionString";
```

#### 2. Load Data

When the form loads, fetch data from the database into a DataTable and bind it to the DataGridView.

```
private void Form1_Load(object sender, EventArgs e)
{
    // Create a data adapter and fill the DataTable
    dataAdapter = new SqlDataAdapter("SELECT * FROM Employees", connectionString);
    dataTableEmployees = new DataTable();
    dataAdapter.Fill(dataTableEmployees);

    // Bind DataTable to DataGridView
    dataGridViewEmployees.DataSource = dataTableEmployees;
}
```

#### 3. Modify Data Locally

Allow users to modify data in the DataGridView. For simplicity, assume users can edit cells directly in the DataGridView.

```
// No specific code needed here as users can directly edit DataGridView
```

#### 4. Update Database

When the user wants to save changes back to the database (e.g., on a button click event), use SqlCommandBuilder to automatically generate and execute SQL commands.

```
private void btnSaveChanges_Click(object sender, EventArgs e)
{
    try
    {
        // Automatically generate INSERT, UPDATE, DELETE commands
        commandBuilder = new SqlCommandBuilder(dataAdapter);

        // Update database with changes from DataTable
        dataAdapter.Update(dataTableEmployees);

        MessageBox.Show("Changes saved successfully.");
    }
    catch (Exception ex)
    {
        MessageBox.Show("Error saving changes: " + ex.Message);
    }
}
```

#### 5. Dispose Resources

Properly dispose of resources to release database connections and clean up.

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
    // Dispose resources
    dataTableEmployees.Dispose();
    dataAdapter.Dispose();
}
```

#### Summary

In this example:

- We initially fetch data into a DataTable using a SqlDataAdapter.
- Users can modify data in a DataGridView.

- When the "Save Changes" button is clicked, SqlCommandBuilder helps generate SQL commands (UPDATE, INSERT, DELETE) based on changes in the DataTable, which are then used by SqlDataAdapter to update the database.
- Finally, resources are properly disposed to maintain efficient memory usage and database connection management.

## **Delegates in C#:**

### **Definition:**

- A delegate in C# is a type that represents references to methods with a specific signature (return type and parameters).
- It essentially acts as a method pointer or a contract for methods that can be invoked through it.

### **Delegate Declaration:**

- You define a delegate using the delegate keyword, specifying the method signature it can reference. □ Example:

```
public delegate void MyDelegate(int x, int y);
```

- This delegate can reference methods that take two int parameters and return void.

### **Usage Scenarios:**

- **Event Handling:** Delegates are commonly used to implement event handling. They allow methods to subscribe (register) and unsubscribe (deregister) from events dynamically.
- **Callback Mechanisms:** Used in scenarios where you want a method to call back another method with results or status updates.
- **Functional Programming:** Facilitates functional programming techniques by treating methods as first-class citizens.

### **Example Scenario:**

- Suppose you have a scenario where you want to calculate and display the sum of two numbers using different methods based on user input.

```
// Define a delegate
public delegate int CalculationDelegate(int a, int b);

// Methods to be assigned to the delegate
```

```

public class Calculator
{
    public int Add(int a, int b)
    {
        return a + b;
    }

    public int Multiply(int a, int b)
    {
        return a * b;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Calculator calculator = new Calculator();

        // Assign methods to the delegate
        CalculationDelegate operationDelegate = calculator.Add;
        int result = PerformCalculation(5, 3, operationDelegate);
        Console.WriteLine($"Result of addition: {result}");

        operationDelegate = calculator.Multiply;      result =
        PerformCalculation(5, 3, operationDelegate);
        Console.WriteLine($"Result of multiplication: {result}");
    }
}

// Method that takes a delegate as parameter and invokes it
static int PerformCalculation(int x, int y, CalculationDelegate calculation)
{
    return calculation(x, y);
}

```

## Multicast Delegates

### 1. Definition:

- A multicast delegate in C# is a type of delegate that can hold references to multiple methods.
- It allows you to call multiple methods through a single delegate invocation.

## 2. Usage:

- Multicast delegates are particularly useful in scenarios where you want to notify or trigger multiple methods in response to an event or action.
- They streamline event handling by allowing multiple subscribers to register and receive notifications from a single event.

## 3. Syntax:

- To declare a multicast delegate, you use the same syntax as a regular delegate but can add multiple methods to it using the += operator.
- Example:

```
public delegate void MyDelegate();
```

```
// Instantiate the delegate
MyDelegate multicastDelegate = Method1;
multicastDelegate += Method2;
multicastDelegate += Method3;
```

```
// Invoke the delegate multicastDelegate();
```

## 4. Execution Order:

- When you invoke a multicast delegate (multicastDelegate() in the example), it sequentially invokes each method that it holds references to.
- The methods are called in the order they were added (Method1, Method2, Method3 in this case).

## 5. Example Scenario:

- Consider an application where you want to notify multiple subsystems when a certain event occurs, such as saving data or updating a user interface.
- Each subsystem registers its method with a multicast delegate, and invoking the delegate triggers all registered methods.

## Benefits of Multicast Delegates

- **Flexibility:** Enables decoupling of event sources from event handlers, allowing for modular and flexible architecture.
- **Simplicity:** Provides a straightforward way to manage multiple method calls through a single delegate instance.
- **Event-driven Design:** Essential for implementing robust event-driven programming patterns, like observer or publisher-subscriber models.

## Introduction to Windows .forms:

### What is Windows Forms?

**1. GUI Framework:**

- Windows Forms (WinForms) is a framework for developing Windows-based applications with a graphical user interface (GUI).
- It provides a collection of pre-built controls like buttons, textboxes, labels, and more, which you can drag and drop onto forms to design the user interface.

**2. Event-driven Programming:**

- WinForms uses event-driven programming model where actions like button clicks or form loading trigger events, and you write code (event handlers) to respond to these events.
- This model allows developers to create responsive applications that react to user input and system events.

**3. Integrated with .NET:**

- WinForms is integrated with the .NET framework, leveraging features like language interoperability (C#, VB.NET, etc.), memory management (via garbage collection), and a rich class library for common tasks.
- It simplifies development by providing a consistent programming model across different .NET languages.

**4. Design and Development:**

- Applications are typically designed using Visual Studio, a popular integrated development environment (IDE) for .NET development. ○ You design forms visually by dragging controls from the toolbox onto a design surface (form), then customize their properties and write code to handle events.

**Example Scenario:**

Imagine you want to create a simple calculator application using Windows Forms:

• **Design Phase:**

- Open Visual Studio and create a new Windows Forms Application project. ○ Design the calculator's user interface by dragging buttons (for digits and operations), textbox (for display), and labels onto the form.

• **Coding Phase:**

- Write code to handle button click events. For example, clicking the 1 button should append 1 to the textbox.
- Implement logic to perform calculations based on user input when clicking the = button.

• **Execution Phase:**

- Build and run the application to interact with the calculator GUI. ○ Test functionalities such as addition, subtraction, and clearing the display.

**Benefits of Windows Forms:**

- **Rapid Development:** Quickly build GUI applications using drag-and-drop controls and event-driven programming.
- **Familiarity:** Windows Forms applications have a native look and feel on Windows, making them intuitive for users.
- **Integration:** Seamlessly integrates with other .NET technologies and frameworks, such as ASP.NET for web applications or WPF for more advanced UI designs.

## **What is JavaScript:**

JavaScript is a programming language used to make websites interactive. It lets developers add features like animations, forms that respond to user input, and updates that happen without reloading the entire webpage. It runs in your web browser and helps create dynamic and engaging experiences for users.

## **Features of JavaScript**

There are following features of JavaScript:

1. All popular web browsers support JavaScript as they provide built-in execution environments.
2. JavaScript follows the syntax and structure of the C programming language. Thus, it is a structured programming language.
3. JavaScript is a weakly typed language, where certain types are implicitly cast (depending on the operation).
4. JavaScript is an object-oriented programming language that uses prototypes rather than using classes for inheritance.
5. It is a light-weighted and interpreted language.
6. It is a case-sensitive language.
7. JavaScript is supportable in several operating systems including, Windows, macOS, etc.
8. It provides good control to the users over the web browsers.

## **Application of JavaScript**

JavaScript is used to create interactive websites. It is mainly used for:

- Client-side validation, ○ Dynamic drop-down menus, ○ Displaying date and time,

- Displaying pop-up windows and dialog boxes (like an alert dialog box, confirm dialog box and prompt dialog box),
- Displaying clocks etc.

## JavaScript Example

1. `<script>`
2. `document.write("Hello JavaScript by JavaScript");`
3. `</script>`

## JavaScript Example

1. [JavaScript Example](#)
2. [Within body tag](#)
3. [Within head tag](#)

Javascript example is easy to code. JavaScript provides 3 places to put the JavaScript code: within body tag, within head tag and external JavaScript file.

Let's create the first JavaScript example.

1. `<script type="text/javascript">`
2. `document.write("JavaScript is a simple language for javatpoint learners");` 3.  
`</script>`

The **script** tag specifies that we are using JavaScript.

The **text/javascript** is the content type that provides information to the browser about the data.

The **document.write()** function is used to display dynamic content through JavaScript. We will learn about document object in detail later.

### 3 Places to put JavaScript code

1. Between the body tag of html
2. Between the head tag of html
3. In .js file (external javaScript)

---

## 1) JavaScript Example : code between the body tag

In the above example, we have displayed the dynamic content using JavaScript. Let's see the simple example of JavaScript that displays alert dialog box.

1. `<script type="text/javascript">`
  2. `alert("Hello Javatpoint");`
  3. `</script>`
- 

## 2) JavaScript Example : code between the head tag

Let's see the same example of displaying alert dialog box of JavaScript that is contained inside the head tag.

In this example, we are creating a function msg(). To create function in JavaScript, you need to write function with function\_name as given below.

To call function, you need to work on event. Here we are using onclick event to call msg() function.

1. `<html>`
2. `<head>`
3. `<script type="text/javascript">`
4. `function msg(){`
5. `alert("Hello Javatpoint");`
6. `}`
7. `</script>`
8. `</head>`
9. `<body>`
10. `<p>Welcome to JavaScript</p>`
11. `<form>`
12. `<input type="button" value="click" onclick="msg()" />`
13. `</form>`
14. `</body>`
15. `</html>`

## External JavaScript file

We can create external JavaScript file and embed it in many html page.

It provides **code re usability** because single JavaScript file can be used in several html pages.

An external JavaScript file must be saved by .js extension. It is recommended to embed all JavaScript files into a single file. It increases the speed of the webpage.

Let's create an external JavaScript file that prints Hello Javatpoint in a alert dialog box.

1. function msg(){
2. alert("Hello Javatpoint");
3. }

Let's include the JavaScript file into html page. It calls the JavaScript function on button click.

### **index.html**

1. <html>
2. <head>
3. <script type="text/javascript" src="message.js"></script>
4. </head>
5. <body>
6. <p>Welcome to JavaScript</p>
7. <form>
8. <input type="button" value="click" onclick="msg()" />
9. </form>
10. </body>
11. </html>

### **Advantages of External JavaScript**

There will be following benefits if a user creates an external javascript:

1. It helps in the reusability of code in more than one HTML file.
2. It allows easy code readability.

3. It is time-efficient as web browsers cache the external js files, which further reduces the page loading time.
4. It enables both web designers and coders to work with html and js files parallelly and separately, i.e., without facing any code conflicts.
5. The length of the code reduces as only we need to specify the location of the js file.

### **Disadvantages of External JavaScript**

There are the following disadvantages of external files:

1. The stealer may download the coder's code using the url of the js file.
2. If two js files are dependent on one another, then a failure in one file may affect the execution of the other dependent file.
3. The web browser needs to make an additional http request to get the js code.
4. A tiny to a large change in the js code may cause unexpected results in all its dependent files.
5. We need to check each file that depends on the commonly created external javascript file.
6. If it is a few lines of code, then better to implement the internal javascript code.

## **JavaScript Basics**

### **JavaScript Comment**

The **JavaScript comments** are meaningful way to deliver message. It is used to add information about the code, warnings or suggestions so that end user can easily interpret the code.

The JavaScript comment is ignored by the JavaScript engine i.e. embedded in the browser.

### ***Advantages of JavaScript comments***

There are mainly two advantages of JavaScript comments.

1. **To make code easy to understand** It can be used to elaborate the code so that end user can easily understand the code.

2. **To avoid the unnecessary code** It can also be used to avoid the code being executed. Sometimes, we add the code to perform some action. But after sometime, there may be need to disable the code. In such case, it is better to use comments.

### Types of JavaScript Comments

There are two types of comments in JavaScript.

1. Single-line Comment
2. Multi-line Comment

### JavaScript Single line Comment

It is represented by double forward slashes (//). It can be used before and after the statement.

Let's see the example of single-line comment i.e. added before the statement.

1. `<script>`
2. `// It is single line comment`
3. `document.write("hello javascript");`
4. `</script>`

Let's see the example of single-line comment i.e. added after the statement.

1. `<script>`
2. `var a=10;`
3. `var b=20;`
4. `var c=a+b;//It adds values of a and b variable`
5. `document.write(c);//prints sum of 10 and 20`
6. `</script>`

### JavaScript Multi line Comment

It can be used to add single as well as multi line comments. So, it is more convenient.

It is represented by forward slash with asterisk then asterisk with forward slash. For example:

1. /\* your code here \*/

It can be used before, after and middle of the statement.

1. <script>
2. /\* It is multi line comment.
3. It will not be displayed \*/
4. document.write("example of javascript multiline comment");
5. </script>

## JavaScript Variable

A **JavaScript variable** is simply a name of storage location. There are two types of variables in JavaScript : local variable and global variable.

There are some rules while declaring a JavaScript variable (also known as identifiers).

1. Name must start with a letter (a to z or A to Z), underscore( \_ ), or dollar( \$ ) sign.
2. After first letter we can use digits (0 to 9), for example value1.
3. JavaScript variables are case sensitive, for example x and X are different variables.

### Correct JavaScript variables

1. var x = 10;
2. var \_value="sonoo";

### Incorrect JavaScript variables

1. var 123=30;
2. var \*aa=320;

### Example of JavaScript variable

Let's see a simple example of JavaScript variable.

1. <script>
2. var x = 10;
3. var y = 20;

4. var z=x+y;
5. document.write(z);
6. </script>

**Output of the above example**

30

### JavaScript local variable

A JavaScript local variable is declared inside block or function. It is accessible within the function or block only. For example:

1. <script>
2. function abc(){
3. var x=10;//local variable
4. }
5. </script>

Or,

1. <script>
2. If(10<13){
3. var y=20;//JavaScript local variable
4. }
5. </script>

### JavaScript global variable

A **JavaScript global variable** is accessible from any function. A variable i.e. declared outside the function or declared with window object is known as global variable. For example:

1. <script>
2. var data=200;//global variable
3. function a(){
4. document.writeln(data);
5. }
6. function b(){

```

7. document.writeln(data);
8. }
9. a();//calling JavaScript function
10. b();
11. </script>

```

## JavaScript Global Variable

A **JavaScript global variable** is declared outside the function or declared with window object. It can be accessed from any function.

Let's see the simple example of global variable in JavaScript.

```

1. <script>
2. var value=50;//global variable
3. function a(){
4. alert(value);
5. }
6. function b(){
7. alert(value);
8. }
9. </script>

```

### ***Declaring JavaScript global variable within function***

To declare JavaScript global variables inside function, you need to use **window object**. For example:

1. **window.value**=90;

Now it can be declared inside any function and can be accessed from any function. For example:

1. function m(){
2. **window.value**=100;//declaring global variable by window object 3.
3. }
4. function n(){
5. alert(**window.value**);//accessing global variable from other function 6.
6. }

## Internals of global variable in JavaScript

When you declare a variable outside the function, it is added in the window object internally. You can access it through window object also. For example:

1. var value=50;
2. function a(){}
3. alert(window.value);//accessing global variable
4. }

## Javascript Data Types

JavaScript provides different **data types** to hold different types of values. There are two types of data types in JavaScript.

1. Primitive data type
2. Non-primitive (reference) data type

JavaScript is a **dynamic type language**, means you don't need to specify type of the variable because it is dynamically used by JavaScript engine. You need to use **var** here to specify the data type. It can hold any type of values such as numbers, strings etc. For example:

1. var a=40;//holding number
2. var b="Rahul";//holding string

### JavaScript primitive data types

Data Type	Description
Object	represents instance through which we can access members
Array	represents group of similar values
RegExp	represents regular expression

There are five types of primitive data types in JavaScript. They are as follows:

## JavaScript non-primitive data types

The non-primitive data types are as follows:

## JavaScript Operators

JavaScript operators are symbols that are used to perform operations on operands. For example:

```
var sum=10+20;
```

Data Type	Description
String	represents sequence of characters e.g. "hello"
Number	represents numeric values e.g. 100
Boolean	represents boolean value either false or true
Undefined	represents undefined value
Null	represents null i.e. no value at all

Here, + is the arithmetic operator and = is the assignment operator.

There are following types of operators in JavaScript.

1. Arithmetic Operators
2. Comparison (Relational) Operators
3. Bitwise Operators
4. Logical Operators
5. Assignment Operators
6. Special Operators

## JavaScript If-else

The **JavaScript if-else statement** is used to execute the code whether condition is true or false. There are three forms of if statement in JavaScript.

1. If Statement
2. If else statement
3. if else if statement

### JavaScript If statement

It evaluates the content only if expression is true. The signature of JavaScript if statement is given below.

1. `if(expression){`
2. `//content to be evaluated`
3. `}`

Let's see the simple example of if statement in javascript.

1. `<script>`
2. `var a=20;`
3. `if(a>10){`
4. `document.write("value of a is greater than 10");`
5. `}`
6. `</script>`

#### ***Output of the above example***

value of a is greater than 10

### JavaScript If...else Statement

It evaluates the content whether condition is true or false. The syntax of JavaScript if-else statement is given below.

1. `if(expression){`
2. `//content to be evaluated if condition is true`

```

3. }
4. else{
5. //content to be evaluated if condition is false
6. }

```

Let's see the example of if-else statement in JavaScript to find out the even or odd number.

```

1. <script>
2. var a=20;
3. if(a%2==0){
4. document.write("a is even number");
5. }
6. else{
7. document.write("a is odd number");
8. }
9. </script>

```

***Output of the above example***

a is even number

### JavaScript If...else if statement

It evaluates the content only if expression is true from several expressions. The signature of JavaScript if else if statement is given below.

```

1. if(expression1){
2. //content to be evaluated if expression1 is true
3. }
4. else if(expression2){
5. //content to be evaluated if expression2 is true
6. }
7. else if(expression3){
8. //content to be evaluated if expression3 is true
9. }
10. else{

```

```

11. //content to be evaluated if no expression is true
12. }

```

Let's see the simple example of if else if statement in javascript.

```

1. <script>
2. var a=20;
3. if(a==10){
4.   document.write("a is equal to 10");
5. }
6. else if(a==15){
7.   document.write("a is equal to 15");
8. }
9. else if(a==20){
10.  document.write("a is equal to 20");
11. }
12. else{
13.   document.write("a is not equal to 10, 15 or 20");
14. }
15. </script>

```

## JavaScript Switch

The **JavaScript switch statement** is used to execute one code from multiple expressions. It is just like else if statement that we have learned in previous page. But it is convenient than *if..else..if* because it can be used with numbers, characters etc.

The signature of JavaScript switch statement is given below.

```

1. switch(expression){
2.   case value1:
3.     code to be executed;
4.     break;
5.   case value2:
6.     code to be executed;
7.     break;

```

8. .....
- 9.
10. default:
11. code to be executed if above values are not matched;
12. }

Let's see the simple example of switch statement in javascript.

1. **<script>**
2. var grade='B';
3. var result;
4. switch(grade){
5. case 'A':
6. result="A Grade";
7. break;
8. case 'B':
9. result="B Grade";
10. break;
11. case 'C':
12. result="C Grade";
13. break;
14. default:
15. result="No Grade";
16. }
17. document.write(result);
18. **</script>**

#### ***Output of the above example***

B Grade

## **JavaScript Loops**

The **JavaScript loops** are used to *iterate the piece of code* using for, while, do while or for-in loops. It makes the code compact. It is mostly used in array.

There are four types of loops in JavaScript.

1. for loop
2. while loop
3. do-while loop
4. for-in loop

---

## 1) JavaScript For loop

The **JavaScript for loop** *iterates the elements for the fixed number of times*. It should be used if number of iteration is known. The syntax of for loop is given below.

1. for (initialization; condition; increment)
2. {
3. code to be executed
4. }

Let's see the simple example of for loop in javascript.

1. **<script>**
2. for (**i=1**; **i<=5**; **i++**)
3. {
4. document.write(**i + "<br/>"**)
5. }
6. **</script>**

Output:

```
1  
2
```

```
3  
4  
5
```

## 2) JavaScript while loop

The **JavaScript while loop** *iterates the elements for the infinite number of times*. It should be used if number of iteration is not known. The syntax of while loop is given below.

1.    while (condition)
2.    {
3.    code to be executed
4.    }

Let's see the simple example of while loop in javascript.

1. **<script>**
2. var i=11;
3. while (i<=15)
4. {
5. document.write(i + "<br/>");
6. i++;
7. }
8. **</script>**

Output:

```
11
12
13
14
15
```

### **3) JavaScript do while loop**

The **JavaScript do while loop** iterates the elements for the infinite number of times like while loop. But, code is executed at least once whether condition is true or false. The syntax of do while loop is given below.

1.    do{
2.    code to be executed
3.    }while (condition);

Let's see the simple example of do while loop in javascript.

1. **<script>**
2. var i=21;
3. do{

4. document.write(i + "<br/>");
5. i++;
6. }while (i<=25);
7. </script>

Output:

```
21  
22  
23  
24  
25
```

## JavaScript Functions

**JavaScript functions** are used to perform operations. We can call JavaScript function many times to reuse the code.

### *Advantage of JavaScript function*

There are mainly two advantages of JavaScript functions.

1. **Code reusability:** We can call a function several times so it save coding.
2. **Less coding:** It makes our program compact. We don't need to write many lines of code each time to perform a common task.

---

### **JavaScript Function Syntax**

The syntax of declaring function is given below.

1. function functionName([arg1, arg2, ...argN]){}  
2. //code to be executed  
3. }

JavaScript Functions can have 0 or more arguments.

### **JavaScript Function Example**

Let's see the simple example of function in JavaScript that does not has arguments.

```

1. <script>
2. function msg(){
3.   alert("hello! this is message");
4. }
5. </script>
6. <input type="button" onclick="msg()" value="call function"/>

```

### JavaScript Function Arguments

We can call function by passing arguments. Let's see the example of function that has one argument.

```

1. <script>
2. function getcube(number){
3.   alert(number*number*number);
4. }
5. </script>
6. <form>
7. <input type="button" value="click" onclick="getcube(4)" />
8. </form>

```

### Function with Return Value

We can call function that returns a value and use it in our program. Let's see the example of function that returns value.

```

1. <script>
2. function getInfo(){
3.   return "hello javatpoint! How r u?";
4. }
5. </script>
6. <script>
7. document.write(getInfo());
8. </script>

```

### Output of the above example

hello javatpoint! How r u?

## JavaScript Function Object

In JavaScript, the purpose of **Function constructor** is to create a new Function object. It executes the code globally. However, if we call the constructor directly, a function is created dynamically but in an unsecured way.

### Syntax

1. `new Function ([arg1[, arg2[, ....argn]],] functionBody)` **Parameter**

**arg1, arg2, .... , argn** - It represents the argument used by function.

**functionBody** - It represents the function definition.

## JavaScript Function Methods

Let's see function methods with description.

### JavaScript Function Object Examples

#### Example 1

Let's see an example to display the sum of given numbers.

1. `<script>`
2. `var add=new Function("num1","num2","return num1+num2");`

Method	Description
<code>apply()</code>	It is used to call a function contains this value and a single array of arguments.
<code>bind()</code>	It is used to create a new function.
<code>call()</code>	It is used to call a function contains this value and an argument list.
<code>toString()</code>	It returns the result in a form of a string.

3. `document.writeln(add(2,5));`

4. </script>

**Output:**

7

## **JavaScript Objects**

### **JavaScript Objects**

A JavaScript object is an entity having state and behavior (properties and method). For example: car, pen, bike, chair, glass, keyboard, monitor etc.

JavaScript is an object-based language. Everything is an object in JavaScript.

JavaScript is template based not class based. Here, we don't create class to get the object. But, we directly create objects.

### **Creating Objects in JavaScript**

There are 3 ways to create objects.

1. By object literal
2. By creating instance of Object directly (using new keyword)
3. By using an object constructor (using new keyword)

#### **1) JavaScript Object by object literal**

The syntax of creating object using object literal is given below:

1. **object={property1:value1,property2:value2.....propertyN:valueN}**

As you can see, property and value is separated by : (colon).

Let's see the simple example of creating object in JavaScript.

1. <script>
2. **emp={id:102,name:"Shyam Kumar",salary:40000}**
3. **document.write(emp.id+" "+emp.name+" "+emp.salary);**
4. </script>

***Output of the above example***

102 Shyam Kumar 40000

---

## 2) By creating instance of Object

The syntax of creating object directly is given below:

1. var **objectname**=**new Object()**;

Here, **new keyword** is used to create object.

Let's see the example of creating object directly.

1. **<script>**
2. var **emp**=**new Object()**;
3. **emp.id**=**101**;
4. **emp.name**=**"Ravi Malik"**;
5. **emp.salary**=**50000**;
6. document.write(**emp.id**+" "+**emp.name**+" "+**emp.salary**);
7. **</script>**

**Output of the above example**

101 Ravi 50000

---

## 3) By using an Object constructor

Here, you need to create function with arguments. Each argument value can be assigned in the current object by using this keyword.

The **this keyword** refers to the current object.

The example of creating object by object constructor is given below.

1. **<script>**
2. function emp(id,name,salary){
3. **this.id**=id;
4. **this.name**=name;
5. **this.salary**=salary;

```

6. }
7. e=new emp(103,"Vimal Jaiswal",30000);
8.
9. document.write(e.id+" "+e.name+" "+e.salary);
10. </script>

```

***Output of the above example***

103 Vimal Jaiswal 30000

### Defining method in JavaScript Object

We can define method in JavaScript object. But before defining method, we need to add property in the function with same name as method.

The example of defining method in object is given below.

```

1. <script>
2. function emp(id,name,salary){
3.   this.id=id;
4.   this.name=name; 5. this.salary=salary;
5.
6.
7. this.changeSalary=changeSalary;
8. function changeSalary(otherSalary){
9.   this.salary=otherSalary;
10. }
11. }
12. e=new emp(103,"Sonoo Jaiswal",30000);
13. document.write(e.id+" "+e.name+" "+e.salary);
14. e.changeSalary(45000);
15. document.write("<br>"+e.id+" "+e.name+" "+e.salary);
16. </script>

```

***Output of  
the above  
example***

103	Sonoo	Jaiswal	30000
103 Sonoo Jaiswal 45000			

### JavaScript Object Methods

The various methods of Object are as follows:

S.No	Methods	Description
1	<a href="#"><u>Object.assign()</u></a>	This method is used to copy enumerable and own properties from a Source object to a target object
2	<a href="#"><u>Object.create()</u></a>	This method is used to create a new object with the specified prototype object and properties.
3	<a href="#"><u>Object.defineProperty()</u></a>	This method is used to describe some behavioral attributes of the property.
4	<a href="#"><u>Object.defineProperties()</u></a>	This method is used to create or configure multiple object properties.
5	<a href="#"><u>Object.entries()</u></a>	This method returns an array with arrays of the key, value pairs.
6	<a href="#"><u>Object.freeze()</u></a>	This method prevents existing properties from being removed.

7	<u><a href="#">Object.getOwnPropertyDescriptor()</a></u>	This method returns a property descriptor for the specified property of the specified object.
8	<u><a href="#">Object.getOwnPropertyDescriptors()</a></u>	This method returns all own property descriptors of a given object.

## JavaScript Array

**JavaScript array** is an object that represents a collection of similar type of elements.

There are 3 ways to construct array in JavaScript

1. By array literal
2. By creating instance of Array directly (using new keyword)
3. By using an Array constructor (using new keyword)

### 1) JavaScript array literal

The syntax of creating array using array literal is given below:

1. var **arrayname**=[value1,value2.....valueN];

As you can see, values are contained inside [ ] and separated by , (comma).

Let's see the simple example of creating and using array in JavaScript.

```
1. <script>
2. var emp=["Sonoo","Vimal","Ratan"];
3. for (i=0;i<emp.length;i++){
4.   document.write(emp[i] + "<br/>");
5. }
6. </script>
```

The .length property returns the length of an array.

### Output of the above example

9	<a href="#"><u>Object.getOwnPropertyNames()</u></a>	This method returns an array of all properties (enumerable or not) found.
10	<a href="#"><u>Object.getOwnPropertySymbols()</u></a>	This method returns an array of all own symbol key properties.
11	<a href="#"><u>Object.getPrototypeOf()</u></a>	This method returns the prototype of the specified object.
12	<a href="#"><u>Object.is()</u></a>	This method determines whether two values are the same value.
13	<a href="#"><u>Object.isExtensible()</u></a>	This method determines if an object is extensible
14	<a href="#"><u>Object.isFrozen()</u></a>	This method determines if an object was frozen.
15	<a href="#"><u>Object.isSealed()</u></a>	This method determines if an object is sealed.
16	<a href="#"><u>Object.keys()</u></a>	This method returns an array of a given object's own property names.
17	<a href="#"><u>Object.preventExtensions()</u></a>	This method is used to prevent any extensions of an object.
18	<a href="#"><u>Object.seal()</u></a>	This method prevents new properties from being added and marks all existing properties as non-configurable.
19	<a href="#"><u>Object.setPrototypeOf()</u></a>	This method sets the prototype of a specified object to another object.
20	<a href="#"><u>Object.values()</u></a>	This method returns an array of values.

Sonoo  
Vimal  
Ratan

## 2) JavaScript Array directly (new keyword)

The syntax of creating array directly is given below:

1. var **arrayname**=**new** Array();

Here, **new keyword** is used to create instance of array.

Let's see the example of creating array directly.

1. **<script>**
2. var i;
3. var emp = **new** Array();
4. emp[0] = "Arun";
5. emp[1] = "Varun";
6. emp[2] = "John";
- 7.
8. for (**i=0;i<emp.length;i++**) {
9. document.write(emp[i] + "**<br>**");
10. }
11. **</script>**

**Output of the above example**

Arun  
Varun  
John

## 3) JavaScript array constructor (new keyword)

Here, you need to create instance of array by passing arguments in constructor so that we don't have to provide value explicitly.

The example of creating object by array constructor is given below.

1. **<script>**
2. var emp=**new** Array("Jai","Vijay","Smith");

```
3. for (i=0;i<emp.length;i++){  
4. document.write(emp[i] + "<br>");  
5. }  
6. </script>
```

#### Output of the above example

```
Jai  
Vijay  
Smith
```

---

### JavaScript Array Methods

Let's see the list of JavaScript array methods with their description.

<u>forEach()</u>	It invokes the provided function once for each element of an array.
<u>includes()</u> <u>concat()</u>	It checks whether the given array contains the specified element. It returns a new array object that contains two or more merged arrays.
<u>indexOf()</u>	It searches the specified element in the given array and returns the index of the first match.
<u>isArray()</u> <u>entries()</u>	It tests if the passed value is an array. It creates an iterator object and a loop that iterates over each key/value
<u>join()</u>	It joins the elements of an array as a string.
<u>keys()</u>	It creates an iterator object that contains only the keys of the array,
<u>flat()</u>	the It creates a new array carrying sub-array elements concatenated loops through these keys. recursively till the specified depth.
<u>lastIndexOf()</u>	It searches the specified element in the given array and returns the index of the last match.
<u>map()</u>	into a new array. It calls the specified function for every array element and returns the new array.
<u>fill()</u>	array It fills elements into an array with static values.
<u>of()</u>	It creates a new array from a variable number of arguments, holding an type of argument.
<u>pop()</u>	provided It removes and returns the last element of an array. function conditions.
<u>push()</u>	It adds one or more elements to the end of an array.

<u>find()</u>	It returns the value of the first element in the given array that satisfies the specified condition.
<u>findIndex()</u>	It returns the index value of the first element in the given array that satisfies the specified condition.
<u>reverse()</u>	It reverses the elements of given array.
<u>reduce(function, initial)</u>	It executes a provided function for each value from left to right and reduces the array to a single value.

## JavaScript String

The **JavaScript string** is an object that represents a sequence of characters.

There are 2 ways to create string in JavaScript

1. By string literal
2. By string object (using new keyword)

### 1) By string literal

The string literal is created using double quotes. The syntax of creating string using string literal is given below:

1. var **stringname**="string value";

Let's see the simple example of creating string literal.

1. **<script>**
2. var **str**="This is string literal";
3. document.write(str);

<u>reduceRight()</u>	It executes a provided function for each value from right to left and reduces the array to a single value.
<u>some()</u>	It determines if any element of the array passes the test of the implemented function.
<u>shift()</u>	It removes and returns the first element of an array.
<u>slice()</u>	It returns a new array containing the copy of the part of the given array.
<u>sort()</u>	It returns the elements of the given array in a sorted order.
<u>splice()</u>	It adds/removes elements to/from the given array.
<u>toLocaleString()</u>	It returns a string containing all the elements of a specified array.
<u>toString()</u>	It converts the elements of a specified array into string form, without affecting the original array.
<u>unshift()</u>	It adds one or more elements in the beginning of the given array.
<u>values()</u>	It creates a new iterator object carrying values for each index in the array.

4. `</script>`

**Output:**

This is string literal

## 2) By string object (using new keyword)

The syntax of creating string object using new keyword is given below:

1. var **stringname**=new String("string literal");

Here, **new keyword** is used to create instance of string.

Let's see the example of creating string in JavaScript by new keyword.

1. **<script>**
2. var **stringname**=new String("hello javascript string");
3. document.write(stringname);
4. **</script>**

**Output:**

```
hello javascript string
```

---

## JavaScript String Methods

Let's see the list of JavaScript string methods with examples.

Methods	Description
<u><a href="#">charAt()</a></u>	It provides the char value present at the specified index.
<u><a href="#">charCodeAt()</a></u>	It provides the Unicode value of a character present at the specified index.
<u><a href="#">concat()</a></u>	It provides a combination of two or more strings.
<u><a href="#">indexOf()</a></u>	It provides the position of a char value present in the given string.
<u><a href="#">lastIndexOf()</a></u>	It provides the position of a char value present in the given string by searchin character from the last position.
<u><a href="#">search()</a></u>	It searches a specified regular expression in a given string and returns its pos if a match occurs.
<u><a href="#">match()</a></u>	It searches a specified regular expression in a given string and returns that re expression if a match occurs.
<u><a href="#">replace()</a></u>	It replaces a given string with the specified replacement.
<u><a href="#">substr()</a></u>	It is used to fetch the part of the given string on the basis of the specified starting position and length.
<u><a href="#">substring()</a></u>	It is used to fetch the part of the given string on the basis of the specified index.

### 1) JavaScript String **charAt(index)** Method

<u><a href="#">slice()</a></u>	It is used to fetch the part of the given string. It allows us to assign positive as well negative index.
<u><a href="#">toLowerCase()</a></u>	It converts the given string into lowercase letter.
<u><a href="#">toLocaleLowerCase()</a></u>	It converts the given string into lowercase letter on the basis of host current locale.
<u><a href="#">toUpperCase()</a></u>	It converts the given string into uppercase letter.
<u><a href="#">toLocaleUpperCase()</a></u>	It converts the given string into uppercase letter on the basis of host current locale.
<u><a href="#">toString()</a></u>	It provides a string representing the particular object.
<u><a href="#">valueOf()</a></u>	It provides the primitive value of string object.
<u><a href="#">split()</a></u>	It splits a string into substring array, then returns that newly created array.
<u><a href="#">trim()</a></u>	It trims the white space from the left and right side of the string.

The JavaScript String `charAt()` method returns the character at the given index.

1. [`<script>`](#)
2. [`var str="javascript";`](#)
3. [`document.write\(str.charAt\(2\)\);`](#)
4. [`</script>`](#)

**Output:**

v

## 2) JavaScript String concat(str) Method

The JavaScript String concat(str) method concatenates or joins two strings. 1.

<script>

```
2. var s1="javascript ";
3. var s2="concat example";
4. var s3=s1.concat(s2);
5. document.write(s3);
6. </script>
```

**Output:**

javascript concat example

## 3) JavaScript String indexOf(str) Method

The JavaScript String indexOf(str) method returns the index position of the given string.

```
1. <script>
2. var s1="javascript from javatpoint indexof";
3. var n=s1.indexOf("from");
4. document.write(n);
5. </script>
```

**Output:**

11

## JavaScript Date Object

The **JavaScript date** object can be used to get year, month and day. You can display a timer on the webpage by the help of JavaScript date object.

You can use different Date constructors to create date object. It provides methods to get and set day, month, year, hour, minute and seconds.

## Constructor

You can use 4 variant of Date constructor to create date object.

1. Date()
2. Date(milliseconds)
3. Date(dateString)
4. Date(year, month, day, hours, minutes, seconds, milliseconds)

---

## JavaScript Date Methods

Let's see the list of JavaScript date methods with their description.

|  |  |
|--|--|
|  | specified date on the basis of universal time.   |
| <u><a href="#">getUTCDay()</a></u>   | It returns the integer value between 0 and 6 that represents the day of the week on the basis of universal time.   |
| <u><a href="#">getUTCFullYear()</a></u><br><u><a href="#">getDate()</a></u>        | It returns the integer value that represents the year on the basis of universal time.<br>It returns the integer value between 1 and 31 that represents the day for time.   |
| <u><a href="#">getUTCHours()</a></u>   | It returns the integer value between 0 and 23 that represents the hours on the basis of universal time.  |
| <u><a href="#">getUTCMinutes()</a></u><br><u><a href="#">getFullYear()</a></u>     | It returns the integer value between 0 and 59 that represents the minutes on the basis of local time.<br>It returns the integer value that represents the year on the basis of local time.                               |
| <u><a href="#">getUTCMonth()</a></u>   | It returns the integer value between 0 and 11 that represents the month on the basis of universal time.  |
| <u><a href="#">getMilliseconds()</a></u><br><u><a href="#">getUTCSeconds()</a></u> | It returns the integer value between 0 and 999 that represents the milliseconds on the basis of universal time.<br>It returns the integer value between 0 and 60 that represents the seconds on the basis of local time. |
| <u><a href="#"> setDate()</a></u>  | It sets the day value for the specified date on the basis of local time.   |
| <u><a href="#">setDay()</a></u><br><u><a href="#">getMonth()</a></u>               | It sets the particular day of the week on the basis of local time.<br>It returns the integer value between 0 and 11 that represents the month on the basis of local time.  |
| <u><a href="#">setFullYear()</a></u>   | It sets the year value for the specified date on the basis of local time.  |

|                                   |   |
|-----------------------------------|---|
| <code>setMonth()</code>           | It sets the month value for the specified date on the basis of local time.                                      |
| <code>setSeconds()</code>         | basis or local time.<br>It sets the second value for the specified date on the basis of local time.             |
| <code>getUTCDate()</code>         | It returns the integer value between 1 and 31 that represents the day for                                       |
| <code>setUTCDate()</code>         | It sets the day value for the specified date on the basis of universal time.                                    |
| <code>setUTCDay()</code>          | It sets the particular day of the week on the basis of universal time.  |
| <code>setMilliseconds()</code>    | It sets the millisecond value for the specified date on the basis of local                                      |
| <code>setUTCFullYear()</code>     | It sets the year value for the specified date on the basis of universal time.                                   |
| <code>setUTCHours()</code>        | It sets the hour value for the specified date on the basis of universal time.                                   |
| <code>setUTCMilliseconds()</code> | It sets the millisecond value for the specified date on the basis of universal t                                |
| <code>setUTCMinutes()</code>      | It sets the minute value for the specified date on the basis of universal time.                                 |
| <code>setUTCMonth()</code>        | It sets the month value for the specified date on the basis of universal time.                                  |
| <code>setUTCSeconds()</code>      | It sets the second value for the specified date on the basis of universal time.                                 |
| <code>toDateString()</code>       | It returns the date portion of a Date object.   |
| <code>toISOString()</code>        | It returns the date in the form ISO format string.  |
| <code> toJSON()</code>            | It returns a string representing the Date object. It also serializes the Date Object during JSON serialization. |

|                       |   |
|-----------------------|---|
| <u>toString()</u>     | It returns the date in the form of string.                                |
| <u>toTimeString()</u> | It returns the time portion of a Date object.                             |
| <u>toUTCString()</u>  | It converts the specified date in the form of string using UTC time zone. |
| <u>valueOf()</u>      | It returns the primitive value of a Date object.                          |

### JavaScript Date Example

Let's see the simple example to print date object. It prints date and time both.

1. Current Date and Time: `<span id="txt"></span>`
2. `<script>`
3. `var today=new Date();`
4. `document.getElementById('txt').innerHTML=today;`
5. `</script>`

#### Output:

Current Date and Time: Mon Jul 15 2024 19:26:45 GMT+0500 (Pakistan Standard Time)

Let's see another code to print date/month/year.

1. `<script>`
2. `var date=new Date();`
3. `var day=date.getDate();`
4. `var month=date.getMonth()+1;`
5. `var year=date.getFullYear();`
6. `document.write("<br>Date is: "+day+"/"+month+"/"+year);`
7. `</script>`

**Output:**

Date is: 15/7/2024

### JavaScript Current Time Example

Let's see the simple example to print current time of system.

1. Current Time: `<span id="txt"></span>`
2. `<script>`
3. `var today=new Date();`
4. `var h=today.getHours();`
5. `var m=today.getMinutes();`
6. `var s=today.getSeconds();`
7. `document.getElementById('txt').innerHTML=h+":"+m+":"+s;`
8. `</script>`

**Output:**

Current Time: 19:26:45

### JavaScript Digital Clock Example

Let's see the simple example to display digital clock using JavaScript date object.

There are two ways to set interval in JavaScript: by `setTimeout()` or `setInterval()` method.

1. Current Time: `<span id="txt"></span>`
2. `<script>`
3. `window.onload=function(){getTime();}`
4. `function getTime(){`
5. `var today=new Date();`
6. `var h=today.getHours();`
7. `var m=today.getMinutes();`
8. `var s=today.getSeconds();`

```
9. // add a zero in front of numbers<10
10. m=checkTime(m);
11. s=checkTime(s);
12. document.getElementById('txt').innerHTML=h+":"+m+":"+s; 13.
    setTimeout(function(){getTime()},1000);
14. }
15. //setInterval("getTime()",1000);//another way
16. function checkTime(i){
17. if (i<10){
18. i="0" + i;
19. }
20. return i;
21. }
22. </script>
```

**Output:**

Current Time:

## JavaScript Math

The **JavaScript math** object provides several constants and methods to perform mathematical operation. Unlike date object, it doesn't have constructors.

### JavaScript Math Methods

Let's see the list of JavaScript Math methods with description.

## Math.sqrt(n)

The JavaScript math.sqrt(n) method returns the square root of the given number.

1. Square Root of 17 is: `<span id="p1"></span>`
2. `<script>`
3. `document.getElementById('p1').innerHTML=Math.sqrt(17);`
4. `</script>`

Output:

```
Square Root of 17 is: 4.123105625617661
```

## Math.random()

The JavaScript math.random() method returns the random number between 0 to 1.

| Methods                       | Description   |
|-------------------------------|---|
| <a href="#"><u>abs()</u></a>  | It returns the absolute value of the given number.                              |
| <a href="#"><u>acos()</u></a> | It returns the arccosine of the given number in radians.                        |
| <a href="#"><u>asin()</u></a> | It returns the arcsine of the given number in radians.                          |
| <a href="#"><u>atan()</u></a> | It returns the arc-tangent of the given number in radians.                      |
| <a href="#"><u>cbrt()</u></a> | It returns the cube root of the given number.                                   |
| <a href="#"><u>ceil()</u></a> | It returns a smallest integer value, greater than or equal to the given number. |
| <a href="#"><u>cos()</u></a>  | It returns the cosine of the given number.                                      |

1. Random Number is: `<span id="p2"></span>`

|                                 |  |
|---------------------------------|--|
| <u><a href="#">floor()</a></u>  | It returns largest integer value, lower than or equal to the given number. |
| <u><a href="#">exp()t()</a></u> | It returns the exponential form of the given numbers.                      |
| <u><a href="#">log()</a></u>    | It returns natural logarithm of a number.                                  |
| <u><a href="#">max()</a></u>    | It returns maximum value of the given numbers.                             |
| <u><a href="#">min()</a></u>    | It returns minimum value of the given numbers.                             |
| <u><a href="#">pow()</a></u>    | It returns value of base to the power of exponent.                         |
| <u><a href="#">random()</a></u> | It returns random number between 0 (inclusive) and 1 (exclusive).          |
| <u><a href="#">round()</a></u>  | It returns closest integer value of the given number.                      |
| <u><a href="#">sign()</a></u>   | It returns the sign of the given number                                    |
| <u><a href="#">sin()</a></u>    | It returns the sine of the given number.                                   |
| <u><a href="#">sinh()</a></u>   | It returns the hyperbolic sine of the given number.                        |
| <u><a href="#">sqrt()</a></u>   | It returns the square root of the given number                             |
| <u><a href="#">tan()</a></u>    | It returns the tangent of the given number.                                |
| <u><a href="#">tanh()</a></u>   | It returns the hyperbolic tangent of the given number.                     |

|                       |   |
|-----------------------|---|
| <b><u>trunc()</u></b> | It returns an integer part of the given number. |
|-----------------------|---|

2. **<script>**
3. **document.getElementById('p2').innerHTML=Math.random();**
4. **</script>**

Output:

Random Number is: 0.48099396004801487

### **Math.pow(m,n)**

The JavaScript math.pow(m,n) method returns the m to the power n that is  $m^n$ .

1. 3 to the power of 4 is: **<span id="p3"></span>**
2. **<script>**
3. **document.getElementById('p3').innerHTML=Math.pow(3,4);**
4. **</script>**

Output:

3 to the power of 4 is: 81

ADVERTISEMENT

### **Math.floor(n)**

The JavaScript math.floor(n) method returns the lowest integer for the given number. For example 3 for 3.7, 5 for 5.9 etc.

1. Floor of 4.6 is: **<span id="p4"></span>**
2. **<script>**
3. **document.getElementById('p4').innerHTML=Math.floor(4.6);**
4. **</script>**

Output:

Floor of 4.6 is: 4

### **Math.ceil(n)**

The JavaScript `math.ceil(n)` method returns the largest integer for the given number. For example 4 for 3.7, 6 for 5.9 etc.

1. Ceil of 4.6 is: `<span id="p5"></span>`
2. `<script>`
3. `document.getElementById('p5').innerHTML=Math.ceil(4.6);`
4. `</script>`

Output:

```
Ceil of 4.6 is: 5
```

### **Math.round(n)**

The JavaScript `math.round(n)` method returns the rounded integer nearest for the given number. If fractional part is equal or greater than 0.5, it goes to upper value 1 otherwise lower value 0. For example 4 for 3.7, 3 for 3.3, 6 for 5.9 etc.

1. Round of 4.3 is: `<span id="p6"></span><br>`
2. Round of 4.7 is: `<span id="p7"></span>`
3. `<script>`
4. `document.getElementById('p6').innerHTML=Math.round(4.3);`
5. `document.getElementById('p7').innerHTML=Math.round(4.7);`
6. `</script>`

Output:

```
Round of 4.3 is: 4
Round of 4.7 is: 5
```

### **Math.abs(n)**

The JavaScript `math.abs(n)` method returns the absolute value for the given number. For example 4 for -4, 6.6 for -6.6 etc.

1. Absolute value of -4 is: `<span id="p8"></span>`
2. `<script>`
3. `document.getElementById('p8').innerHTML=Math.abs(-4);`
4. `</script>`

Output:

Absolute value of -4 is: 4

## JavaScript Number Object

The **JavaScript number** object *enables you to represent a numeric value*. It may be integer or floating-point. JavaScript number object follows IEEE standard to represent the floating-point numbers.

By the help of `Number()` constructor, you can create number object in JavaScript. For example:

1. `var n=new Number(value);`

If value can't be converted to number, it returns `NaN`(Not a Number) that can be checked by `isNaN()` method.

You can direct assign a number to a variable also. For example:

1. `var x=102;//integer value`
2. `var y=102.7;//floating point value`
3. `var z=13e4;//exponent value, output: 130000`
4. `var n=new Number(16);//integer value by number object`

**Output:**

102 102.7 130000 16

## JavaScript Number Constants

Let's see the list of JavaScript number constants with description.

---

## JavaScript Number Methods

Let's see the list of JavaScript number methods with their description.

| Constant          | Description                                |
|-------------------|--|
| MIN_VALUE         | returns the largest minimum value.         |
| MAX_VALUE         | returns the largest maximum value.         |
| POSITIVE_INFINITY | returns positive infinity, overflow value. |
| NEGATIVE_INFINITY | returns negative infinity, overflow value. |
| NaN               | represents "Not a Number" value.           |

## JavaScript Boolean

| Methods                                | Description   |
|--|---|
| <a href="#"><u>isFinite()</u></a>      | It determines whether the given value is a finite number.                       |
| <a href="#"><u>isInteger()</u></a>     | It determines whether the given value is an integer.                            |
| <a href="#"><u>parseFloat()</u></a>    | It converts the given string into a floating point number.                      |
| <a href="#"><u>parseInt()</u></a>      | It converts the given string into an integer number.                            |
| <a href="#"><u>toExponential()</u></a> | It returns the string that represents exponential notation of the given number. |

**JavaScript Boolean** is an object that represents value in two states: *true* or *false*. You can create the JavaScript Boolean object by Boolean() constructor as given below.

|                                      |   |
|--------------------------------------|---|
| <u><a href="#">toFixed()</a></u>     | It returns the string that represents a number with exact digits after a decimal point. |
| <u><a href="#">toPrecision()</a></u> | It returns the string representing a number of specified precision.                     |
| <u><a href="#">toString()</a></u>    | It returns the given number in the form of string.                                      |
|                                      |   |

1. Boolean **b=new Boolean(value);**

The default value of JavaScript Boolean object is *false*.

### JavaScript Boolean Example

1. **<script>**
2. **document.write(10<20);//true**
3. **document.write(10<5);//false**
4. **</script>**

## JavaScript BOM Browser Object Model

The **Browser Object Model (BOM)** is used to interact with the browser.

The default object of browser is **window** means you can call all the functions of **window** by specifying **window** or directly. For example:

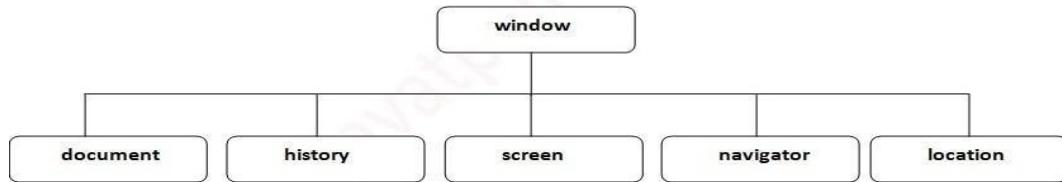
1. **window.alert("hello javatpoint");** is

same as:

1. **alert("hello javatpoint");**

You can use a lot of properties (other objects) defined underneath the **window** object like **document, history, screen, navigator, location, innerHeight, innerWidth**,

**Note:** The document object represents an html document. It forms DOM (Document Object Model).



## Window Object

The **window object** represents a window in browser. An object of window is created automatically by the browser.

Window is the object of browser, **it is not the object of javascript**. The javascript objects are string, array, date etc.

**Note:** if html document contains frame or iframe, browser creates additional window objects for each frame.

### Methods of window object

The important methods of window object are as follows:

| Method    | Description   |
|-----------|---|
| alert()   | displays the alert box containing message with ok button.                     |
| confirm() | displays the confirm dialog box containing message with ok and cancel button. |
| prompt()  | displays a dialog box to get input from the user.                             |
| open()    | opens the new window.   |

---

**Example of alert() in javascript**

|              |   |
|--------------|---|
| close()      | closes the current window.  |
| setTimeout() | performs action after specified time like calling function, evaluating expressions etc. |

It displays alert dialog box. It has message and ok button.

1. `<script type="text/javascript">`
2. `function msg(){`
3. `alert("Hello Alert Box");`
4. `}`
5. `</script>`
6. `<input type="button" value="click" onclick="msg()" />`

***Output of the above example***

[www.javatpoint.com](http://www.javatpoint.com) says

Hello Alert Box



### ***Example of confirm() in javascript***

It displays the confirm dialog box. It has message with ok and cancel buttons.

1. `<script type="text/javascript">`
2. `function msg(){`
3. `var v= confirm("Are u sure?");`
4. `if(v==true){`
5. `alert("ok");`
6. `}`
7. `else{`
8. `alert("cancel");`
9. `}`
10.
11. `}`

12. </script>
- 13.
14. <input type="button" value="delete record" onclick="msg()" />

***Output of the above example***

www.javatpoint.com says

Are u sure?

***Example of prompt() in javascript***

It displays prompt dialog box for input. It has message and textfield.

1. <script type="text/javascript">
2. function msg(){
3. var v= prompt("Who are you?");
4. alert("I am "+v);
- 5.
6. }
7. </script>
- 8.
9. <input type="button" value="click" onclick="msg()" />

***Output of the above example***

www.javatpoint.com says

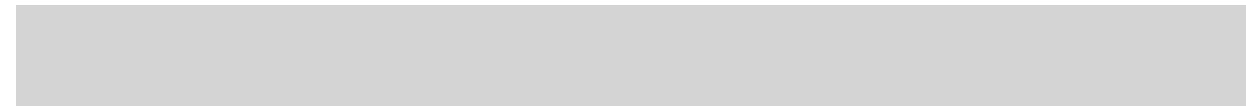
Who are you?

***Example of open() in javascript***

It displays the content in a new window.

1. `<script type="text/javascript">`
2. `function msg(){`
3. `open("http://www.javatpoint.com");`
4. `}`
5. `</script>`
6. `<input type="button" value="javatpoint" onclick="msg()" />`

#### ***Output of the above example***



#### ***Example of setTimeout() in javascript***

It performs its task after the given milliseconds.

1. `<script type="text/javascript">`
2. `function msg(){`
3. `setTimeout(`
4. `function(){`
5. `alert("Welcome to Javatpoint after 2 seconds")`
6. `,2000);`
7.
8. `}`
9. `</script>`
10.
11. `<input type="button" value="click" onclick="msg()" />`

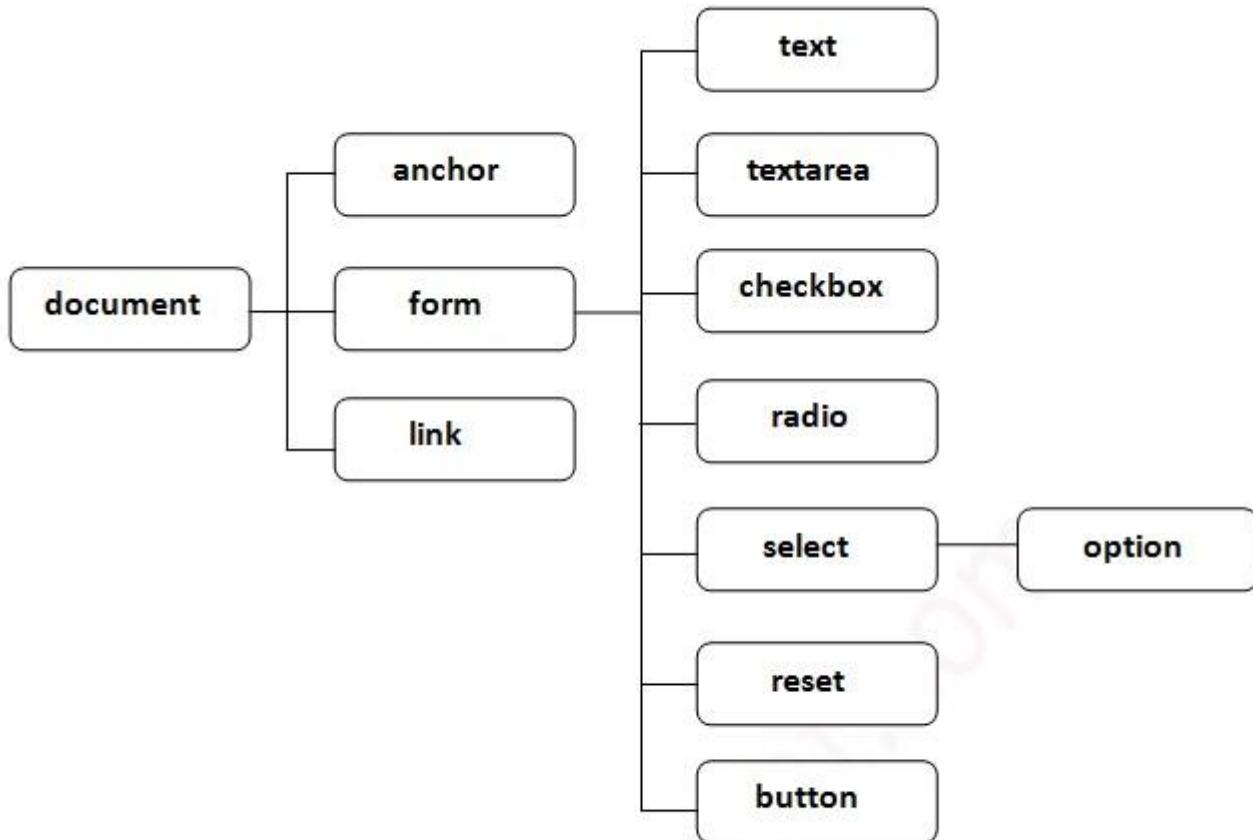
## **JavaScript DOM Document Object Model**

The **document object** represents the whole html document.

When html document is loaded in the browser, it becomes a document object. It is the **root element** that represents the html document. It has properties and methods. By the help of document object, we can add dynamic content to our web page.

## Properties of document object

Let's see the properties of document object that can be accessed and modified by the document object.



## Methods of document object

We can access and change the contents of document by its methods.

The important methods of document object are as follows:

| Method          | Description                              |
|-----------------|--|
| write("string") | writes the given string on the document. |

|                          |  |
|--------------------------|--|
| writeln("string")        | writes the given string on the document with newline character at the end. |
| getElementById()         | returns the element having the given id value.                             |
| getElementsByName()      | returns all the elements having the given name value.                      |
| getElementsByTagName()   | returns all the elements having the given tag name.                        |
| getElementsByClassName() | returns all the elements having the given class name.                      |

### Accessing field value by document object

In this example, we are going to get the value of input text by user. Here, we are using **document.form1.name.value** to get the value of name field. Here, **document** is the root element that represents the html document. **form1** is the name of the form. **name** is the attribute name of the input text. **value** is the property, that returns the value of the input text.

Let's see the simple example of document object that prints name with welcome message.

1. `<script type="text/javascript">`
2. `function printvalue(){`
3. `var name=document.form1.name.value;`
4. `alert("Welcome: "+name);`
5. `}`
6. `</script>`
- 7.
8. `<form name="form1">`
9. `Enter Name:<input type="text" name="name"/>`
10. `<input type="button" onclick="printvalue()" value="print name"/>`
11. `</form>`

---

***Output of the above example***

Enter Name:  print

## Javascript - innerHTML

The **innerHTML** property can be used to write the dynamic html on the html document.

It is used mostly in the web pages to generate the dynamic html such as registration form, comment form, links etc.

### Example of innerHTML property

In this example, we are going to create the html form when user clicks on the button.

In this example, we are dynamically writing the html form inside the div name having the id mylocation. We are identifying this position by calling the document.getElementById() method.

1. `<script type="text/javascript" >`
2. `function showcommentform() {`
3. `var data="Name:<input type='text' name='name'><br>Comment:<br><textarea rows='5' cols='80'></textarea>`
4. `<br><input type='submit' value='Post Comment'>";`
5. `document.getElementById('mylocation').innerHTML=data; 6. }`
7. `</script>`
8. `<form name="myForm">`
9. `<input type="button" value="comment" onclick="showcommentform()">`
10. `<div id="mylocation"></div>`
11. `</form>`

---

***Output of the above example***

The screenshot shows a simple comment form within a green header bar. The form consists of three main parts: a text input field labeled "Name:", a larger text area labeled "Comment:", and a submit button labeled "comment".

### Show/Hide Comment Form Example using innerHTML

1. <!DOCTYPE html>
2. <html>
3. <head>
4. <title>First JS</title>
5. <script>
6. var flag=true;
7. function commentform(){
8. var cform="
9. Enter Email:<br><input type='email' name='email' /><br>Enter Comment:<br/>
10. <textarea rows='5' cols='70'></textarea><br><input type='submit' value='Post Comment' /></form>";
11. if(flag){
12. document.getElementById("mylocation").innerHTML=cform;
13. flag=false;
14. }else{
15. document.getElementById("mylocation").innerHTML="";
16. flag=true;
17. }
18. }
19. </script>
20. </head>
21. <body>

22. `<button onclick="commentform()">Comment</button>`
23. `<div id="mylocation"></div>`
24. `</body>`
25. `</html>`

## **JavaScript Form Validation**

It is important to validate the form submitted by the user because it can have inappropriate values. So, validation is must to authenticate user.

JavaScript provides facility to validate the form on the client-side so data processing will be faster than server-side validation. Most of the web developers prefer JavaScript form validation.

Through JavaScript, we can validate name, password, email, date, mobile numbers and more fields.

---

### **JavaScript Form Validation Example**

In this example, we are going to validate the name and password. The name can't be empty and password can't be less than 6 characters long.

Here, we are validating the form on form submit. The user will not be forwarded to the next page until given values are correct.

1. `<script>`
2. `function validateform(){`
3. `var name=document.myform.name.value; 4. var`  
`password=document.myform.password.value;`
- 5.
6. `if (name==null || name==""){`
7. `alert("Name can't be blank");`
8. `return false;`
9. `}else if(password.length<6){`
10. `alert("Password must be at least 6 characters long.");`
11. `return false;`
12. `}`
13. `}`

```

14. </script>
15. <body>
16. <form name="myform" method="post" action="abc.jsp" onsubmit="return validateform()" >
17. Name: <input type="text" name="name"><br/>
18. Password: <input type="password" name="password"><br/> 19. <input type="submit"
   value="register">
20. </form>

```

**Output:**

The screenshot shows a simple registration form. It consists of two text input fields stacked vertically. The top field is labeled "Name:" and the bottom field is labeled "Password:". Below these fields is a single button labeled "register". The entire form is contained within a light gray rectangular area.

**JavaScript Retype Password Validation**

```

1. <script type="text/javascript">
2. function matchpass(){
3. var firstpassword=document.f1.password.value; 4. var
   secondpassword=document.f1.password2.value;
5.
6. if(firstpassword==secondpassword){
7. return true;
8. }
9. else{
10. alert("password must be same!");
11. return false;
12. }
13. }
14. </script>
15.
16. <form name="f1" action="register.jsp" onsubmit="return matchpass()"> 17.
   Password:<input type="password" name="password" /><br/>

```

18. Re-enter Password:<input type="password" name="password2"/><br/>
  19. <input type="submit">
  20. </form>
- 

### JavaScript Number Validation

Let's validate the textfield for numeric value only. Here, we are using isNaN() function.

1. <script>
  2. function validate(){
  3. var num=document.myform.num.value;
  4. if (isNaN(num)){
  5. document.getElementById("numloc").innerHTML="Enter Numeric value only";
  6. return false;
  7. }else{
  8. return true;
  9. }
  10. }
  11. </script>
  12. <form name="myform" onsubmit="return validate()" >
  13. Number: <input type="text" name="num"><span id="numloc"></span><br/>
  14. <input type="submit" value="submit">
  15. </form>
- 

### JavaScript validation with image

Let's see an interactive JavaScript form validation example that displays correct and incorrect image if input is correct or incorrect.

1. <script>
2. function validate(){
3. var name=document.f1.name.value;
4. var password=document.f1.password.value;
5. var status=false;

```
6.  
7. if(name.length<1){  
8. document.getElementById("nameloc").innerHTML=  
9. " <img src='unchecked.gif'> Please enter your name";  
10. status=false;  
11. }else{  
12. document.getElementById("nameloc").innerHTML=" <img src='checked.gif'>";  
13. status=true;  
14. }  
15. if(password.length<6){  
16. document.getElementById("passwordloc").innerHTML=  
17. " <img src='unchecked.gif'> Password must be at least 6 char long";  
18. status=false;  
19. }else{  
20. document.getElementById("passwordloc").innerHTML=" <img src='checked.gif'>";  
21. }  
22. return status;  
23. }  
24. </script>  
25.  
26. <form name="f1" action="#" onsubmit="return validate()">  
27. <table>  
28. <tr><td>Enter Name:</td><td><input type="text" name="name"/>  
29. <span id="nameloc"></span></td></tr>  
30. <tr><td>Enter Password:</td><td><input type="password" name="password"/>  
31. <span id="passwordloc"></span></td></tr>  
32. <tr><td colspan="2"><input type="submit" value="register"/></td></tr>  
33. </table>  
34. </form> Output:
```

Enter Name:

Enter Password:

register

---

### JavaScript email validation

We can validate the email by the help of JavaScript.

There are many criteria that need to be follow to validate the email id such as:

- o email id must contain the @ and . character o  
There must be at least one character before and after  
the @. o There must be at least two characters after .  
(dot).

Let's see the simple example to validate the email field.

```

1. <script>
2. function validateemail()
3. {
4. var x=document.myform.email.value;
5. var atposition=x.indexOf("@");
6. var dotposition=x.lastIndexOf(".");
7. if (atposition<1 || dotposition<atposition+2 || dotposition+2>=x.length){
8. alert("Please enter a valid email address \n atpostion:"+atposition+"\n
dotposition:"+dotposition); 9. return false;
10. }
11. }
12. </script>
13. <body>
14. <form name="myform" method="post" action="#" onsubmit="return validateemail();"> 15.
Email: <input type="text" name="email"><br/>
16.

```

17. `<input type="submit" value="register">`
18. `</form>`

## JavaScript OOPs

### JavaScript Classes

In JavaScript, classes are the special type of functions. We can define the class just like function declarations and function expressions.

The JavaScript class contains various class members within a body including methods or constructor. The class is executed in strict mode. So, the code containing the silent error or mistake throws an error.

The class syntax contains two components:

- o Class declarations
- o Class expressions

#### Class Declarations

A class can be defined by using a class declaration. A class keyword is used to declare a class with any particular name. According to JavaScript naming conventions, the name of the class always starts with an uppercase letter.

#### Class Declarations Example

Let's see a simple example of declaring the class.

1. `<script>`
2. `//Declaring class`
3. `class Employee`
4. `{`
5. `//Initializing an object`
6. `constructor(id,name)`
7. `{`
8. `this.id=id;`
9. `this.name=name;`

```

10.    }
11.    //Declaring method
12.    detail()
13.    {
14.        document.writeln(this.id+" "+this.name+"<br>")
15.    }
16.    }
17.    //passing object to a variable
18.    var e1=new Employee(101,"Martin Roy");
19.    var e2=new Employee(102,"Duke William");
20.    e1.detail(); //calling method
21.    e2.detail();
22.    </script>

```

**Output:**

```

101 Martin Roy
102 Duke William

```

### Class expressions

Another way to define a class is by using a class expression. Here, it is not mandatory to assign the name of the class. So, the class expression can be named or unnamed. The class expression allows us to fetch the class name. However, this will not be possible with class declaration.

#### Unnamed Class Expression

The class can be expressed without assigning any name to it.

Let's see an example.

```

1.    <script>
2.    var emp = class {
3.        constructor(id, name) {
4.            this.id = id;
5.            this.name = name;
6.        }
7.    };

```

```
8. document.writeln(emp.name);
9. </script>
```

**Output:**

```
emp
```

**Class Expression Example: Re-declaring Class**

Unlike class declaration, the class expression allows us to re-declare the same class. So, if we try to declare the class more than one time, it throws an error.

```
1. <script>
2. //Declaring class
3. var emp=class
4. {
5. //Initializing an object
6. constructor(id,name)
7. {
8. this.id=id;
9. this.name=name;
10. }
11. //Declaring method
12. detail()
13. {
14. document.writeln(this.id+" "+this.name+"<br>")
15. }
16. }
17. //passing object to a variable
18. var e1=new emp(101,"Martin Roy");
19. var e2=new emp(102,"Duke William");
20. e1.detail(); //calling method
21. e2.detail();
22.
23. //Re-declaring class
```

```

24. var emp=class
25. {
26.   //Initializing an object
27.   constructor(id,name)
28.   {
29.     this.id=id;
30.     this.name=name;
31.   }
32.   detail()
33.   {
34.     document.writeln(this.id+" "+this.name+"<br>")
35.   }
36. }
37. //passing object to a variable
38. var e1=new emp(103,"James Bella");
39. var e2=new emp(104,"Nick Johnson");
40. e1.detail(); //calling method
41. e2.detail();
42. </script>

```

**Output:**

```

101 Martin Roy
102 Duke William
103 James Bella
104 Nick Johnson

```

**Named Class Expression Example**

We can express the class with the particular name. Here, the scope of the class name is up to the class body. The class is retrieved using class.name property.

```

1. <script>
2. var emp = class Employee {
3.   constructor(id, name) {
4.     this.id = id;

```

```

5.   this.name = name;
6. }
7. );
8. document.writeln(emp.name); 9. /*document.writeln(Employee.name);
10. Error occurs on console:
11. "ReferenceError: Employee is not defined
12. */
13. </script>

```

## JavaScript Objects

A JavaScript object is an entity having state and behavior (properties and method). For example: car, pen, bike, chair, glass, keyboard, monitor etc.

JavaScript is an object-based language. Everything is an object in JavaScript.

JavaScript is template based not class based. Here, we don't create class to get the object. But, we direct create objects.

### Creating Objects in JavaScript

There are 3 ways to create objects.

1. By object literal
2. By creating instance of Object directly (using new keyword)
3. By using an object constructor (using new keyword)

#### 1) JavaScript Object by object literal

The syntax of creating object using object literal is given below:

1. **object**={property1:value1,property2:value2.....propertyN:valueN}

As you can see, property and value is separated by : (colon).

Let's see the simple example of creating object in JavaScript.

1. `<script>`
2. `emp={id:102,name:"Shyam Kumar",salary:40000}`
3. `document.write(emp.id+" "+emp.name+" "+emp.salary);`
4. `</script>`

***Output of the above example***

```
102 Shyam Kumar 40000
```

## 2) By creating instance of Object

The syntax of creating object directly is given below:

1. `var objectname=new Object();`

Here, **new keyword** is used to create object.

Let's see the example of creating object directly.

1. `<script>`
2. `var emp=new Object();`
3. `emp.id=101;`
4. `emp.name="Ravi Malik";`
5. `emp.salary=50000;`
6. `document.write(emp.id+" "+emp.name+" "+emp.salary);`
7. `</script>`

***Output of the above example***

```
101 Ravi 50000
```

## 3) By using an Object constructor

Here, you need to create function with arguments. Each argument value can be assigned in the current object by using this keyword.

The **this keyword** refers to the current object.

The example of creating object by object constructor is given below.

1. `<script>`

```

2. function emp(id,name,salary){
3.   this.id=id;
4.   this.name=name;
5.   this.salary=salary;
6. }
7. e=new emp(103,"Vimal Jaiswal",30000);
8.
9. document.write(e.id+" "+e.name+" "+e.salary);
10. </script>

```

***Output of the above example***

103 Vimal Jaiswal 30000

### Defining method in JavaScript Object

We can define method in JavaScript object. But before defining method, we need to add property in the function with same name as method.

The example of defining method in object is given below.

```

1. <script>
2. function emp(id,name,salary){
3.   this.id=id;
4.   this.name=name; 5. this.salary=salary;
6.
7. this.changeSalary=changeSalary;
8. function changeSalary(otherSalary){
9.   this.salary=otherSalary;
10. }
11. }
12. e=new emp(103,"Sonoo Jaiswal",30000);
13. document.write(e.id+" "+e.name+" "+e.salary);
14. e.changeSalary(45000);
15. document.write("<br>"+e.id+" "+e.name+" "+e.salary);
16. </script>

```

## JavaScript Constructor Method

A JavaScript constructor method is a special type of method which is used to initialize and create an object. It is called when memory is allocated for an object.

- Points to remember**
  - The constructor keyword is used to declare a constructor method.
  - The class can contain one constructor method only.
  - JavaScript allows us to use parent class constructor through super keyword.

### Constructor Method Example

Let's see a simple example of a constructor method.

```

1. <script>
2. class Employee {
3.   constructor() {
4.     this.id=101;
5.     this.name = "Martin Roy";
6.   }
7. }
8. var emp = new Employee();
9. document.writeln(emp.id+ " "+emp.name);
10. </script> Output:
```

101 Martin Roy

### Constructor Method Example: super keyword

The super keyword is used to call the parent class constructor. Let's see an example.

```

1. <script>
2. class CompanyName
3. {
4.   constructor()
5.   {
6.     this.company="Javatpoint";
```

```

7. }
8. }
9. class Employee extends CompanyName {
10. constructor(id,name) {
11. super();
12. this.id=id;
13. this.name=name;
14. }
15. }
16. var emp = new Employee(1,"John");
17. document.writeln(emp.id+ " "+emp.name+ " "+emp.company);
18. </script>

```

**Output:**

1 John Javatpoint

## JavaScript Encapsulation

The JavaScript Encapsulation is a process of binding the data (i.e. variables) with the functions acting on that data. It allows us to control the data and validate it. To achieve an encapsulation in JavaScript: -

- Use var keyword to make data members private.
- Use setter methods to set the data and getter methods to get that data.

The encapsulation allows us to handle an object using the following properties:

**Read/Write** - Here, we use setter methods to write the data and getter methods read that data.

**Read Only** - In this case, we use getter methods only.

**Write Only** - In this case, we use setter methods only.

### JavaScript Encapsulation Example

Let's see a simple example of encapsulation that contains two data members with its setter and getter methods.

1. <script>

```
2. class Student
3. {
4.     constructor()
5.     {
6.         var name;
7.         var marks;
8.     }
9.     getName()
10.    {
11.        return this.name;
12.    }
13.    setName(name)
14.    {
15.        this.name=name;
16.    }
17.
18.    getMarks()
19.    {
20.        return this.marks;
21.    }
22.    setMarks(marks)
23.    {
24.        this.marks=marks;
25.    }
26.
27.    }
28. var stud=new Student();
29. stud.setName("John");
30. stud.setMarks(80);
31. document.writeln(stud.getName()+" "+stud.getMarks());
32. </script>
```

**Output:**

John 80

## JavaScript Inheritance

The JavaScript inheritance is a mechanism that allows us to create new classes on the basis of already existing classes. It provides flexibility to the child class to reuse the methods and variables of a parent class.

The JavaScript **extends** keyword is used to create a child class on the basis of a parent class. It facilitates child class to acquire all the properties and behavior of its parent class.

**Points to remember**

- It maintains

an IS-A relationship.

- The extends keyword is used in class expressions or class declarations.
- Using extends keyword, we can acquire all the properties and behavior of the inbuilt object as well as custom classes.
- We can also use a prototype-based approach to achieve inheritance.

### JavaScript extends Example: inbuilt object

In this example, we extends **Date** object to display today's date.

1. **<script>**
2. class Moment extends Date {
3. constructor() {
4. super();
5. }}
6. var m=new Moment();
7. document.writeln("Current date:")
8. document.writeln(m.getDate()+"-"+(m.getMonth()+1)+"-"+m.getFullYear()); 9. **</script>**

### Output:

Current date: 31-8-2018

Let's see one more example to display the year value from the given date.

1. **<script>**
2. class Moment extends Date {
3. constructor(year) {

```

4.     super(year);
5.   }
6.   var m=new Moment("August 15, 1947 20:22:10");
7.   document.writeln("Year value:")
8.   document.writeln(m.getFullYear());
9. </script>

```

**Output:**

Year value: 1947

**JavaScript extends Example: Custom class**

In this example, we declare sub-class that extends the properties of its parent class.

```

1. <script>
2. class Bike
3. {
4.   constructor()
5.   {
6.     this.company="Honda";
7.   }
8. }
9. class Vehicle extends Bike {
10.  constructor(name,price) {
11.    super();
12.    this.name=name;
13.    this.price=price;
14.  }
15. }
16. var v = new Vehicle("Shine","70000");
17. document.writeln(v.company+" "+v.name+" "+v.price);
18. </script>

```

**Output:**

Honda Shine 70000

## JavaScript extends Example: a Prototype-based approach

Here, we perform prototype-based inheritance. In this approach, there is no need to use class and extends keywords.

```

1. <script>
2. //Constructor function
3. function Bike(company)
4. {
5.   this.company=company;
6. }
7.
8. Bike.prototype.getCompany=function()
9. {
10. return this.company;
11. }
12. //Another constructor function
13. function Vehicle(name,price) {
14.   this.name=name;
15.   this.price=price;
16. }
17. var bike = new Bike("Honda");
18. Vehicle.prototype=bike; //Now Bike treats as a parent of Vehicle.
19. var vehicle=new Vehicle("Shine",70000);
20. document.writeln(vehicle.getCompany()+" "+vehicle.name+" "+vehicle.price);
21. </script> Test it Now Output:
```

Honda Shine 70000

## JavaScript Polymorphism

The polymorphism is a core concept of an object-oriented paradigm that provides a way to perform a single action in different forms. It provides an ability to call the same method on different JavaScript objects. As JavaScript is not a type-safe language, we can pass any type of data members with the methods.

## JavaScript Polymorphism Example 1

Let's see an example where a child class object invokes the parent class method.

```

1. <script>
2. class A
3. {
4.   display()
5.   {
6.     document.writeln("A is invoked");
7.   }
8. }
9. class B extends A
10. {
11. }
12. var b=new B();
13. b.display();
14. </script>
```

### Output:

A is invoked

## JavaScript Abstraction

An abstraction is a way of hiding the implementation details and showing only the functionality to the users. In other words, it ignores the irrelevant details and shows only the required one.

**Points to remember** ○ We cannot create an instance of Abstract Class. ○ It reduces the duplication of code.

## JavaScript Abstraction Example

### Example 1

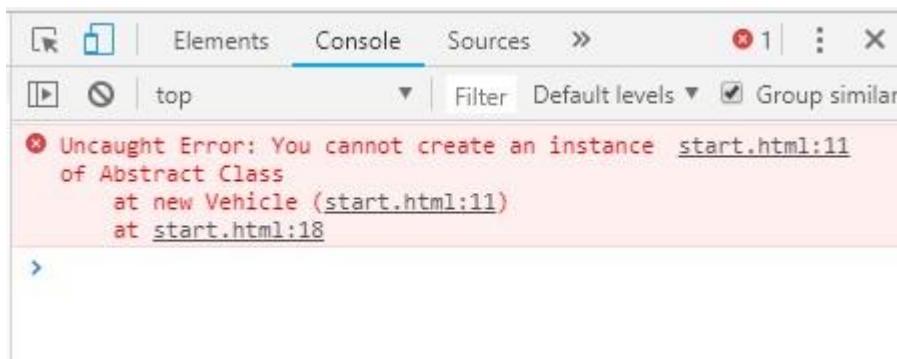
Let's check whether we can create an instance of Abstract class or not.

```
1. <script>
```

```

2. //Creating a constructor function
3. function Vehicle()
4. {
5.     this.vehicleName= vehicleName;
6.     throw new Error("You cannot create an instance of Abstract class");
7.
8. }
9. Vehicle.prototype.display=function()
10. {
11.     return this.vehicleName;
12. }
13. var vehicle=new Vehicle();
14. </script>

```



## JavaScript Cookies

### JavaScript Cookies

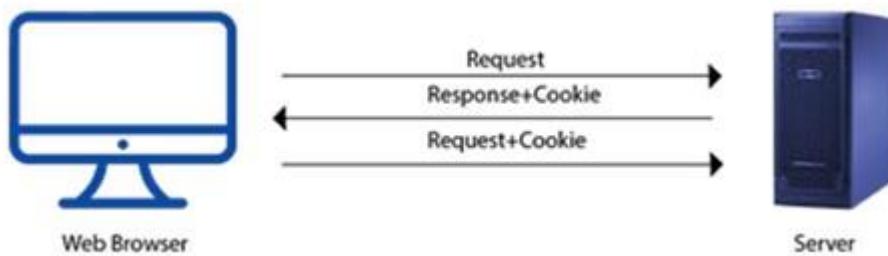
A cookie is an amount of information that persists between a server-side and a clientside. A web browser stores this information at the time of browsing.

A cookie contains the information as a string generally in the form of a name-value pair separated by semi-colons. It maintains the state of a user and remembers the user's information among all the web pages.

#### How Cookies Works?

- When a user sends a request to the server, then each of that request is treated as a new request sent by the different user.

- So, to recognize the old user, we need to add the cookie with the response from the server.
- browser at the client-side.
- Now, whenever a user sends a request to the server, the cookie is added with that request automatically. Due to the cookie, the server recognizes the users.



### How to create a Cookie in JavaScript?

In JavaScript, we can create, read, update and delete a cookie by using **document.cookie** property.

The following syntax is used to create a cookie:

1. `document.cookie="name=value";`

### JavaScript Cookie Example

#### Example 1

Let's see an example to set and get a cookie.

```

1.   <!DOCTYPE html>
2.   <html>
3.   <head>
4.   </head>
5.   <body>
6.   <input type="button" value="setCookie" onclick="setCookie()">
7.   <input type="button" value="getCookie" onclick="getCookie()">
8.   <script>
9.   function setCookie()
10.  {

```

```
11. document.cookie="username=Duke Martin";
12. }
13. function getCookie()
14. {
15. if(document.cookie.length!=0)
16. {
17. alert(document.cookie);
18. }
19. else
20. {
21. alert("Cookie not available");
22. }
23. } 24. </script>
25.
26. </body>
27. </html>
```

## Example 2

Here, we display the cookie's name-value pair separately.

```
1.      <!DOCTYPE html>
2.      <html>
3.      <head>
4.      </head>
5.      <body>
6.      <input type="button" value="setCookie" onclick="setCookie()">
7.      <input type="button" value="getCookie" onclick="getCookie()">
8.      <script>
9.      function setCookie()
10.     {
11.       document.cookie="username=Duke Martin";
12.     }
```

```
13.     function getCookie()
14.     {
15.         if(document.cookie.length!=0)
16.         {
17.             var array=document.cookie.split("=");
18.             alert("Name="+array[0]+" "+Value="+array[1]);
19.         }
20.     else
21.     {
22.         alert("Cookie not available");
23.     }
24. }
25. </script>
26.
27. </body>
28. </html>
```

## Deleting a Cookie in JavaScript

In the previous section, we learned the different ways to set and update a cookie in JavaScript. Apart from that, JavaScript also allows us to delete a cookie. Here, we see all the possible ways to delete a cookie.

### Different ways to delete a Cookie

These are the following ways to delete a cookie:

- A cookie can be deleted by using expire attribute. ○  
A cookie can also be deleted by using max-age attribute.
- We can delete a cookie explicitly, by using a web browser.

## Examples to delete a Cookie

### Example 1

In this example, we use expire attribute to delete a cookie by providing expiry date (i.e. any past date) to it.

```
1. <!DOCTYPE html>
2. <html>
3. <head>
4. </head>
5. <body>
6.
7. <input type="button" value="Set Cookie" onclick="setCookie()">
8. <input type="button" value="Get Cookie" onclick="getCookie()">
9. <script>
10. function setCookie()
11. {
12.   document.cookie="name=Martin Roy; expires=Sun, 20 Aug 2000 12:00:00 UTC";
13.
14. }
15. function getCookie()
16. {
17.   if(document.cookie.length!=0)
18.   {
19.     alert(document.cookie);
20.   }
21.   else
22.   {
23.     alert("Cookie not available");
24.   }
25. }
26. </script>
27. </body>
```

28. </html>

## **JavaScript Events**

### **JavaScript Events**

The change in the state of an object is known as an **Event**. In html, there are various events which represents that some activity is performed by the user or by the browser. When **javascript** code is included in **HTML**, js react over these events and allow the execution. This process of reacting over the events is called **Event Handling**. Thus, js handles the HTML events via **Event Handlers**.

**For example**, when a user clicks over the browser, add js code, which will execute the task to be performed on the event.

### **Click Event**

```

1. <html>
2. <head> Javascript Events </head>
3. <body>
4. <script language="Javascript" type="text/Javascript">
5.   <!--
6.   function clickevent()
7.   {
8.     document.write("This is JavaTpoint");
9.   }
10.  //-->
11. </script>
12. <form>
13. <input type="button" onclick="clickevent()" value="Who's this?"/>
14. </form>
15. </body>
16. </html>

```

### **MouseOver Event**

```

1. <html>
2. <head>
3. <h1> Javascript Events </h1>

```

```
4. </head>
5. <body>
6. <script language="Javascript" type="text/Javascript">
7.   <!--
8.   function mouseoverevent()
9.   {
10.     alert("This is JavaTpoint");
11.   }
12. //-->
13. </script>
14. <p onmouseover="mouseoverevent()"> Keep cursor over me</p>
15. </body>
16. </html>
```

## Focus Event

```
1. <html>
2. <head> Javascript Events</head>
3. <body>
4. <h2> Enter something here</h2>
5. <input type="text" id="input1" onfocus="focusevent()" />
6. <script>
7. <!--
8.   function focusevent()
9.   {
10.     document.getElementById("input1").style.background=" aqua";
11.   }
12. //-->
13. </script>
14. </body>
15. </html>
```

## Keydown Event

```
1. <html>
2. <head> Javascript Events</head>
3. <body>
```

```

4. <h2> Enter something here</h2>
5. <input type="text" id="input1" onkeydown="keydownevent()"/>
6. <script>
7. <!--
8.   function keydownevent()
9.   {
10.     document.getElementById("input1");
11.     alert("Pressed a key");
12.   }
13. //-->
14. </script>
15. </body>
16. </html>

```

## Load event

```

1. <html>
2. <head>Javascript Events</head>
3. <br>
4. <body onload="window.alert('Page successfully loaded');">
5. <script>
6. <!--
7.   document.write("The page is loaded successfully");
8. //-->
9. </script>
10. </body>
11. </html>

```

## Check if the value exists in Array in Javascript

In a programming language like Javascript, to check if the value exists in an array, there are certain methods. To be precise, there are plenty of ways to check if the value we are looking for resides amongst the elements in an array given by the user or is predefined. Let's discuss these methods one by one using various examples.

## indexof() method

The indexof() method in Javascript is one of the most convenient ways to find out whether a value exists in an array or not. The indexof() method works on the phenomenon of index numbers. This method returns the index of the array if found and returns -1 otherwise. Let's consider the below code:

```

1. <script>
2. var army=["Marcos", "DeltaForce", "Seals", "SWAT", "HeadHunters"];
3.
4. if(army.indexOf("Marcos") !== -1)
5. {
6.     alert("Yes, the value exists!")
7. }
8. else
9. {
10.    alert("No, the value is absent.")
11. }
12.
13. </script>
```

### Output

```
Yes, the value exists!
```

The above code prints the given out because the value is already present in the array. It is quite easy to understand that the expected value is present at position 0. Thus, the indexof() method tells you that the value expected is present in the given array.

## includes() method

The includes() method is one such method using which we can easily find out whether the expected value exists in the given array. There are various ways to use include() method. This method returns a Boolean value i.e. **true** if the value exists and **false** if it incorrect. The includes() method can be used in various ways to find out if the value exists. To name a few, take a look at the below examples to understand.

```

1. varspecialForces=["BlackCats","Marcos", "Demolishers","HeadHunters"];
2. r name = specialForces.includes("HeadHunters");
```

In the above method, we have defined two variables as shown. The includes() methods return **true** because the value which we are looking for is already present in the given array. If the value was not present in the array, the includes() methods might have returned false.

Another way of using the includes() method is by assigning the index value through which the element we are looking for is generated as output. See the below code for reference.

1. var **actors** = ["Hrithik", "SRK", "Salman", "Vidyut"];
2. var **names** = actors.includes("Vidyut", 3);

In the above code snippet, we have defined the variable "**actors**" which the value. We have also defined a variable "**names**" which would return true or false, if the includes() method returns the shown result. The code above will return true since the value and the index number have been correctly assigned and would return the output.

The above examples are some of the predefined methods that we have used to check whether an element exists in the array or not. We have another approach to find out an array element using loops. Let's discuss how can we check if the element exists in an array using loops as shown in the below code snippet.

## Using loops

1. var **example\_array** = ['Rahul','Rajesh','Sonu','Siddhi','Mark','George'];
- 2.
3. function checkArray(value,array)
4. {
5. var **status** = 'Absent';
- 6.
7. for(var **i**=0; **i**<**array.length**; **i**++)
8. {
9. var **name** = array[**i**];
10. if(**name** == **value**){
11. **status** = 'Present';
12. break;
13. }
14. }
15. return **status**;
16. }

## Output

```
status: Present
status: Absent
```

# How to disable radio button using JavaScript?

Radio button is an input type that is used to get input from the user by selecting one value from multiple choices. You have seen the radio buttons to choose gender between male and female. We select only one entry, either male or female and leave the other entries are unselected.

There might be some cases when we need to disable the other entries based on some conditions. You can enable and disable the radio button by using the **disabled** property of [HTML](#) DOM. Set this property to true (**disable=true**) to disable the radio button in [JavaScript](#).

## Disable the radio button

Sometimes, we need to disable the radio button for a specific condition. These are special conditions when we disable the radio button. When the radio buttons get disabled, their color changed to **grey**.

In the below examples, we will learn how to disable the radio button:

ADVERTISEMENT

## Disable radio button using dropdown

Here, we will use a dropdown list having **Yes** and **No** as values to enable or disable the radio buttons. If you choose No, all the radio buttons will be disabled. On the other hand, all the radio buttons will be enabled if you select Yes.

**Note that you can use the checkbox as well instead of a dropdown list.**

### Copy Code

1. <html>
2. <script>
3. function verifyAnswer() {
4. //get the selected value from the dropdown list
5. var myList = document.getElementById("myAns");
6. var result = myList.options[myList.selectedIndex].text;
7. if (result == 'No') {
8. //disable all the radio button

```

9. document.getElementById("csharp").disabled = true;
10. document.getElementById("js").disabled = true;
11. document.getElementById("angular").disabled = true;
12. } else {
13. //enable all the radio button
14. document.getElementById("csharp").disabled = false;
15. document.getElementById("js").disabled = false;
16. document.getElementById("angular").disabled = false;
17. }
18. }
19. </script>
20.
21. <body>
22. <h2> Disable radio Button using dropdown </h2>
23. <form>
24. <!-- create a dropdown list -->
25. <h3> Are you a developer? </h3>
26. <select id = "myAns" onchange = "verifyAnswer()" >
27. <option value="choose"> --choose -- </option>
28. <option value="yes"> Yes </option>
29. <option value="no"> No </option>
30. </select>
31. </form>
32.
33. <p> <b> If Yes, Choose language your preferred programming Language</b> </p>
34. <!-- create a set of radio buttons -->
35. <label> <input type="radio" name="programming" id="csharp" value= "csharp"> C# </label>
36. <label> <input type="radio" name="programming" id="js" value= "js"> JavaScript </label>
37. <label> <input type="radio" name="programming" id="angular" value= "angular"> Angular </label>
38. </body>
39. </html>

```

**Test it Now**

**Output 1**

You will get the output same as the given below by running the above code. Here, first choose either Yes or No from the dropdown list.

### Disable radio Button using dropdown

Are you a developer?

--choose -- ▾  
--choose --  
Yes      If Yes, Choose language your preferred programming Language  
No  
 C#  JavaScript  Angular

#### Output 2

By selecting **No** from the dropdown list, all radio buttons will get disabled, and you will not be allowed to choose any of the programming languages. See the screenshot below:

ADVERTISEMENT  
ADVERTISEMENT

### Disable radio Button using dropdown

Are you a developer?

No ▾

If Yes, Choose language your preferred programming Language

C#  JavaScript  Angular

#### Output 2

By selecting **Yes** from the dropdown list, all radio buttons will get enabled again. So, you will be able to choose one programming language. See the screenshot below:

## Disable radio Button using dropdown

**Are you a developer?**

Yes

**If Yes, Choose language your preferred programming Language**

- C#  JavaScript  Angular

## Disable radio button using checkbox

Now we will use a checkbox to disable the radio button.

**Copy Code**

```

1. <html>
2. <script>
3. function verifyAnswer() {
4.     //disable all the radio button
5.     document.getElementById("csharp").disabled = true;
6.     document.getElementById("js").disabled = true;
7.     document.getElementById("angular").disabled = true;
8.
9.     //get the value if checkbox is checked
10.    var dev = document.getElementById("myCheck").checked;
11.    if (dev == true) {
12.        //enable all the radio button
13.        document.getElementById("csharp").disabled = false;
14.        document.getElementById("js").disabled = false;
15.        document.getElementById("angular").disabled = false;
16.    }
17. }
18. </script>
19.

```

20. **<body>**
21. **<h2>** Disable radio Button using checkbox **</h2>**
22. **<form>**
23. **<!-- create a dropdown list -->**
24. **<h3>** Are you a developer? **</h3>**
25. Yes:
26. **<input type="checkbox" id="myCheck" onchange="verifyAnswer()" checked>**
27. **</form>**
- 28.
29. **<p> <b>** If Yes, Choose language your preferred programming Language**</b> </p>**
30. **<!-- create a set of radio buttons -->**
31. **<label> <input type="radio" name="programming" id="csharp" value= "csharp"> C# </label>**
32. **<label> <input type="radio" name="programming" id="js" value= "js"> JavaScript </label>**
33. **<label> <input type="radio" name="programming" id="angular" value= "angular"> Angular </label>**
34. **</body>**
35. **</html>**

**Test it Now**

## Output

You will get the output same as the given below by running the above code. Here, checkbox is already checked, you just need to choose one programming language by clicking on radio button.

**Disable radio Button using checkbox**

**Are you a developer?**

Yes:

**If Yes, Choose language your preferred programming Language**

C#  JavaScript  Angular

Uncheck the checkbox if you are not a developer. The radio buttons will get disabled by unchecking the checkbox.

ADVERTISEMENT

## Disable radio Button using checkbox

**Are you a developer?**

Yes:

**If Yes, Choose language your preferred programming Language**

C#  JavaScript  Angular

ADVERTISEMENT

So, these are the different methods to disable the radio button.

## A simple radio button example

It is a simple example of radio button created. In this example, we will create set of radio button for gender and language selection. The input will be taken using the HTML form and calculated by JavaScript. See the code below:

### Copy Code

```
1. <html>
2. <script>
3. function calValue() {
4. //fetch all gender radio button data
5. var male = document.getElementById('g1');
6. var female = document.getElementById('g2');
7. var otherg = document.getElementById('g3');
8.
9. //fetch all language radio button data
10. var hindi = document.getElementById('l1');
11. var english = document.getElementById('l2');
12. var otherl = document.getElementById('l3');
13.
14. var gender;
```

```
15. var language;
16. //check which gender is selected using radio button
17. if(male.checked == true) {
18.   gender = male;
19. } else if(female.checked == true) {
20.   gender = female;
21. } else if(otherg.checked == true) {
22.   gender = otherg
23. }
24.
25. //check which language is selected using radio button
26. if(hindi.checked == true) {
27.   language = hindi;
28. } else if(english.checked == true) {
29.   language = english;
30. } else if(otherl.checked == true) {
31.   language = otherl
32. }
33. //return data to HTML form
34. return document.getElementById("result").innerHTML = "Your selected gender is: " + gender.value + "<br> and
<br> Selected language is: " + language.value;
35. }
36. </script>
37.
38. <b><body>
39. <h2> Simple radio Buttons Example </h2>
40. <!-- create radio button for gender selection -->
41. <p> <b> Select your gender: </b> </p>
42. <input type="radio" id="g1" name="gender" value="male">
43. <label for="male"> Male </label> <br>
44. <input type="radio" id="g2" name="gender" value="female">
45. <label for="female"> Female </label> <br>
46. <input type="radio" id="g3" name="gender" value="other">
47. <label for="other"> Other </label>
```

```

48.
49. <br>
50. <!-- create radio button for language selection -->
51. <p> <b> Select your language: </b> </p>
52. <input type = "radio" id="l1" name="language" value="hindi">
53. <label for = "male"> Hindi </label> <br>
54. <input type="radio" id="l2" name="language" value="english">
55. <label for="female"> English </label> <br>
56. <input type="radio" id="l3" name="language" value="other">
57. <label for="other"> Other </label> <br> <br>
58.
59. <input type="submit" value="Submit" onclick="calValue()">
60. <h3 id="result" style="color:blue"> </h3>
61. </body>
62. </html>

```

**Test it Now**

## Output

See the below screenshot for radio button. Here, selects one value from each set of radio buttons. We have selected **gender = female** and **language = hindi**.

### Simple radio Buttons Example

Select your gender:

- Male
- Female
- Other

Select your language:

- Hindi
- English
- Other

**Show Selected Values**

After selecting the radio button values, click on the **Show Selected Values** button, and you will get the selected values have displayed on the web.

**Simple radio Buttons Example**

**Select your gender:**

Male  
 Female  
 Other

**Select your language:**

Hindi  
 English  
 Other

**Show Selected Values**

**Your selected gender is: male**  
**and**  
**Selected language is: english**

## How to add JavaScript to html

JavaScript, also known as JS, is one of the scripting (client-side scripting) languages, that is usually used in web development to create modern and interactive web-pages. The term "script" is used to refer to the languages that are not standalone in nature and here it refers to JavaScript which run on the client machine.

In other words, we can say that the term scripting is used for languages that require the support of another language to get executed. For example, JavaScript programs cannot get executed without the help of HTML or without integrated into HTML code.

JavaScript is used in several ways in web pages such as generate warning messages, build image galleries, DOM manipulation, form validation, and more.

## Adding JavaScript to HTML Pages

There are following three ways in which users can add JavaScript

1. Embedding code
2. Inline code
3. External file

We will see three of them step by step

## I. Embedding code:-

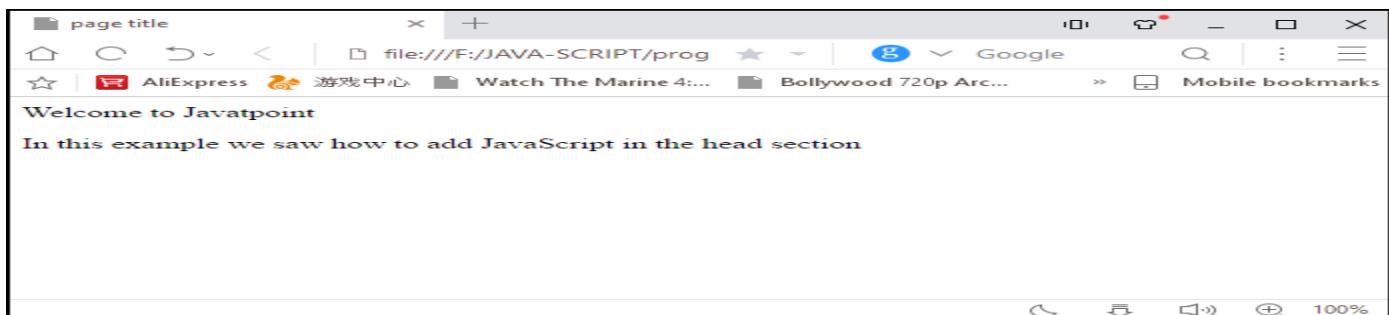
To add the JavaScript code into the HTML pages, we can use the `<script>.....</script>` tag of the HTML that wrap around JavaScript code inside the HTML program. Users can also define JavaScript code in the `<body>` tag (or we can say body section) or `<head>` tag because it completely depends on the structure of the web page that the users use.

We can understand this more clearly with the help of an example, how to add JavaScript to html

### Example

1. `<!DOCTYPE html >`
2. `<html>`
3. `<head>`
4. `<title> page title</title>`
5. `<script>`
6. `document.write("Welcome to Javatpoint");`
7. `</script>`
8. `</head>`
9. `<body>`
10. `<p>In this example we saw how to add JavaScript in the head section </p>`
11. `</body>`
12. `</html>`

### Output:



We can also define the JavaScript code in the `<body>` tags or body section.

Let's understand through an example.

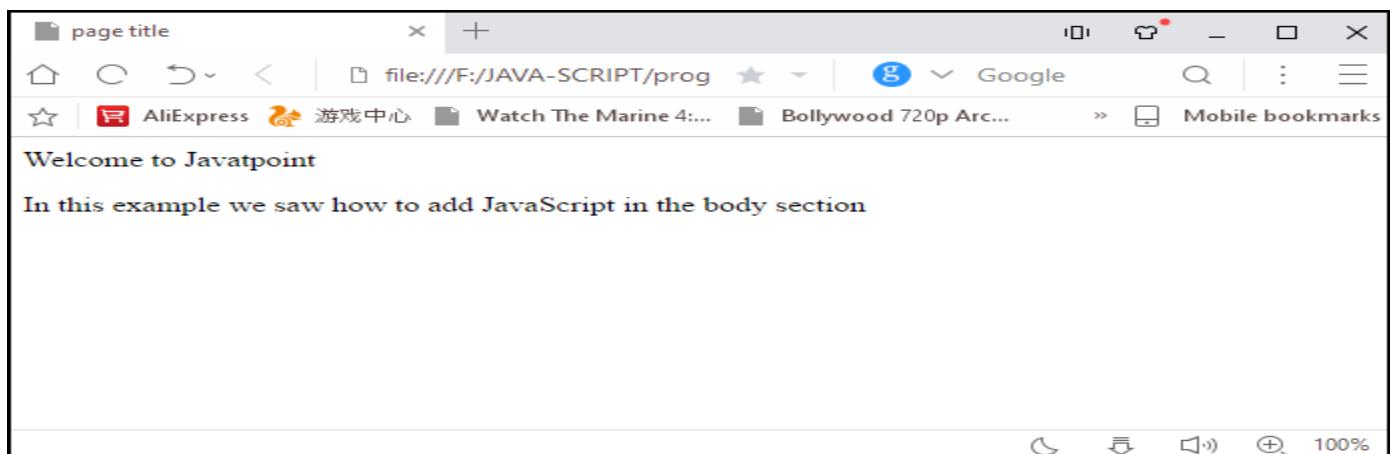
1. `<!DOCTYPE html >`

```

2. <html>
3. <head>
4. <title> page title</title>
5. </head>
6. <body>
7. <script>
8. document.write("Welcome to Javatpoint");
9. </script>
10. <p> In this example we saw how to add JavaScript in the body section </p>
11. </body>
12. </html>

```

## Output



## II. Inline code:-

Generally, this method is used when we have to call a function in the HTML event attributes. There are many cases (or events) in which we have to add JavaScript code directly eg., OnMover event, OnClick, etc.

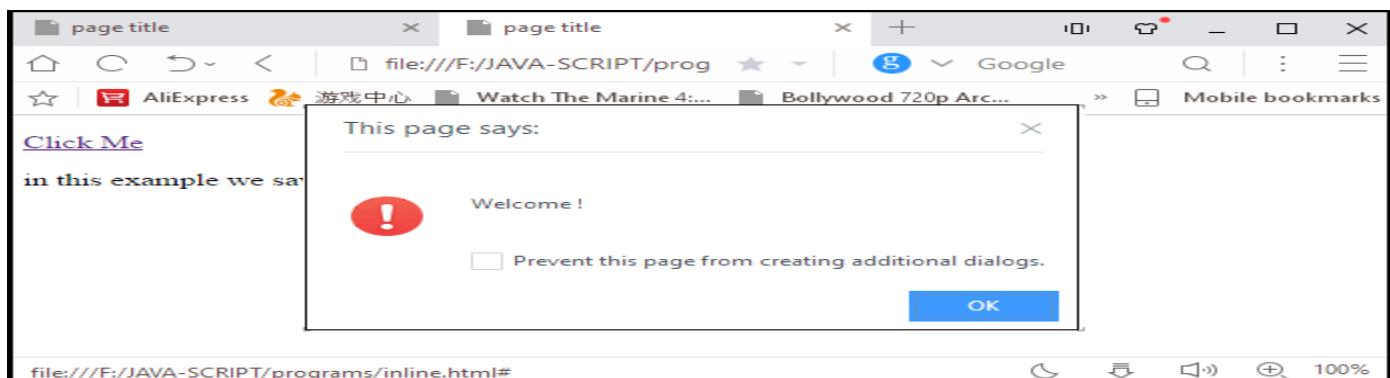
Let's see with the help of an example, how we can add JavaScript directly in the html without using the <script>.... </script> tag.

Let's look at the example.

1. <!DOCTYPE html >
2. <html>
3. <head>

4. <title> page title</title>
5. </head>
6. <body>
7. <p>
8. <a href="#" onClick="alert('Welcome !');">Click Me</a>
9. </p>
10. <p> in this example we saw how to use inline JavaScript or directly in an HTML tag. </p>
11. </body>
12. </html>

## Output



### III. External file:-

We can also create a separate file to hold the code of JavaScript with the (.js) extension and later incorporate/include it into our HTML document using the **src** attribute of the **<script>** tag. It becomes very helpful if we want to use the same code in multiple HTML documents. It also saves us from the task of writing the same code over and over again and makes it easier to maintain web pages.

In this example, we will see how we can include an external JavaScript file in an HTML document.

Let's understand through a simple example.

1. <html>
2. <head>
3. <meta charset="utf-8">
4. <title> Including a External JavaScript File</title>
5. </head>
6. <body>

```

7. <form>
8. <input type="button" value="Result" onclick="display()"/>
9. </form>
10. <script src="hello.js">
11. </script>
12. </body>
13. </html>

```

### Now let's create separate JavaScript file

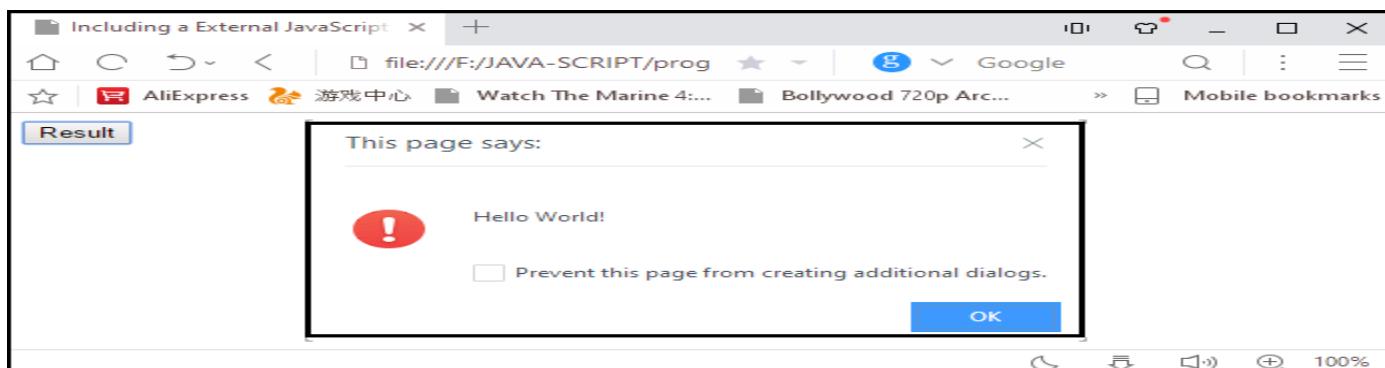
#### Hello.js

```

1. function display() {
2. alert("Hello World!");
3. }

```

#### Output



Both of the above programs are saved in the same folder, but you can also store JavaScript code in a separate folder, all just you need to provide the address/path of the (.js) file in the src attribute of <script> tag.

#### Important points

JavaScript files are common text files with (.js) extensions such as we created and used in the above program.

External JavaScript file only contains JavaScript code and nothing else, even the <script>.... </script> tag are also not used in it.

## The HTML no script Element

The <noscript> element provides us an alternate way to create content for the users that either have browsers that don't support the JavaScript or have disabled JavaScript in the browser.

This element can contain any HTML element other than the <script> tag that can be included in the <HTML> element.

Let's understand it more clearly with the help of an example:

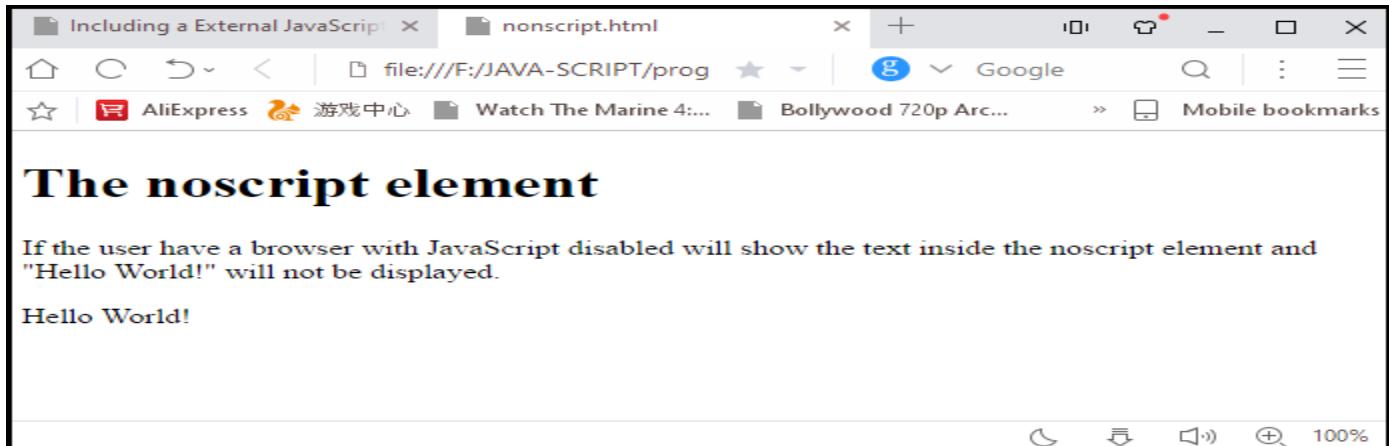
Program

1. <!DOCTYPE html>
2. <html>
3. <body>
4. <h1>Thenoscript element</h1>
5. <p>If the user have a browser with JavaScript disabled will show the text inside the noscript element and "Hello World!" will not be displayed.</p>
6. <script>
7. document.write("Hello World!")
8. </script>
9. <noscript>Sorry, your browser may not support JavaScript! orJavaScript is disabled in your browser </noscript>
- 10.
11. </body>
12. </html>

In the above program, we have used the javascript to print the message "**Hello World!**" and also used <nonscript> element to print the message " **Sorry, your browser may not support JavaScript! Or JavaScript is disabled in your browser**". The <nonscript> element will print this message only if JavaScript is disabled in the user's browser or his browser does not support JavaScript at all.

Let's see what output we get when we run the above given program in the JavaScript enabled browser.

**Output**



Now, let's see what output we will get when we run the same program in a JavaScript disabled browser (or in a browser that does not support JavaScript).

### Output



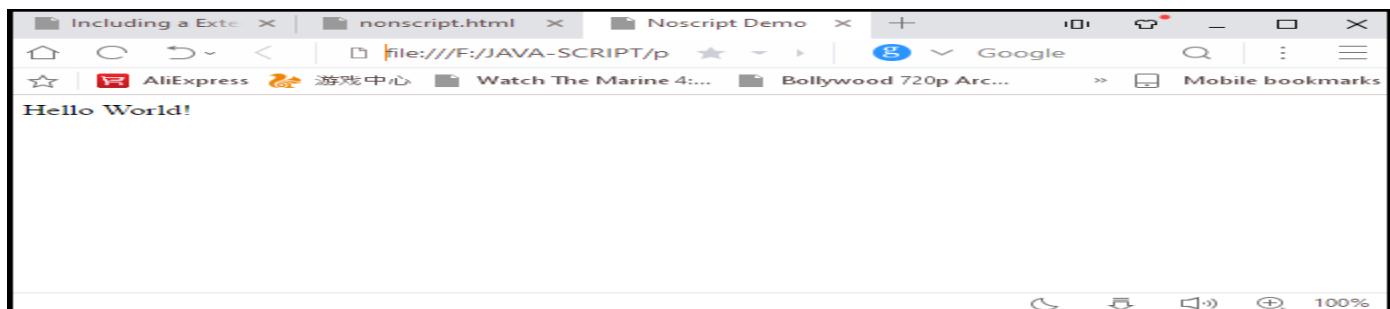
### Example 2

1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="utf-8">
5. <title>Noscript Demo</title>
6. </head>
7. <body>

8. `<div id="greet"></div>`
9. `<script>`
10. `document.getElementById("greet").innerHTML = "Hello World!";`
11. `</script>`
12. `<noscript>` //This element will print the given message only when the execution of the `<script>` tags does not take place.
13. `<p>Oops! This website requires a JavaScript-enabled browser.</p>`
14. `</noscript>`
15. `</body>`
16. `</html>`

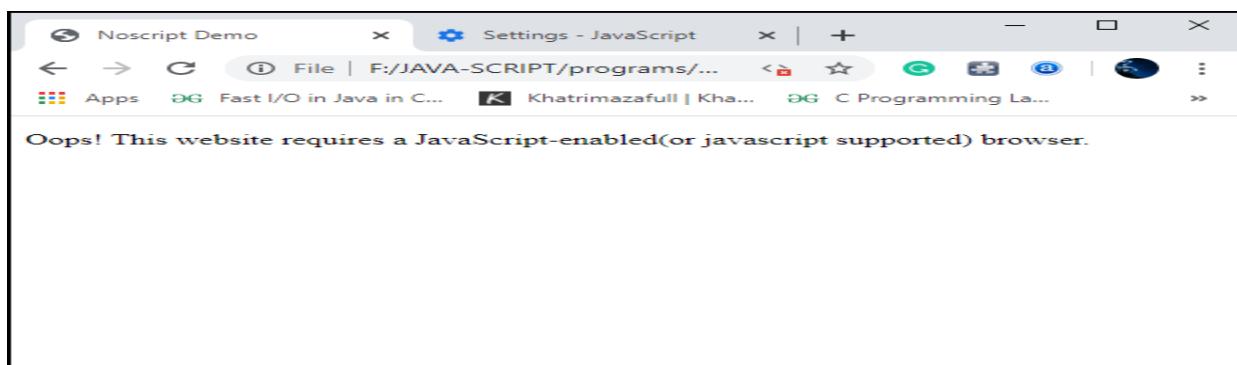
Let's run the above given program in the JavaScript enabled browser.

### Output



Now let's run the above program on JavaScript disabled browser

### Output



## Debugging javascript using firebug:

## Debugging JavaScript Using Developer Tools (Simple Explanation)

Modern web browsers like Google Chrome and Firefox have built-in tools to help you find and fix problems in your JavaScript code. Here's how to do it step-by-step, with an example.

Example Scenario:

Imagine you have a webpage where a button is supposed to display an alert saying "Hello, World!" when clicked, but it's not working.

### Step 1: Open Developer Tools

#### How to Open:

Press F12 on your keyboard, or

Right-click anywhere on the page and select "Inspect" or "Inspect Element."

### Step 2: Go to the Console Tab

#### What it Does:

The Console shows errors in your code and lets you run JavaScript commands directly.

#### Example:

If your button click isn't working, you might see an error message here.

### Step 3: Go to the Sources Tab (Chrome) or Debugger Tab (Firefox)

#### What it Does:

This is where you can see your JavaScript files and find the exact line of code that might be causing the problem.

#### How to Use:

Find your JavaScript file in the list on the left.

Click on it to see the code.

Click on a line number to add a **breakpoint** (this pauses the code here so you can see what's happening).

### Step 4: Use Breakpoints

#### What it Does:

Breakpoints let you pause the code and check what each part is doing.

#### Example:

If your code is supposed to show "Hello, World!" but doesn't, you can set a breakpoint where the alert should happen.

When you click the button, the code will pause, and you can check if it's reaching the alert line.

### **Step 5: Check Variables**

#### **What it Does:**

You can see the current value of variables in your code when it's paused.

#### **Example:**

If the alert isn't working, maybe the button isn't being detected properly. You can check the value of variables related to the button and see if they look correct.

### **Step 6: Fix the Problem**

#### **How to Fix:**

Based on what you see, you can figure out what's wrong and fix it in your code.

#### **Example:**

Maybe the button's ID in your HTML is different from what you have in your JavaScript. You can correct it and try again.

#### **Final Result**

Once you've used these steps to find and fix the problem, your button should work and show the "Hello, World!" alert when clicked.

This is a basic introduction to debugging JavaScript. As you get more comfortable, you can explore more features in the developer tools.

## **Introduction to various object models:**

### **Browser objects BOM:**

The **Browser Object Model (BOM)** in JavaScript refers to the objects provided by the browser that allow you to interact with the browser itself, outside of the web page content. Unlike the Document Object Model (DOM), which deals with the HTML and CSS of a webpage, the BOM provides methods and properties for interacting with the browser window.

Here are the key components of the BOM:

#### **Window Object :**

The `window` object is the main object in the BOM. It represents the browser window and acts as the global object in JavaScript, meaning all global variables and functions are properties of `window`.

#### **Example:**

```
console.log(window.innerWidth); // Gets the width of the window's content area
```

#### **Common properties and methods:**

window.innerWidth and window.innerHeight: Get the width and height of the browser window's content area.

window.open(): Opens a new browser window or tab.

window.close(): Closes the current window.

window.alert(): Displays an alert dialog.

window.setTimeout(): Calls a function after a specified delay.

window.location: Provides information about the current URL and allows navigation to different URLs.

## **2. location Object**

The `location` object contains information about the current URL and allows you to redirect the browser to a new URL.

#### **Example:**

```
console.log(window.location.href); // Prints the full URL of the current page
```

```
window.location.href = "https://www.example.com"; // Redirects to a new URL
```

#### **Common properties:**

`location.href`: Gets or sets the full URL.

`location.hostname`: Gets the domain name of the web host.

`location.pathname`: Gets the path of the URL.

`location.search`: Gets the query string part of the URL (e.g., ?id=123).

`location.reload()`: Reloads the current page.

## **3. navigator Object**

The `navigator` object provides information about the browser and the user's system.

#### **Example:**

```
console.log(navigator.userAgent); // Prints the user agent string (browser info)
```

#### **Common properties:**

`navigator.userAgent`: Contains the user agent string, which includes information about the browser version, operating system, etc.

`navigator.language`: Returns the language of the browser.

`navigator.geolocation`: Provides access to the user's location (with permission).

## **4. history Object**

The `history` object allows you to interact with the browser's history (the list of pages the user has visited in the current session).

**Example:**

```
window.history.back(); // Goes back to the previous page in history
```

**Common methods:**

`history.back()`: Goes back one page in the session history.

`history.forward()`: Goes forward one page in the session history.

`history.go(n)`: Moves forward or backward by `n` pages in the session history.

## 5. screen Object

The `screen` object provides information about the user's screen, such as its resolution.

**Example:**

```
console.log(screen.width); // Prints the width of the user's screen
```

**Common properties:**

`screen.width`: The width of the screen in pixels.

`screen.height`: The height of the screen in pixels.

`screen.availWidth`: The width of the screen excluding the operating system taskbar.

`screen.availHeight`: The height of the screen excluding the operating system taskbar.

**Summary**

The BOM gives you tools to interact with the browser environment, such as controlling the browser window, navigating between pages, and getting information about the user's browser and screen. These features help make web pages more dynamic and interactive by integrating with the browser itself.

## Document Object Model

1. [Document Object](#)
2. [Properties of document object](#)
3. [Methods of document object](#)
4. [Example of document object](#)

The `document object` represents the whole html document.

When html document is loaded in the browser, it becomes a document object. It is the **root element** that represents the html document. It has properties and methods. By the help of document object, we can add dynamic content to our web page.

As mentioned earlier, it is the object of window. So

### **window.document**

Is same as

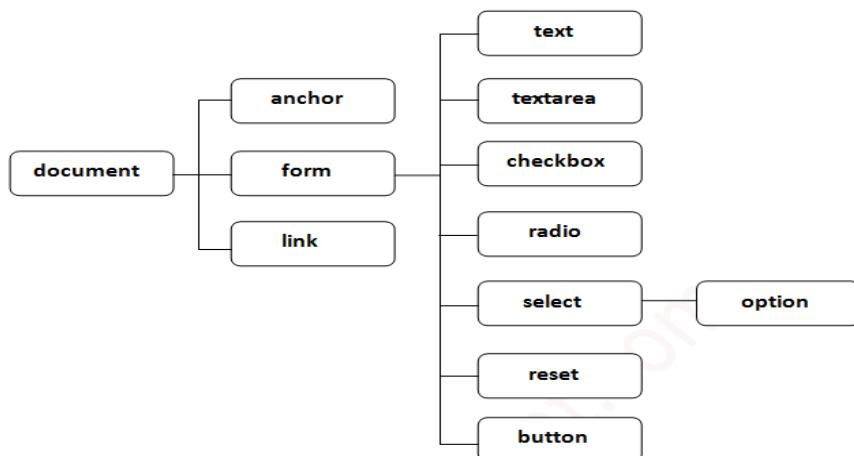
### **document**

According to W3C - "*The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document.*"

---

### **Properties of document object**

Let's see the properties of document object that can be accessed and modified by the document



object.

---

### **Methods of document object**

We can access and change the contents of document by its methods.

The important methods of document object are as follows:

| Method          | Description                              |
|-----------------|--|
| write("string") | writes the given string on the document. |

|                          |  |
|--------------------------|--|
| writeln("string")        | writes the given string on the document with newline character at the end. |
| getElementById()         | returns the element having the given id value.                             |
| getElementsByName()      | returns all the elements having the given name value.                      |
| getElementsByTagName()   | returns all the elements having the given tag name.                        |
| getElementsByClassName() | returns all the elements having the given class name.                      |

### Accessing field value by document object

In this example, we are going to get the value of input text by user. Here, we are using **document.form1.name.value** to get the value of name field.

Here, **document** is the root element that represents the html document.

**form1** is the name of the form.

**name** is the attribute name of the input text.

**value** is the property, that returns the value of the input text.

Let's see the simple example of document object that prints name with welcome message.

1. **<script type="text/javascript">**
2. **function printvalue(){**
3. **var name=document.form1.name.value;**
4. **alert("Welcome: "+name);**
5. **}**
6. **</script>**
- 7.
8. **<form name="form1">**
9. **Enter Name:<input type="text" name="name"/>**
10. **<input type="button" onclick="printvalue()" value="print name"/>**
11. **</form>**

### Output of the above example

Enter Name:

## Javascript - document.getElementById() method

1. [getElementById\(\) method](#)
2. [Example of getElementById\(\)](#)

The **document.getElementById()** method returns the element of specified id.

In the previous page, we have used **document.form1.name.value** to get the value of the input value. Instead of this, we can use **document.getElementById()** method to get value of the input text. But we need to define id for the input field.

Let's see the simple example of **document.getElementById()** method that prints cube of the given number.

1. **<script type="text/javascript">**
2. **function getcube(){**
3. **var number=document.getElementById("number").value;**
4. **alert(number\*number\*number);**
5. **}**
6. **</script>**
7. **<form>**
8. **Enter No:<input type="text" id="number" name="number"/><br/>**
9. **<input type="button" value="cube" onclick="getcube()" />**
10. **</form>**

### Output of the above example

EnterNo:

## Javascript - document.getElementsByName() method

1. [getElementsByName\(\) method](#)
2. [Example of getElementsByName\(\)](#)

The **document.getElementsByName()** method returns all the element of specified name.

The syntax of the `getElementsByName()` method is given below:

1. `document.getElementsByName("name")`

Here, name is required.

#### **Example of `document.getElementsByName()` method**

In this example, we going to count total number of genders. Here, we are using `getElementsByName()` method to get all the genders.

```

1. <script type="text/javascript">
2. function totalelements()
3. {
4. var allgenders=document.getElementsByName("gender");
5. alert("Total Genders:"+allgenders.length);
6. }
7. </script>
8. <form>
9. Male:<input type="radio" name="gender" value="male">
10. Female:<input type="radio" name="gender" value="female">
11.
12. <input type="button" onclick="totalelements()" value="Total Genders">
13. </form>
```

#### **Output of the above example**

Male:  Female:

### **Javascript - `document.getElementsByTagName()` method**

1. [`getElementsByName\(\)` method](#)
2. [Example of `getElementsByName\(\)`](#)

The `document.getElementsByTagName()` method returns all the element of specified tag name.

The syntax of the `getElementsByName()` method is given below:

1. `document.getElementsByTagName("name")`

Here, name is required.

#### Example of document.getElementsByTagName() method

In this example, we going to count total number of paragraphs used in the document. To do this, we have called the document.getElementsByTagName("p") method that returns the total paragraphs.

```

1. <script type="text/javascript">
2. function countpara(){
3. var totalpara=document.getElementsByTagName("p");
4. alert("total p tags are: "+totalpara.length);
5.
6. }
7. </script>
8. <p>This is a paragraph</p>
9. <p>Here we are going to count total number of paragraphs by getElementByTagName() method.</p>
10. <p>Let's see the simple example</p>
11. <button onclick="countpara()">count paragraph</button>
```

#### Output of the above example

This is a paragraph

Here we are going to count total number of paragraphs by getElementByTagName() method.

Let's see the simple example

count paragraph

#### Another example of document.getElementsByTagName() method

In this example, we going to count total number of h2 and h3 tags used in the document.

```

1. <script type="text/javascript">
2. function counth2(){
3. var totalh2=document.getElementsByTagName("h2");
4. alert("total h2 tags are: "+totalh2.length);
5. }
6. function counth3(){
```

```

7. var totalh3=document.getElementsByTagName("h3");
8. alert("total h3 tags are: "+totalh3.length);
9. }
10.</script>
11.<h2>This is h2 tag</h2>
12.<h2>This is h2 tag</h2>
13.<h3>This is h3 tag</h3>
14.<h3>This is h3 tag</h3>
15.<h3>This is h3 tag</h3>
16.<button onclick="counth2()">count h2</button>
17.<button onclick="counth3()">count h3</button>

```

---

### Output of the above example

This is h2 tag

This is h2 tag

This is h3 tag

This is h3 tag

This is h3 tag

count h2 count h3

## Javascript - innerHTML

1. [javascript innerHTML](#)
2. [Example of innerHTML property](#)

The **innerHTML** property can be used to write the dynamic html on the html document.

It is used mostly in the web pages to generate the dynamic html such as registration form, comment form, links etc.

### Example of innerHTML property

In this example, we are going to create the html form when user clicks on the button.

In this example, we are dynamically writing the html form inside the div name having the id mylocation. We are identifying this position by calling the document.getElementById() method.

```

1. <script type="text/javascript" >
2. function showcommentform() {
3. var data="Name:<input type='text' name='name'><br>Comment:<br><textarea rows='5' cols='80'></textarea>
4. <br><input type='submit' value='Post Comment'>";
5. document.getElementById('mylocation').innerHTML=data;
6. }
7. </script>
8. <form name="myForm">
9. <input type="button" value="comment" onclick="showcommentform()">
10. <div id="mylocation"></div>
11. </form>
```

### Show/Hide Comment Form Example using innerHTML

```

1. <!DOCTYPE html>
2. <html>
3. <head>
4. <title>First JS</title>
5. <script>
6. var flag=true;
7. function commentform(){
8. var cform=<form action='Comment'>Enter Name:<br><input type='text' name='name' /><br/>
9. Enter Email:<br><input type='email' name='email' /><br>Enter Comment:<br/>
10. <textarea rows='5' cols='70'></textarea><br><input type='submit' value='Post Comment' /></form>;
11. if(flag){
12. document.getElementById("mylocation").innerHTML=cform;
13. flag=false;
14. }else{
15. document.getElementById("mylocation").innerHTML="";
16. flag=true;
17. }
18. }
```

```

19. </script>
20. </head>
21. <body>
22. <button onclick="commentform()">Comment</button>
23. <div id="mylocation"></div>
24. </body>
25. </html>

```

## Javascript - innerText

1. [javascript innerText](#)
2. [Example of innerText property](#)

The **innerText** property can be used to write the dynamic text on the html document. Here, text will not be interpreted as html text but a normal text.

It is used mostly in the web pages to generate the dynamic content such as writing the validation message, password strength etc.

### Javascript innerText Example

In this example, we are going to display the password strength when releases the key after press.

```

1. <script type="text/javascript" >
2. function validate() {
3. var msg;
4. if(document.myForm.userPass.value.length>5){
5. msg="good";
6. }
7. else{
8. msg="poor";
9. }
10. document.getElementById('mylocation').innerText=msg;
11. }
12.
13. </script>
14. <form name="myForm">

```

15. `<input type="password" value="" name="userPass" onkeyup="validate()">`

16. Strength:`<span id="mylocation">no strength</span>`

17. `</form>`

**Test it Now**

### Output of the above example

Strength: no strength

## Introduction to jQuery

**jQuery** is a fast, small, and feature-rich JavaScript library. It was designed to simplify the client-side scripting of HTML. jQuery makes things like HTML document traversal and manipulation, event handling, animation, and AJAX much simpler with an easy-to-use API that works across a multitude of browsers.

### Why Use jQuery?

1. **Simplifies JavaScript:** jQuery provides a simplified syntax that makes it easier to write JavaScript code. It takes common JavaScript tasks and reduces the amount of code needed to accomplish them.
2. **Cross-Browser Compatibility:** jQuery handles the differences between browsers, allowing you to write code that works consistently across all major browsers.
3. **Rich Set of Features:** jQuery comes with a large number of built-in functions for tasks like DOM manipulation, event handling, animations, and AJAX.
4. **Extensibility:** jQuery can be extended with plugins, which allow developers to add custom methods and functions.
5. **Large Community and Documentation:** jQuery has a large community of users, which means plenty of resources, tutorials, and plugins are available.

### How to Include jQuery

You can include jQuery in your web project by using a `<script>` tag in your HTML file. There are two main ways to do this:

1. **Using a CDN (Content Delivery Network):**
  - Example:
  - `<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>`
  - This loads jQuery from a remote server, reducing the load on your own server.
2. **Downloading jQuery:**
  - You can download jQuery from jQuery's official website and include it in your project.
  - Example:

```
<script src="path/to/jquery.min.js"></script>
```

## Basic jQuery Syntax

The basic syntax of jQuery is:

```
$(selector).action();
```

- `$`: This is the jQuery object or function.
- `selector`: A string that specifies the HTML element(s) you want to select.
- `action()`: The jQuery action or method you want to perform on the selected elements.

## Example: Hello World with jQuery

Here's a simple example that changes the text of an HTML element when a button is clicked:

```
<!DOCTYPE html>
<html>
<head>
    <title>jQuery Example</title>
    <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
    <script>
        $(document).ready(function() {
            $("#myButton").click(function() {
                $("#myText").text("Hello, World!");
            });
        });
    </script>
</head>
<body>
    <h1 id="myText">Welcome!</h1>
    <button id="myButton">Change Text</button>
</body>
</html>
```

### Explanation:

- `$(document).ready(function() { ... })` ;: This ensures that the code inside the function runs only after the DOM is fully loaded.
- `$("#myButton").click(function() { ... })` ;: This selects the button with `id="myButton"` and sets up a click event handler.
- `$("#myText").text("Hello, World!")` ;: When the button is clicked, the text of the element with `id="myText"` is changed to "Hello, World!".

## Common jQuery Methods

- **DOM Manipulation:**
  - `text()`: Gets or sets the text content of elements.
  - `html()`: Gets or sets the HTML content of elements.
  - `val()`: Gets or sets the value of form fields.
  - `addClass()`: Adds one or more classes to elements.
  - `removeClass()`: Removes one or more classes from elements.
- **Event Handling:**

- click(): Attaches a click event to elements.
- hover(): Attaches a hover event (mouse over and mouse out) to elements.
- on(): Attaches event handlers for multiple events.
- **Effects and Animations:**
  - show(): Displays hidden elements.
  - hide(): Hides elements.
  - fadeIn(): Fades in elements.
  - fadeOut(): Fades out elements.
  - slideUp(): Slides up elements.
  - slideDown(): Slides down elements.
- **AJAX:**
  - \$.ajax(): Performs asynchronous HTTP requests.
  - \$.get(): Performs an HTTP GET request.
  - \$.post(): Performs an HTTP POST request.

## Summary

jQuery is a powerful tool for simplifying JavaScript development, providing an easy-to-use API for common tasks like DOM manipulation, event handling, and AJAX. It's widely used and supported, making it a great choice for enhancing web development projects.

## jQuery Example

jQuery is developed by Google. To create the first jQuery example, you need to use JavaScript file for jQuery. You can download the jQuery file from [jquery.com](http://jquery.com) or use the absolute URL of jQuery file.

In this jQuery example, we are using the absolute URL of jQuery file. The jQuery example is written inside the script tag.

Let's see a simple example of jQuery.

*File: firstjquery.html*

1. <!DOCTYPE html>
2. <html>
3. <head>
4. <title>First jQuery Example</title>
5. <script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js">
6. </script>
7. <script type="text/javascript" language="javascript">
8. \$(document).ready(function() {
9. \$("p").css("background-color", "cyan");

```
10. });
11. </script>
12. </head>
13. <body>
14. <p>The first paragraph is selected.</p>
15. <p>The second paragraph is selected.</p>
16. <p>The third paragraph is selected.</p>
17. </body>
18. </html>
```

Output:

The first paragraph is selected.

The second paragraph is selected.

The third paragraph is selected.

`$(document).ready() and $()`

The code inserted between `$(document).ready()` is executed only once when page is ready for JavaScript code to execute.

In place of `$(document).ready()`, you can use shorthand notation `$()` only.

```
1. $(document).ready(function() {
2.   $("p").css("color", "red");
3. });
```

The above code is equivalent to this code.

```
1. $(function() {
2.   $("p").css("color", "red");
3. });
```

Let's see the full example of jQuery using shorthand notation `$()`.

*File: shortjquery.html*

```
1. <!DOCTYPE html>
2. <html>
3. <head>
4. <title>Second jQuery Example</title>
5. <script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js">
6. </script>
7. <script type="text/javascript" language="javascript">
8. $(function() {
9. $("p").css("color", "red");
10. });
11. </script>
12. </head>
13. <body>
14. <p>The first paragraph is selected.</p>
15. <p>The second paragraph is selected.</p>
16. <p>The third paragraph is selected.</p>
17. </body>
18. </html>
```

Output:

The first paragraph is selected.

The second paragraph is selected.

The third paragraph is selected.

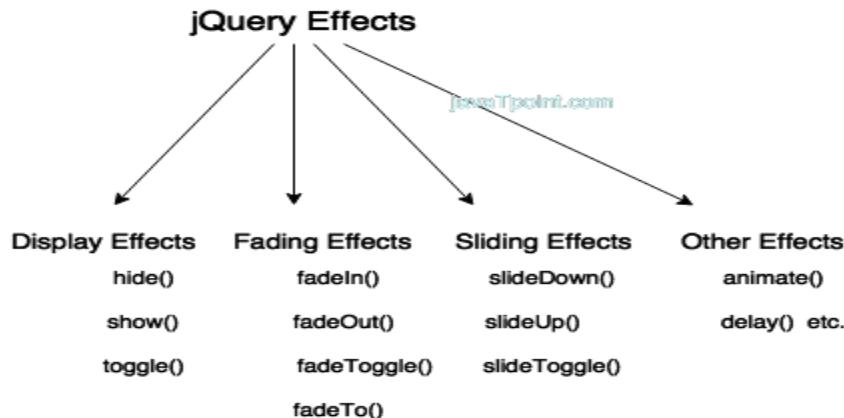
---

```
function() { $("p").css("background-color", "cyan"); }
```

It changes the background-color of all <p> tag or paragraph to cyan.

## jQuery Effects:

jQuery enables us to add effects on a web page. jQuery effects can be categorized into fading, sliding, hiding/showing and animation effects.



### jQuery hide()

The jQuery `hide()` method is used to hide the selected elements.

#### Syntax:

1. `$(selector).hide();`
2. `$(selector).hide(speed, callback);`
3. `$(selector).hide(speed, easing, callback);`

**speed:** It is an optional parameter. It specifies the speed of the delay. Its possible values are slow, fast and milliseconds.

**easing:** It specifies the easing function to be used for transition.

**callback:** It is also an optional parameter. It specifies the function to be called after completion of `hide()` effect.

Let's take an example to see the jQuery hide effect.

1. `<!DOCTYPE html>`
2. `<html>`
3. `<head>`
4. `<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js"></script>`
5. `<script>`
6. `$(document).ready(function(){`
7.  `$("#hide").click(function(){`

```

8.      $("p").hide();
9.    });
10. });
11. </script>
12. </head>
13. <body>
14. <p>
15. <b>This is a little poem: </b><br/>
16. Twinkle, twinkle, little star<br/>
17. How I wonder what you are<br/>
18. Up above the world so high<br/>
19. Like a diamond in the sky<br/>
20. Twinkle, twinkle little star<br/>
21. How I wonder what you are
22. </p>
23. <button id="hide">Hide</button>
24. </body>
25. </html>

```

### Test it Now

Output:

This	is	a	little	poem:
Twinkle,		twinkle,	little	star
How	I	wonder	what	are
Up	above	the	world	high
Like	a	diamond	in	sky
Twinkle,		twinkle		star
How I wonder what you are				

Hide

## jQuery show()

The jQuery show() method is used to show the selected elements.

**Syntax:**

1. `$(selector).show();`

2. \$(selector).show(speed, callback);
3. \$(selector).show(speed, easing, callback);

**speed:** It is an optional parameter. It specifies the speed of the delay. Its possible values are slow, fast and milliseconds.

**easing:** It specifies the easing function to be used for transition.

**callback:** It is also an optional parameter. It specifies the function to be called after completion of show() effect.

Let's take an example to see the jQuery show effect.

1. <!DOCTYPE html>
2. <html>
3. <head>
4. <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js"></script>
5. <script>
6. \$(document).ready(function(){
7.     \$("#hide").click(function(){
8.         \$("p").hide();
9.     });
10.    \$("#show").click(function(){
11.         \$("p").show();
12.     });
13. });
14. </script>
15. </head>
16. <body>
17. <p>
18. <b>This is a little poem: </b><br/>
19. Twinkle, twinkle, little star<br/>
20. How I wonder what you are<br/>
21. Up above the world so high<br/>
22. Like a diamond in the sky<br/>
23. Twinkle, twinkle little star<br/>
24. How I wonder what you are
25. </p>

26. <button id="hide">Hide</button>
27. <button id="show">Show</button>
28. </body>
29. </html>

Output:

This	is	a	little	poem:
Twinkle,		twinkle,	little	star
How	I	wonder	what	are
Up	above	the	world	high
Like	a	diamond	in	sky
Twinkle,		twinkle	little	star
How I wonder what you are				

Hide Show

### jQuery show() effect with speed parameter

Let's see the example of jQuery show effect with 1500 milliseconds speed.

1. \$(document).ready(function(){
2.     \$("#hide").click(function(){
3.         \$("p").hide(1000);
4.     });
5.     \$("#show").click(function(){
6.         \$("p").show(1500);
7.     });
8. });

### jQuery toggle()

The jQuery toggle() is a special type of method which is used to toggle between the hide() and show() method. It shows the hidden elements and hides the shown element.

**Syntax:**

1. \$(selector).toggle();
2. \$(selector).toggle(speed, callback);
3. \$(selector).toggle(speed, easing, callback);
4. \$(selector).toggle(display);

**speed:** It is an optional parameter. It specifies the speed of the delay. Its possible values are slow, fast and milliseconds.

**easing:** It specifies the easing function to be used for transition.

**callback:** It is also an optional parameter. It specifies the function to be called after completion of toggle() effect.

**display:** If true, it displays element. If false, it hides the element.

Let's take an example to see the jQuery toggle effect.

```

1. <!DOCTYPE html>
2. <html>
3. <head>
4. <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js"></script>
5. <script>
6. $(document).ready(function(){
7.     $("button").click(function(){
8.         $("div.d1").toggle();
9.     });
10. });
11. </script>
12. </head>
13. <body>
14. <button>Toggle</button>
15. <div class="d1" style="border:1px solid black;padding:10px;width:250px">
16. <p><b>This is a little poem:</b><br/>
17. Twinkle, twinkle, little star<br/>
18. How I wonder what you are<br/>
19. Up above the world so high<br/>
20. Like a diamond in the sky<br/>
21. Twinkle, twinkle little star<br/>
22. How I wonder what you are</p>
23. </div>
24. </body>
25. </html>
```

Output:

## Toggle

This	is	a	little	poem:
Twinkle,		twinkle,	little	star
How	I	wonder	what	are
Up	above	the	world	high
Like	a	diamond	in	sky
Twinkle,		twinkle	little	star
How I wonder what you are				

### jQuery toggle() effect with speed parameter

Let's see the example of jQuery toggle effect with 1500 milliseconds speed.

1. \$(document).ready(function(){
2.     \$("button").click(function(){
3.         \$("div.d1").toggle(1500);
4.     });
5. });

## jQuery fadeIn()

jQuery fadeIn() method is used to fade in the element.

### Syntax:

1. \$(selector).fadeIn();
2. \$(selector).fadeIn(speed,callback);
3. \$(selector).fadeIn(speed, easing, callback);

**speed:** It is an optional parameter. It specifies the speed of the delay. Its possible values are slow, fast and milliseconds.

**easing:** It specifies the easing function to be used for transition.

**callback:** It is also an optional parameter. It specifies the function to be called after completion of fadeIn() effect.

Let's take an example to demonstrate jQuery fadeIn() effect.

1. <!DOCTYPE html>
2. <html>

```

3. <head>
4. <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js"></script>
5. <script>
6. $(document).ready(function(){
7.     $("button").click(function(){
8.         $("#div1").fadeIn();
9.         $("#div2").fadeIn("slow");
10.        $("#div3").fadeIn(3000);
11.    });
12. });
13. </script>
14. </head>
15. <body>
16. <p>See the fadeIn() method example with different parameters.</p>
17. <button>Click to fade in boxes</button><br><br>
18. <div id="div1" style="width:80px;height:80px;display:none;background-color:red;"></div><br>
19. <div id="div2" style="width:80px;height:80px;display:none;background-color:green;"></div><br>
20. <div id="div3" style="width:80px;height:80px;display:none;background-color:blue;"></div>
21. </body>
22. </html>

```

## jQuery fadeOut()

The jQuery fadeOut() method is used to fade out the element.

**Syntax:**

1. \$(selector).fadeOut();
2. \$(selector).fadeOut(speed,callback);
3. \$(selector).fadeOut(speed, easing, callback);

**speed:** It is an optional parameter. It specifies the speed of the delay. Its possible values are slow, fast and milliseconds.

**easing:** It specifies the easing function to be used for transition.

**callback:** It is also an optional parameter. It specifies the function to be called after completion of fadeOut() effect.

Let's take an example to demonstrate jQuery fadeOut() effect.

```

1. <!DOCTYPE html>
2. <html>
3. <head>
4. <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js"></script>
5. <script>
6. $(document).ready(function(){
7.   $("button").click(function(){
8.     $("#div1").fadeOut();
9.     $("#div2").fadeOut("slow");
10.    $("#div3").fadeOut(3000);
11.  });
12. });
13. </script>
14. </head>
15. <body>
16. <p>See the fadeOut() method example with different parameters.</p>
17. <button>Click to fade out boxes</button><br><br>
18. <div id="div1" style="width:80px;height:80px;background-color:red;"></div><br>
19. <div id="div2" style="width:80px;height:80px;background-color:green;"></div><br>
20. <div id="div3" style="width:80px;height:80px;background-color:blue;"></div>
21. </body>
22. </html>

```

## jQuery fadeToggle()

jQuery fadeToggle() method is used to toggle between the fadeIn() and fadeOut() methods. If the elements are faded in, it will make them faded out and if they are faded out it will make them faded in.

**Syntax:**

1. \$(selector).fadeToggle();
2. \$(selector).fadeToggle(speed,callback);
3. \$(selector).fadeToggle(speed, easing, callback);

**speed:** It is an optional parameter. It specifies the speed of the delay. Its possible values are slow, fast and milliseconds.

**easing:** It specifies the easing function to be used for transition.

**callback:** It is also an optional parameter. It specifies the function to be called after completion of fadeToggle() effect.

Let's take an example to demonstrate jQuery fadeToggle() effect.

```

1. <!DOCTYPE html>
2. <html>
3. <head>
4. <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js"></script>
5. <script>
6. $(document).ready(function(){
7.   $("button").click(function(){
8.     $("#div1").fadeToggle();
9.     $("#div2").fadeToggle("slow");
10.    $("#div3").fadeToggle(3000);
11.  });
12. });
13. </script>
14. </head>
15. <body>
16. <p>See the fadeToggle() method example with different parameters.</p>
17. <button>Click to fade Toggle boxes</button><br><br>
18. <div id="div1" style="width:80px;height:80px;background-color:red;"></div><br>
19. <div id="div2" style="width:80px;height:80px;background-color:green;"></div><br>
20. <div id="div3" style="width:80px;height:80px;background-color:blue;"></div>
21. </body>
22. </html>
```

## jQuery fadeTo()

jQuery fadeTo() method is used to fading to a given opacity.

**Syntax:**

1. \$(selector).fadeTo(speed, opacity);
2. \$(selector).fadeTo(speed, opacity, callback);
3. \$(selector).fadeTo(speed, opacity, easing, callback);

**speed:** It specifies the speed of the delay. Its possible values are slow, fast and milliseconds.

**opacity:** It specifies the opacity. The opacity value ranges between 0 and 1.

#### ADVERTISEMENT

**easing:** It specifies the easing function to be used for transition.

**callback:** It is also an optional parameter. It specifies the function to be called after completion of fadeToggle() effect.

Let's take an example to demonstrate jQuery fadeTo() effect.

```

1. <!DOCTYPE html>
2. <html>
3. <head>
4. <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js"></script>
5. <script>
6. $(document).ready(function(){
7.   $("button").click(function(){
8.     $("#div1").fadeTo("slow", 0.3);
9.     $("#div2").fadeTo("slow", 0.4);
10.    $("#div3").fadeTo("slow", 0.5);
11.  });
12. });
13. </script>
14. </head>
15. <body>
16. <p>See the fadeTo() method example with different parameters.</p>
17. <button>Click to fade boxes</button><br><br>
18. <div id="div1" style="width:80px;height:80px;background-color:red;"></div><br>
19. <div id="div2" style="width:80px;height:80px;background-color:green;"></div><br>
20. <div id="div3" style="width:80px;height:80px;background-color:blue;"></div>
21. </body>
22. </html>
```

## jQuery slideDown()

jQuery slideDown() method is used to slide down an element.

**Syntax:**

1. \$(selector).slideDown(speed);
2. \$(selector).slideDown(speed, callback);
3. \$(selector).slideDown(speed, easing, callback);

**speed:** It specifies the speed of the delay. Its possible values are slow, fast and milliseconds.

**easing:** It specifies the easing function to be used for transition.

**callback:** It is also an optional parameter. It specifies the function to be called after completion of slideDown() effect.

Let's take an example to demonstrate jQuery slideDown() effect.

```

1. <!DOCTYPE html>
2. <html>
3. <head>
4. <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js"></script>
5. <script>
6. $(document).ready(function(){
7.   $("#flip").click(function(){
8.     $("#panel").slideDown("slow");
9.   });
10. });
11. </script>
12. <style>
13. #panel, #flip {
14.   padding: 5px;
15.   text-align: center;
16.   background-color: #00FFFF;
17.   border: solid 1px #c3c3c3;
18. }
19. #panel {
20.   padding: 50px;
21.   display: none;
22. }
```

23. </style>
24. </head>
25. <body>
26. <div id="flip">Click to slide down panel</div>
27. <div id="panel">Hello javatpoint.com!
28. It is the best tutorial website to learn jQuery and other languages.</div>
29. </body>
30. </html>

## jQuery slideUp()

jQuery slideDown() method is used to slide up an element.

**Syntax:**

1. \$(selector).slideUp(speed);
2. \$(selector).slideUp(speed, callback);
3. \$(selector).slideUp(speed, easing, callback);

**speed:** It specifies the speed of the delay. Its possible values are slow, fast and milliseconds.

**easing:** It specifies the easing function to be used for transition.

**callback:** It is also an optional parameter. It specifies the function to be called after completion of slideUp() effect.

Let's take an example to demonstrate jQuery slideUp() effect.

1. <!DOCTYPE html>
2. <html>
3. <head>
4. <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js"></script>
5. <script>
6. \$(document).ready(function(){
7. \$("#flip").click(function(){
8. \$("#panel").slideUp("slow");
9. });
- 10.});
11. </script>
12. <style>

```

13. #panel, #flip {
14.   padding: 5px;
15.   text-align: center;
16.   background-color: #00FFFF;
17.   border: solid 1px #c3c3c3;
18. }
19. #panel {
20.   padding: 50px;
21. }
22. </style>
23. </head>
24. <body>
25. <div id="flip">Click to slide up panel</div>
26. <div id="panel">Hello javatpoint.com!
27. It is the best tutorial website to learn jQuery and other languages.</div>
28. </body>
29. </html>

```

## jQuery animate()

The jQuery animate() method provides you a way to create custom animations.

### Syntax:

1. \$(selector).animate({params}, speed, callback);

Here, **params** parameter defines the CSS properties to be animated.

The **speed** parameter is optional and specifies the duration of the effect. It can be set as "slow" , "fast" or milliseconds.

The **callback** parameter is also optional and it is a function which is executed after the animation completes.

Let's take a simple example to see the animation effect.

1. <!DOCTYPE html>
2. <html>
3. <head>
4. <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js"></script>

```

5. <script>
6. $(document).ready(function(){
7.   $("button").click(function(){
8.     $("div").animate({left: '450px'});
9.   });
10. });
11. </script>
12. </head>
13. <body>
14. <button>Start Animation</button>
15. <p>A simple animation example:</p>
16. <div style="background:#98bf21;height:100px;width:100px;position:absolute;"></div>
17. </body>
18. </html>

```

## jQuery delay()

The jQuery delay() method is used to delay the execution of functions in the queue. It is a best method to make a delay between the queued jQuery effects. The jQUery delay () method sets a timer to delay the execution of the next item in the queue.

### Syntax:

1. \$(selector).delay (speed, queueName)

**speed:** It is an optional parameter. It specifies the speed of the delay. Its possible values are slow, fast and milliseconds.

**queueName:** It is also an optional parameter. It specifies the name of the queue. Its default value is "fx" the standard queue effect.

Let's take an example to see the delay effect:

1. <!DOCTYPE html>
2. <html>
3. <head>
4. <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"></script>
5. <script>
6. \$(document).ready(function(){

```

7. $("button").click(function(){
8.   $("#div1").delay("slow").fadeIn();
9. });
10. });
11. </script>
12. </head>
13. <body>
14. <button>Click me</button><br>
15. <div id="div1" style="width:90px;height:90px;display:none;background-color:black;"></div><br>
16. </body>
17. </html>

```

### jQuery delay() example with different values

Let's see a jQuery delay() effect example where we are using fast, slow and milliseconds values.

```

1. <!DOCTYPE html>
2. <html>
3. <head>
4. <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"></script>
5. <script>
6. $(document).ready(function(){
7.   $("button").click(function(){
8.     $("#div1").delay("fast").fadeIn();
9.     $("#div2").delay("slow").fadeIn();
10.    $("#div3").delay(1000).fadeIn();
11.    $("#div4").delay(2000).fadeIn();
12.    $("#div5").delay(4000).fadeIn();
13.  });
14. });
15. </script>
16. </head>
17. <body>
18. <p>This example sets different speed values for the delay() method.</p>
19. <button>Click to fade in boxes with a different delay time</button>
20. <br><br>

```

```

21. <div id="div1" style="width:90px;height:90px;display:none;background-color:black;"></div><br>
22. <div id="div2" style="width:90px;height:90px;display:none;background-color:green;"></div><br>
23. <div id="div3" style="width:90px;height:90px;display:none;background-color:blue;"></div><br>
24. <div id="div4" style="width:90px;height:90px;display:none;background-color:red;"></div><br>
25. <div id="div5" style="width:90px;height:90px;display:none;background-color:purple;"></div><br>
26. </body>
27. </html>

```

## How to change the background image using jQuery?

Changing a background-image using jQuery is an easy task. We can use the **css()** method and the **url()** function notation to change the background-image.

The syntax to change the background-image using **jQuery** is given as follows.

1. `$(“selector”).css({“background-image”: “url(image)”});`

We can directly pass the image to the **url()** function as given above, or it can be done by storing the image in a variable and pass the variable name to the **url** as given below.

1. `var img = “image”;`
2. `$(“selector”).css({“background-image”: “url(“ + img + “)”});`

**Example**

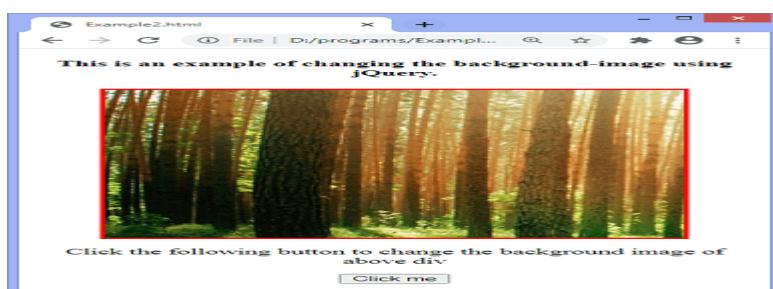
In this example, there is a div element with its background-image. We are using the **css()** method and **url()** function notation to change the background-image of the corresponding div element.

1. `<!DOCTYPE html>`
2. `<html>`
3. `<head>`
4. `<style>`
5. `#d1 {`
6. `border: 3px solid red;`
7. `background-image: url("forest.jpg");`
8. `height: 300px;`
9. `width: 350px;`
10. `}`
- 11.

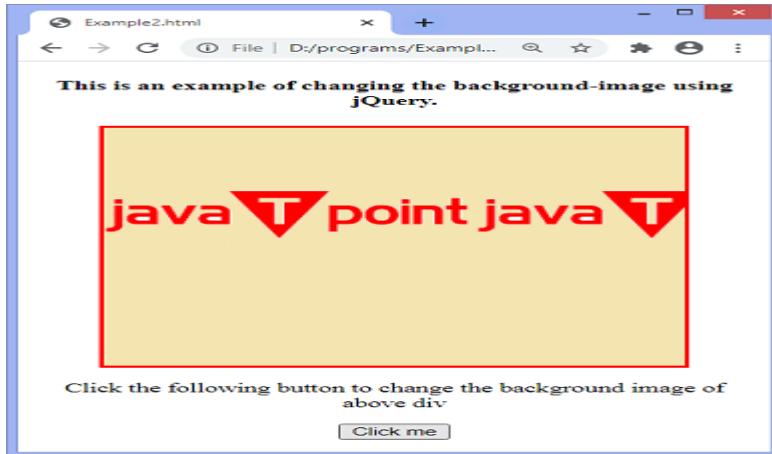
```
12. </style>
13. <script src = "https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"> </script>
14.
15. <script>
16. $(document).ready(function() {
17.   $("button").click(function() {
18.     $("#d1").css({"background-image": "url(jtp.png)"});
19.   });
20. });
21. </script>
22. </head>
23.
24. <body>
25. <center>
26. <h4> This is an example of changing the background-image using jQuery. </h4>
27. <div id = "d1"> </div>
28. <p> Click the following button to change the background image of above div </p>
29. <button>
30. Click me
31. </button>
32. </center>
33. </body>
34.
35. </html>
```

### Output

After the execution of the above code, the output will be -



On clicking the given button, the output will be -



## What is LINQ?

**LINQ** (Language Integrated Query) is a powerful feature in .NET languages (like C# and VB.NET) that allows you to write queries directly within your code using a syntax that is similar to SQL. LINQ enables querying and manipulating data from different sources like collections, databases, XML, and more in a consistent manner.

### Why Use LINQ?

- Readable Code:** LINQ queries are more readable and expressive than traditional loops and conditions.
- Type Safety:** Since LINQ is integrated into the language, it provides compile-time checking, making it less prone to runtime errors.
- Flexibility:** LINQ can query different data sources such as arrays, lists, XML documents, databases, and more.
- Less Code:** LINQ allows you to perform complex queries with fewer lines of code.

### LINQ Syntax

LINQ syntax can be written in two main styles:

- Query Syntax** (similar to SQL):

```

var result = from item in collection
            where condition
            select item;

```

- Method Syntax** (using extension methods):

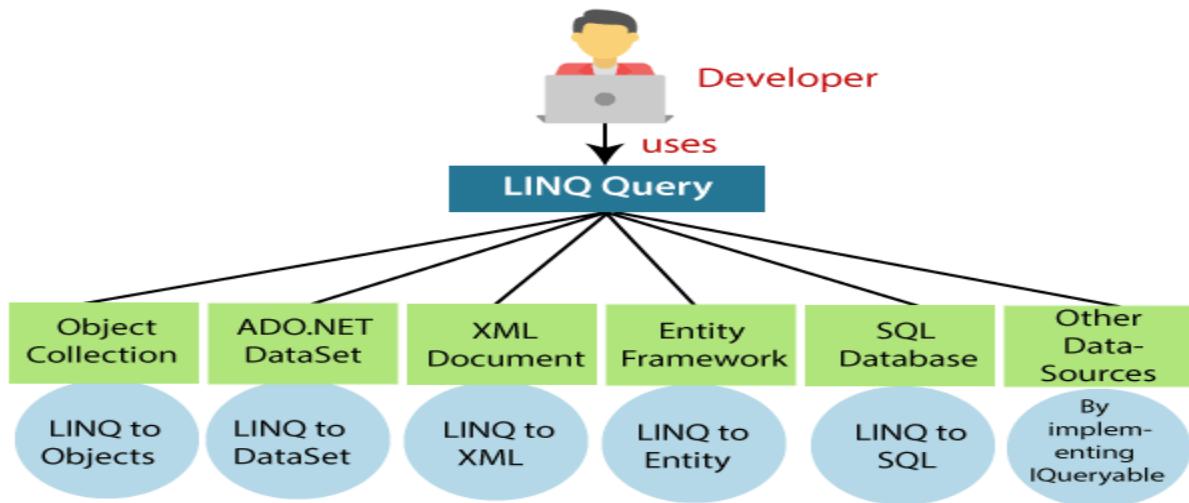
```
var result = collection.Where(item => condition)
```

```
.Select(item => item);
```

## LINQ Providers

LINQ works with different data sources through **LINQ providers**:

- **LINQ to Objects:** Queries on in-memory collections like arrays, lists, etc.
- **LINQ to SQL:** Queries on a SQL Server database.
- **LINQ to XML:** Queries on XML documents.
- **LINQ to Entities:** Part of Entity Framework, it allows querying on the database via entity objects.



## Linq to object:

**LINQ to Objects** refers to the use of LINQ (Language Integrated Query) to query collections of objects in memory, such as arrays, lists, or other `IEnumerable<T>` collections. With LINQ to Objects, you can perform complex filtering, ordering, and grouping operations on in-memory data structures in a concise and readable way.

### Key Features of LINQ to Objects

- **No External Dependencies:** LINQ to Objects works directly with collections like arrays, lists, dictionaries, etc., without requiring any external data source like a database.
- **Query Capabilities:** LINQ allows you to perform SQL-like operations such as where, select, group by, order by, and more on in-memory data.
- **Deferred Execution:** LINQ queries are not executed when they are defined. Instead, they are executed when you iterate over the query result. This allows for more efficient querying and data processing.

## Common LINQ Operations

Let's look at some common operations using LINQ to Objects with examples.

### 1. Filtering (Where)

The Where method filters a collection based on a given condition.

**Example:**

```
using System;
using System.Collections.Generic;
using System.Linq;

class Program
{
    static void Main()
    {
        // Sample list of integers
        List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

        // Using LINQ to filter even numbers
        var evenNumbers = numbers.Where(n => n % 2 == 0);

        // Displaying the result
        Console.WriteLine("Even Numbers:");
        foreach (var number in evenNumbers)
        {
            Console.WriteLine(number);
        }
    }
}
```

**Explanation:**

- The Where method filters the numbers list to include only even numbers ( $n \% 2 == 0$ ).
- The query result is then iterated to display each even number.

### 2. Projection (Select)

The Select method is used to project each element of a collection into a new form.

**Example:**

```

using System;
using System.Collections.Generic;
using System.Linq;

class Program
{
    static void Main()
    {
        // Sample list of strings
        List<string> fruits = new List<string> { "Apple", "Banana", "Mango", "Orange" };

        // Using LINQ to select the length of each fruit name
        var fruitLengths = fruits.Select(fruit => fruit.Length);

        // Displaying the result
        Console.WriteLine("Length of each fruit name:");
        foreach (var length in fruitLengths)
        {
            Console.WriteLine(length);
        }
    }
}

```

**Explanation:**

- The Select method projects each fruit name into its length.
- The result is a collection of integers representing the length of each fruit name.

### 3. Sorting (OrderBy, OrderByDescending)

The OrderBy and OrderByDescending methods are used to sort a collection in ascending and descending order, respectively.

**Example:**

```

using System;
using System.Collections.Generic;
using System.Linq;

class Program
{
    static void Main()
    {
        // Sample list of integers

```

```

List<int> numbers = new List<int> { 7, 1, 4, 9, 6, 3, 8, 2, 5 };

// Using LINQ to order numbers in ascending order
var sortedNumbers = numbers.OrderBy(n => n);

// Using LINQ to order numbers in descending order
var descendingNumbers = numbers.OrderByDescending(n => n);

// Displaying the results
Console.WriteLine("Numbers in ascending order:");
foreach (var number in sortedNumbers)
{
    Console.WriteLine(number);
}

Console.WriteLine("Numbers in descending order:");
foreach (var number in descendingNumbers)
{
    Console.WriteLine(number);
}
}
}

```

**Explanation:**

- `OrderBy(n => n)` sorts the numbers in ascending order.
- `OrderByDescending(n => n)` sorts the numbers in descending order.

## 4. Grouping (GroupBy)

The `GroupBy` method groups elements of a collection based on a specified key.

**Example:**

```

using System;
using System.Collections.Generic;
using System.Linq;

class Program
{
    static void Main()
    {
        // Sample list of strings
        List<string> words = new List<string> { "apple", "banana", "apricot", "blueberry", "avocado", "blackberry" };
    }
}

```

```
// Using LINQ to group words by their first letter
var wordGroups = words.GroupBy(word => word[0]);

// Displaying the grouped result
foreach (var group in wordGroups)
{
    Console.WriteLine($"Words that start with '{group.Key}':");
    foreach (var word in group)
    {
        Console.WriteLine(word);
    }
}
```

**Explanation:**

- The GroupBy method groups words based on their first letter (word[0]).
- Each group is displayed with its key (the first letter) and the corresponding words.

## 5. Aggregating (Sum, Count, Average, etc.)

LINQ provides various aggregation methods to perform calculations on collections, such as Sum, Count, Average, etc.

**Example:**

```
using System;
using System.Collections.Generic;
using System.Linq;

class Program
{
    static void Main()
    {
        // Sample list of integers
        List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

        // Using LINQ to calculate the sum of all numbers
        int sum = numbers.Sum();

        // Using LINQ to count the number of elements
        int count = numbers.Count();
```

```
// Using LINQ to calculate the average of all numbers
double average = numbers.Average();

// Displaying the results
Console.WriteLine($"Sum: {sum}");
Console.WriteLine($"Count: {count}");
Console.WriteLine($"Average: {average}");

}
```

**Explanation:**

- Sum() calculates the sum of all elements in the list.
- Count() returns the number of elements in the list.
- Average() calculates the average value of the elements in the list.

**Example: Real-World LINQ to Objects Scenario**

Suppose you have a list of employees, and you want to find out which departments have an average salary greater than a certain threshold.

**Code:**

```
using System;
using System.Collections.Generic;
using System.Linq;

class Program
{
    public class Employee
    {
        public string Name { get; set; }
        public string Department { get; set; }
        public double Salary { get; set; }
    }

    static void Main()
    {
        // Sample list of employees
        List<Employee> employees = new List<Employee>
        {
            new Employee { Name = "Alice", Department = "HR", Salary = 60000 },
            new Employee { Name = "Bob", Department = "IT", Salary = 80000 },
        }
    }
}
```

```

new Employee { Name = "Charlie", Department = "IT", Salary = 70000 },
new Employee { Name = "Dave", Department = "HR", Salary = 55000 },
new Employee { Name = "Eve", Department = "Finance", Salary = 90000 }
};

// LINQ query to group by department and calculate the average salary
var highEarningDepartments = from emp in employees
    group emp by emp.Department into deptGroup
    where deptGroup.Average(emp => emp.Salary) > 60000
    select new
    {
        Department = deptGroup.Key,
        AverageSalary = deptGroup.Average(emp => emp.Salary)
    };

// Displaying the results
Console.WriteLine("Departments with an average salary greater than $60,000:");
foreach (var dept in highEarningDepartments)
{
    Console.WriteLine($"Department: {dept.Department}, Average Salary: {dept.AverageSalary}");
}
}
}

```

### **Explanation:**

- Employees are grouped by their department using group by.
- The where clause filters departments with an average salary greater than \$60,000.
- The result is a list of departments and their average salaries, which are then displayed.

### **Summary**

LINQ to Objects allows you to perform sophisticated queries on in-memory collections using a concise and readable syntax. Whether you need to filter, sort, group, or aggregate data, LINQ provides a powerful and flexible toolset that can significantly simplify your code. By leveraging LINQ to Objects, you can write more maintainable, efficient, and expressive code when working with collections.

## **Linq to SQL:**

**LINQ to SQL** is a tool in .NET that makes it easier to work with databases using your regular programming language (like C#). Instead of writing complex SQL commands, you can use simple and familiar code to interact with your database. Here's a basic overview:

## What is LINQ to SQL?

- **LINQ to SQL** is a way to interact with a SQL Server database using LINQ (Language Integrated Query).
- It helps you work with database data as if it were just regular objects in your code.

## Key Components

1. **DataContext**: This is the main class that helps you connect to the database and perform operations like querying and updating data.
2. **Entities**: These are classes that represent tables in the database. Each object of an entity class corresponds to a row in the table.

## How to Use LINQ to SQL

### 1. Setup

- **Create a Database**: You need a database with tables. For example, a table called Students.
- **Generate Classes**: Use Visual Studio to create classes that represent your database tables. This is usually done by dragging the table onto a designer surface which automatically creates these classes.

### 2. Performing Operations

#### 1. Adding Data (Create)

To add a new student to the Students table:

```
using System;
using System.Linq;

class Program
{
    static void Main()
    {
        // Create a DataContext
        var db = new SchoolDBDataContext();

        // Create a new student
        var newStudent = new Student
        {
            StudentID = 1,
            Name = "John Doe",
            Age = 20,
            Grade = "A"
        };
    }
}
```

```

// Add the student to the database
db.Students.InsertOnSubmit(newStudent);
db.SubmitChanges();

Console.WriteLine("Student added successfully!");
}
}

```

**Explanation:**

- You create a new Student object.
- Add this student to the database using InsertOnSubmit.
- Save changes with SubmitChanges.

**2. Retrieving Data (Read)**

To get students older than 18:

```

using System;
using System.Linq;

class Program
{
    static void Main()
    {
        var db = new SchoolDBDataContext();

        // Query to get students older than 18
        var students = from s in db.Students
                      where s.Age > 18
                      select s;

        // Display students
        foreach (var student in students)
        {
            Console.WriteLine($"ID: {student.StudentID}, Name: {student.Name}, Age: {student.Age}");
        }
    }
}

```

**Explanation:**

- Use LINQ to filter students by age.
- Display the list of students who match the criteria.

### 3. Updating Data (Update)

To update a student's grade:

```
using System;
using System.Linq;

class Program
{
    static void Main()
    {
        var db = new SchoolDBDataContext();

        // Find the student with ID 1
        var student = db.Students.SingleOrDefault(s => s.StudentID == 1);

        if (student != null)
        {
            // Change the student's grade
            student.Grade = "B+";
            db.SubmitChanges();

            Console.WriteLine("Student grade updated successfully!");
        }
        else
        {
            Console.WriteLine("Student not found!");
        }
    }
}
```

#### Explanation:

- Retrieve the student using their ID.
- Update the student's grade and save the changes.

### 4. Deleting Data (Delete)

To remove a student:

```
using System;
using System.Linq;

class Program
{
```

```

static void Main()
{
    var db = new SchoolDBDataContext();

    // Find the student with ID 1
    var student = db.Students.SingleOrDefault(s => s.StudentID == 1);

    if (student != null)
    {
        // Remove the student
        db.Students.DeleteOnSubmit(student);
        db.SubmitChanges();

        Console.WriteLine("Student deleted successfully!");
    }
    else
    {
        Console.WriteLine("Student not found!");
    }
}
}

```

### **Explanation:**

- Retrieve the student and mark them for deletion.
- Save changes to remove the student from the database.

### **Benefits**

- **Simpler Code:** You don't have to write complex SQL; just use regular C# code.
- **Integrated:** Queries and updates are written in the same language as your application code.

### **Summary**

LINQ to SQL helps you interact with databases in a more straightforward and integrated way by using C# (or other .NET languages) instead of writing raw SQL. It makes common database operations like adding, retrieving, updating, and deleting data easier and more intuitive.

## **Linq to XML:**

**LINQ to XML** is a tool in .NET that makes working with XML data easier. XML (eXtensible Markup Language) is a format used to store and transport data, and LINQ to XML lets you handle this data using simple and familiar C# (or VB.NET) code.

## What You Can Do with LINQ to XML

1. **Load XML Data:** Read XML files and load them into your program.
2. **Query XML Data:** Find specific information in your XML data.
3. **Modify XML Data:** Change or add data in your XML file.
4. **Delete XML Data:** Remove data from your XML file.

### Examples

#### 1. Loading XML Data

To load an XML file into your program:

```
using System;
using System.Xml.Linq;

class Program
{
    static void Main()
    {
        // Load the XML file
        XDocument doc = XDocument.Load("example.xml");

        // Print the XML content
        Console.WriteLine(doc);
    }
}
```

**Explanation:** This code reads the XML file example.xml and displays its content.

#### 2. Querying XML Data

To find specific information, such as getting all books from an XML file:

```
using System;
using System.Linq;
using System.Xml.Linq;

class Program
{
    static void Main()
    {
        XDocument doc = XDocument.Load("example.xml");
```

```

// Get all books
var books = from book in doc.Descendants("Book")
    select new
    {
        Title = book.Element("Title")?.Value,
        Author = book.Element("Author")?.Value
    };

// Print book details
foreach (var book in books)
{
    Console.WriteLine($"Title: {book.Title}, Author: {book.Author}");
}
}
}

```

**Explanation:** This code finds all <Book> elements in the XML file and prints their titles and authors.

### 3. Modifying XML Data

To add new data, such as adding a new book:

```

using System;
using System.Xml.Linq;

class Program
{
    static void Main()
    {
        XDocument doc = XDocument.Load("example.xml");

        // Create a new book element
        XElement newBook = new XElement("Book",
            new XElement("Title", "New Book Title"),
            new XElement("Author", "New Author"))
        );

        // Add the new book to the XML
        doc.Root.Add(newBook);

        // Save changes to the XML file
        doc.Save("example.xml");

        Console.WriteLine("New book added successfully!");
    }
}

```

```
    }  
}
```

**Explanation:** This code creates a new book element and adds it to the XML file.

## 4. Deleting XML Data

To remove data, such as deleting a book with a specific title:

```
using System;  
using System.Linq;  
using System.Xml.Linq;  
  
class Program  
{  
    static void Main()  
    {  
        XDocument doc = XDocument.Load("example.xml");  
  
        // Find the book to remove  
        XElement bookToRemove = doc.Descendants("Book")  
            .FirstOrDefault(b => b.Element("Title")?.Value == "Book Title to Remove");  
  
        if (bookToRemove != null)  
        {  
            // Remove the book from the XML  
            bookToRemove.Remove();  
            doc.Save("example.xml");  
            Console.WriteLine("Book removed successfully!");  
        }  
        else  
        {  
            Console.WriteLine("Book not found!");  
        }  
    }  
}
```

**Explanation:** This code finds and removes a book with a specific title from the XML file.

## Summary

LINQ to XML allows you to easily work with XML data using C# code. You can load, search, update, and delete XML data without writing complex code. It makes handling XML files simple and integrated with your .NET applications.

## Linq to ado.net:

**LINQ to ADO.NET** (often referred to as **LINQ to DataSet**) is a feature in .NET that lets you use LINQ (Language Integrated Query) to work with data stored in ADO.NET DataSets.

Here's a simple way to understand it:

### What is ADO.NET?

- **ADO.NET** is a technology used to interact with databases in .NET applications.
- **DataSet** is a part of ADO.NET that stores data in memory as tables, similar to how a database stores data.

### What is LINQ to ADO.NET?

- **LINQ to ADO.NET** lets you use LINQ queries to interact with the data in a DataSet.
- This means you can write queries in C# (or VB.NET) to search, filter, and manipulate data in a DataSet, just like you would with a database.

## How to Use LINQ to ADO.NET

### 1. Setting Up

First, you need to have a DataSet filled with data. Here's how you can set up a DataSet with some data:

```
using System;
using System.Data;
using System.Linq;

class Program
{
    static void Main()
    {
        // Create a DataSet
        DataSet dataSet = new DataSet();

        // Create a DataTable
        DataTable table = new DataTable("Students");

```

```

table.Columns.Add("StudentID", typeof(int));
table.Columns.Add("Name", typeof(string));
table.Columns.Add("Age", typeof(int));

// Add some rows
table.Rows.Add(1, "John Doe", 20);
table.Rows.Add(2, "Jane Smith", 22);

// Add the table to the DataSet
dataSet.Tables.Add(table);

// Query the DataSet using LINQ
var students = from student in dataSet.Tables["Students"].AsEnumerable()
    where student.Field<int>("Age") > 21
    select new
    {
        Name = student.Field<string>("Name"),
        Age = student.Field<int>("Age")
    };

// Display the results
foreach (var student in students)
{
    Console.WriteLine($"Name: {student.Name}, Age: {student.Age}");
}
}
}

```

**Explanation:**

- A DataSet is created, and a DataTable named "Students" is added to it.
- Rows are added to the table with student information.
- LINQ queries are used to filter students older than 21.

## 2. Querying Data

You can use LINQ to perform queries on the DataSet:

**Example:**

```

var students = from student in dataSet.Tables["Students"].AsEnumerable()
    where student.Field<int>("Age") < 22
    select new
    {

```

```

    Name = student.Field<string>("Name"),
    Age = student.Field<int>("Age")
};
```

**Explanation:**

- This query filters students who are younger than 22.

### 3. Modifying Data

You can also use LINQ to update data:

**Example:**

```

foreach (DataRow row in dataSet.Tables["Students"].Rows)
{
    if (row.Field<int>("StudentID") == 1)
    {
        row.SetField("Name", "John Doe Updated");
    }
}
```

**Explanation:**

- This code finds the student with StudentID 1 and updates their name.

### 4. Adding and Removing Rows

**Adding Rows:**

```
dataSet.Tables["Students"].Rows.Add(3, "Alice Johnson", 23);
```

**Removing Rows:**

```

DataRow rowToRemove = dataSet.Tables["Students"].Rows.Find(2);
if (rowToRemove != null)
{
    dataSet.Tables["Students"].Rows.Remove(rowToRemove);
}
```

**Explanation:**

- Adding and removing rows from the DataTable is straightforward. You can use Rows.Add() to add and Rows.Remove() to delete rows.

## Benefits

- **Familiar Syntax:** You can use LINQ, which is a familiar and powerful querying language in .NET, to work with in-memory data.
- **Simplifies Data Handling:** LINQ to ADO.NET makes it easier to filter and manipulate data stored in DataSets without having to write complex code.

## Summary

LINQ to ADO.NET allows you to use LINQ to work with data in ADO.NET DataSets. This means you can write simple and readable queries to interact with in-memory data, making data handling easier and more intuitive.

## Linq to dataset:

**LINQ to DataSet** is a feature in .NET that allows you to use LINQ (Language Integrated Query) to work with data stored in ADO.NET DataSets. It makes querying and manipulating in-memory data more straightforward by allowing you to write queries in C# or VB.NET.

### What is a DataSet?

- **DataSet** is a .NET class that can hold data in memory, like a database but not on disk.
- It consists of one or more **DataTables**, which represent tables of data, and relationships between those tables.

### What is LINQ to DataSet?

- **LINQ to DataSet** allows you to use LINQ queries to interact with the data in a DataSet.
- This means you can write queries to filter, sort, and manipulate data in your DataTables using LINQ syntax.

## How to Use LINQ to DataSet

### 1. Setting Up a DataSet

First, you need to create and populate a DataSet. Here's an example:

```
using System;
using System.Data;
using System.Linq;

class Program
{
    static void Main()
```

```
{  
    // Create a DataSet  
    DataSet dataSet = new DataSet();  
  
    // Create a DataTable  
    DataTable table = new DataTable("Students");  
    table.Columns.Add("StudentID", typeof(int));  
    table.Columns.Add("Name", typeof(string));  
    table.Columns.Add("Age", typeof(int));  
  
    // Add rows to the DataTable  
    table.Rows.Add(1, "John Doe", 20);  
    table.Rows.Add(2, "Jane Smith", 22);  
    table.Rows.Add(3, "Sam Brown", 19);  
  
    // Add the DataTable to the DataSet  
    dataSet.Tables.Add(table);  
  
    // Use LINQ to query the DataSet  
    var query = from student in dataSet.Tables["Students"].AsEnumerable()  
               where student.Field<int>("Age") > 20  
               select new  
               {  
                   Name = student.Field<string>("Name"),  
                   Age = student.Field<int>("Age")  
               };  
  
    // Display the results  
    foreach (var student in query)  
    {  
        Console.WriteLine($"Name: {student.Name}, Age: {student.Age}");  
    }  
}
```

## **Explanation:**

- A DataSet is created with one DataTable named "Students".
  - Rows are added to the table with student information.
  - LINQ is used to query students older than 20.

## 2. Querying Data

You can perform queries using LINQ syntax to filter and select data.

**Example:**

```
var youngStudents = from student in dataSet.Tables["Students"].AsEnumerable()
    where student.Field<int>("Age") < 21
    select new
    {
        Name = student.Field<string>("Name"),
        Age = student.Field<int>("Age")
    };
}
```

**Explanation:**

- This query selects students who are younger than 21.

### 3. Sorting Data

You can sort data using LINQ:

**Example:**

```
var sortedStudents = from student in dataSet.Tables["Students"].AsEnumerable()
    orderby student.Field<int>("Age")
    select new
    {
        Name = student.Field<string>("Name"),
        Age = student.Field<int>("Age")
    };
}
```

**Explanation:**

- This query sorts students by age in ascending order.

### 4. Modifying Data

You can modify data by directly interacting with the DataTable.

**Example:**

```
// Update student age
foreach (DataRow row in dataSet.Tables["Students"].Rows)
{
    if (row.Field<int>("StudentID") == 1)
    {
        row.SetField("Age", 21);
    }
}
```

```

    }
}

```

**Explanation:**

- This code updates the age of the student with StudentID 1.

## 5. Adding and Removing Rows

**Adding Rows:**

```
dataSet.Tables["Students"].Rows.Add(4, "Alice Johnson", 23);
```

**Removing Rows:**

```
DataRow rowToRemove = dataSet.Tables["Students"].Rows.Find(2);
if (rowToRemove != null)
{
    dataSet.Tables["Students"].Rows.Remove(rowToRemove);
}
```

**Explanation:**

- Use Rows.Add() to add new rows.
- Use Rows.Remove() to delete specific rows.

**Benefits**

- **Familiar Syntax:** You can use LINQ queries, which are familiar and powerful, to work with in-memory data.
- **Easy Data Manipulation:** LINQ to DataSet simplifies querying and modifying data in a DataSet.

**Summary**

LINQ to DataSet allows you to use LINQ queries to work with data in ADO.NET DataSets. This makes it easier to filter, sort, and manipulate in-memory data using simple and readable C# code.

## Operation (projection, filtering and join ) using linq queries:

Let's explore how to perform the following operations using LINQ queries in C#:

1. **Projection** (using Select)
2. **Filtering** (using Where)
3. **Join** (using Join)

We'll use a scenario involving two lists: a list of students and a list of courses.

## Setup

Let's start with the data classes and sample data:

```
public class Student
{
    public int StudentId { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}

public class Course
{
    public int CourseId { get; set; }
    public string CourseName { get; set; }
    public int StudentId { get; set; } // Foreign key linking to Student
}
```

```
List<Student> students = new List<Student>
{
    new Student { StudentId = 1, Name = "John", Age = 18 },
    new Student { StudentId = 2, Name = "Jane", Age = 20 },
    new Student { StudentId = 3, Name = "Sam", Age = 17 },
};
```

```
List<Course> courses = new List<Course>
{
    new Course { CourseId = 101, CourseName = "Math", StudentId = 1 },
    new Course { CourseId = 102, CourseName = "Science", StudentId = 2 },
    new Course { CourseId = 103, CourseName = "History", StudentId = 3 },
    new Course { CourseId = 104, CourseName = "Math", StudentId = 3 },
};
```

## 1. Projection (Using Select)

Projection allows you to transform the data in a collection. Suppose you want to create a list of student names only:

```
var studentNames = students.Select(s => s.Name).ToList();

foreach (var name in studentNames)
{
    Console.WriteLine(name);
```

```
}
```

- **Explanation:**

- Select(s => s.Name) projects each Student object to just its Name property.

## 2. Filtering (Using Where)

Filtering allows you to retrieve a subset of the data based on a condition. For instance, if you want to get a list of students who are 18 years or older:

```
var adultStudents = students.Where(s => s.Age >= 18).ToList();
```

```
foreach (var student in adultStudents)
{
    Console.WriteLine($"{student.Name}, {student.Age}");
}
```

- **Explanation:**

- Where(s => s.Age >= 18) filters the students list to include only those who are 18 or older.

## 3. Joining (Using Join)

Joining combines elements from two collections based on a common key. For example, if you want to get a list of students along with the courses they are enrolled in:

```
var studentCourses = students.Join(
    courses,
    student => student.StudentId,
    course => course.StudentId,
    (student, course) => new
    {
        StudentName = student.Name,
        CourseName = course.CourseName
    }).ToList();

foreach (var sc in studentCourses)
{
    Console.WriteLine($"{sc.StudentName} is enrolled in {sc.CourseName}");
}
```

- **Explanation:**

- Join matches StudentId in the students list with StudentId in the courses list.
- The result is projected into a new anonymous type containing the StudentName and CourseName.

## Summary

- **Projection** (Select) transforms data from one form to another.
- **Filtering** (Where) extracts a subset of data based on a condition.
- **Join** combines data from two collections based on a common key.

These LINQ operations allow you to manipulate and query your collections efficiently and with clear, readable code.

## Introduction to ADO.net entity framework:

### What is entity frame work?

**Entity Framework (EF)** is a tool in .NET that helps you work with databases using C# (or another .NET language) in a simple and straightforward way. Instead of writing a lot of SQL code to interact with the database, you can use C# objects and classes.

#### What Does Entity Framework Do?

1. **Works with Data as Objects:**
  - Imagine you have a table in your database called Students. With EF, you can create a Student class in C#, and Entity Framework will connect that class to the Students table. This means you can work with your data as objects, which is easier to understand and work with.
2. **No Need for SQL Code:**
  - Normally, to interact with a database, you would need to write SQL code to fetch, insert, update, or delete data. EF lets you do all of this using simple C# code instead of SQL.

#### How Entity Framework Works

1. **Create Classes for Your Data:**
  - You start by creating C# classes that represent your database tables. For example:

```
public class Student
{
    public int StudentId { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}
```

2. **Set Up a DbContext:**
  - The DbContext class is like a bridge between your code and the database. You create a DbContext class that tells EF which tables you want to work with:

```
public class SchoolContext : DbContext
```

```
{
    public DbSet<Student> Students { get; set; }
}
```

### 3. Use C# to Interact with the Database:

- Once everything is set up, you can use C# code to get data from the database, add new data, update existing data, or delete data. EF takes care of translating your C# code into SQL.

```
using (var context = new SchoolContext())
{
    // Get all students who are older than 18
    var students = context.Students.Where(s => s.Age > 18).ToList();
}
```

### Example Operations with Entity Framework

- Adding a New Student:**

```
context.Students.Add(new Student { Name = "John", Age = 20 });
context.SaveChanges();
```

- Finding a Student:**

```
var student = context.Students.Find(1);
```

- Updating a Student:**

```
student.Name = "John Doe";
context.SaveChanges();
```

- Deleting a Student:**

```
context.Students.Remove(student);
context.SaveChanges();
```

### Why Use Entity Framework?

- Easier to Use:** You can work with data using familiar C# code instead of learning complex SQL.
- Less Code:** EF reduces the amount of code you need to write to interact with a database.
- Keeps Things Organized:** By working with objects, your code is more organized and easier to read.

## Entity data model:

The **Entity Data Model (EDM)** is a key part of the Entity Framework architecture. It provides a way to describe the data in your application in a way that's independent of the database. The EDM defines the structure of your

data and the relationships between different pieces of data, allowing Entity Framework to translate between your application's classes and the underlying database.

## Components of the Entity Data Model

The Entity Data Model consists of three main components:

1. **Conceptual Model**
2. **Storage Model**
3. **Mapping Model**

### 1. Conceptual Model

- **What it is:** This is where you define your data as classes, properties, and relationships in a way that makes sense for your application.
- **Example:** If you have a Student class with properties like StudentId, Name, and Age, this class would be part of your conceptual model.
- **Purpose:** The conceptual model is independent of the database. It represents your application's view of the data.

### 2. Storage Model

- **What it is:** This part of the EDM describes the actual structure of the database, including tables, columns, relationships, and constraints.
- **Example:** In your database, there might be a table called Students with columns StudentId, Name, and Age. This table is part of the storage model.
- **Purpose:** The storage model represents the physical database schema.

### 3. Mapping Model

- **What it is:** The mapping model connects the conceptual model to the storage model. It defines how the classes and properties in the conceptual model relate to the tables and columns in the storage model.
- **Example:** It maps the Student class in your code to the Students table in the database, and the Name property of the Student class to the Name column in the Students table.
- **Purpose:** The mapping model ensures that data can be translated between your application's objects and the database.

## How the Entity Data Model Works

When you use Entity Framework to query data, insert new records, or update existing records, the EDM plays a crucial role:

1. **Querying Data:** When you write a LINQ query, EF uses the EDM to translate your query into SQL. The mapping model helps EF understand which database tables and columns to query.

- Inserting/Updating Data:** When you add or modify data, EF uses the EDM to map your changes to the appropriate tables and columns in the database.

### Example of EDM in Action

Imagine you have a Student class in your application:

```
public class Student
{
    public int StudentId { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}
```

And you have a Students table in your database:

StudentId	Name	Age
-----------	------	-----

1	John	20
2	Jane	22

Here's how the EDM maps this:

- Conceptual Model:** The Student class with StudentId, Name, and Age.
- Storage Model:** The Students table with columns StudentId, Name, and Age.
- Mapping Model:** Links Student.StudentId to Students.StudentId, Student.Name to Students.Name, and Student.Age to Students.Age.

### Summary

- Conceptual Model:** Your application's classes and properties.
- Storage Model:** The actual tables and columns in your database.
- Mapping Model:** The glue that connects your classes to the database schema.

The Entity Data Model allows Entity Framework to seamlessly translate between the way you think about your data in your application and the way the data is stored in the database.

## CSDL:

The term **CSDL** stands for **Conceptual Schema Definition Language**. It's a language used in the context of the Entity Data Model (EDM) within Entity Framework. CSDL describes the conceptual model, which is the application's view of the data, independent of how the data is stored in the database.

## What is CSDL?

- **CSDL** is an XML-based language that defines the structure of the conceptual model in the EDM. This includes the entities (like classes), their properties (like fields), and the relationships between them (like associations or navigations).
- It describes how the data is structured from the perspective of the application, not the database. This means it focuses on how the data is used and understood in your code.

## Key Components of CSDL

1. **Entity Types:**
  - These represent the classes in your application.
  - Example: If you have a Student class, it will be represented as an EntityType in CSDL.
2. **Properties:**
  - These define the data fields or attributes of each entity type.
  - Example: The Name and Age properties of the Student class would be defined as properties within the Student entity type.
3. **Associations:**
  - These represent the relationships between different entity types.
  - Example: If a Student is enrolled in multiple Courses, the relationship between Student and Course would be defined as an association.
4. **Navigation Properties:**
  - These allow you to navigate from one entity to another, based on the relationships.
  - Example: You could have a navigation property in the Student entity that lets you access the Courses the student is enrolled in.

## How CSDL Fits into the Entity Framework

- **CSDL** is one part of the overall Entity Data Model (EDM), which also includes the Storage Schema Definition Language (SSDL) for the database schema and the Mapping Schema Definition Language (MSL) for mapping between the conceptual and storage models.
- **CSDL** is generated when you create your Entity Framework model, either by using a visual designer or by writing it by hand (in advanced scenarios). It is typically part of the .edmx file, which combines CSDL, SSDL, and MSL into a single XML file.

## Example of CSDL

Here's a very simplified example of what CSDL might look like for a Student entity:

```
<Schema Namespace="SchoolModel" Alias="Self" xmlns="http://schemas.microsoft.com/ado/2008/09/edm">
<EntityType Name="Student">
<Key>
<PropertyRef Name="StudentId" />
</Key>
```

```

<Property Name="StudentId" Type="Int32" Nullable="false" />
<Property Name="Name" Type="String" Nullable="false" />
<Property Name="Age" Type="Int32" Nullable="false" />
</EntityType>
</Schema>

```

- **EntityType Name="Student"**: This defines an entity named Student.
- **Key**: Defines StudentId as the primary key.
- **Properties**: StudentId, Name, and Age are defined as properties of the Student entity.

### Summary

- **CSDL** is the part of Entity Framework that describes how your data is structured within your application, independent of the database.
- It defines entities, their properties, and relationships in an XML format.
- CSDL is used in the Entity Data Model to map your application's data to the actual database schema.

In essence, CSDL helps Entity Framework understand how to represent and work with your data in the context of your application.

## Eager vs lazy loading:

**Eager loading** and **lazy loading** are two strategies used in Entity Framework (and other ORM frameworks) to load related data from a database. These strategies determine when and how related data (like navigation properties) is loaded when you query an entity.

### 1. Eager Loading

**Eager loading** means that related data is loaded from the database at the same time as the main entity. In other words, when you query for an entity, Entity Framework will also retrieve the related data in a single query, avoiding the need for additional queries later on.

#### How to Use Eager Loading:

- You can use the **Include** method in Entity Framework to specify which related data should be loaded eagerly.

#### Example:

Suppose you have two entities: Student and Course. A Student can enroll in multiple Courses.

```

var students = context.Students
    .Include(s => s.Courses) .ToList();

```

- **What happens:** When you query for Students, Entity Framework also retrieves the Courses each Student is enrolled in at the same time.
- **Advantage:** Reduces the number of database queries because all necessary data is fetched in one go.
- **Disadvantage:** Can load more data than you need, which might slow down performance if the related data set is large.

## 2. Lazy Loading

**Lazy loading** means that related data is only loaded when you access it for the first time. When you query for an entity, Entity Framework retrieves just the main entity, and the related data is fetched separately only when it is accessed in your code.

How to Use Lazy Loading:

- In Entity Framework, lazy loading is enabled by default if you use navigation properties that are marked as virtual.

Example:

Using the same Student and Course example:

```
var student = context.Students.First();
var courses = student.Courses; // This triggers a separate query to load Courses
```

- **What happens:** Initially, only the Student entity is loaded. When you access student.Courses, a separate query is made to the database to load the related Courses.
- **Advantage:** Avoids loading unnecessary data upfront, which can improve performance if you don't need the related data immediately.
- **Disadvantage:** Can result in multiple database queries, which might lead to performance issues known as the "N+1 query problem" if not managed properly.

### Summary of Differences

- **Eager Loading:**
  - Loads related data immediately with the main entity.
  - Fewer database queries, but might load more data than necessary.
  - Use when you know you will need the related data.
- **Lazy Loading:**
  - Loads related data only when it's accessed.
  - More efficient upfront, but might result in multiple queries.
  - Use when you might not need the related data immediately.

## Choosing Between Eager and Lazy Loading

- **Eager Loading** is better when:
  - You know you will need the related data.
  - You want to avoid multiple database queries.
- **Lazy Loading** is better when:
  - You might not need the related data at all.
  - You want to minimize the initial data load and improve performance on the first query.

Choosing the right strategy depends on the specific use case and performance considerations in your application.

## POCO classes:

**POCO** stands for **Plain Old CLR Object**. In the context of Entity Framework, POCO classes are simple C# classes that represent your data model without any dependency on the Entity Framework itself. They are used to define the structure of your data in a way that is clean and easy to work with.

### Key Features of POCO Classes

1. **No Dependencies on Entity Framework:**
  - POCO classes do not require any specific base class or interface from Entity Framework. This means they are not tied to the framework, making them reusable and easy to test.
2. **Properties Only:**
  - They typically contain properties that represent the data fields. There are no methods or behavior; they are just data containers.
3. **Encapsulation:**
  - POCO classes can encapsulate data and behavior. While they primarily focus on data properties, you can also add methods if needed (although this is not common).
4. **Easy to Work With:**
  - They make it easy to manage data in a way that is straightforward and understandable, without dealing with the complexities of the Entity Framework.

### Example of a POCO Class

Here's a simple example of a POCO class representing a Student:

```
public class Student
{
    public int StudentId { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}
```

## How POCO Classes Work with Entity Framework

- When you create a POCO class, you can use it in Entity Framework to define your data model.
- Entity Framework can map these POCO classes to database tables without requiring any special attributes or interfaces.
- You can also create relationships between POCO classes, like having a Course class that relates to the Student class.

### Example with Relationships

Here's how you can define a Course class and create a relationship with the Student class:

```
public class Course
{
    public int CourseId { get; set; }
    public string Title { get; set; }

    // Navigation property to link to Students
    public virtual ICollection<Student> Students { get; set; }
}
```

### Advantages of Using POCO Classes

- **Separation of Concerns:** POCO classes help separate your business logic from the data access logic.
- **Testability:** Since they have no dependencies on Entity Framework, they are easier to test in isolation.
- **Flexibility:** You can use POCO classes with different data access technologies, not just Entity Framework.

### Summary

- **POCO Classes:** Simple C# classes representing your data without ties to any framework.
- **Focus on Data:** They primarily store data through properties.
- **Easy to Use:** They work well with Entity Framework for mapping to database tables.

In summary, POCO classes are a clean and effective way to represent data in your application, making it easier to manage and work with that data.

## DB context API:

The **DbContext** API is a key component of Entity Framework, providing a way to interact with the database using your domain model (like POCO classes). It serves as a bridge between your application and the database, allowing you to perform CRUD (Create, Read, Update, Delete) operations on your data.

### Key Features of DbContext

1. **Database Connection:**

- **DbContext** manages the connection to the database. It uses a connection string from your configuration file to connect to the appropriate database.

## 2. DbSet Properties:

- You define **DbSet** properties in your **DbContext** class for each entity type you want to work with. Each **DbSet** represents a table in the database.

## 3. Change Tracking:

- **DbContext** keeps track of changes made to your entities. When you modify an entity, **DbContext** knows what has changed and can generate the appropriate SQL commands to save those changes to the database.

## 4. Querying Data:

- You can use LINQ queries to retrieve data through **DbContext**. It allows you to write expressive queries using your C# objects.

## 5. SaveChanges Method:

- After making changes to your entities, you call the **SaveChanges** method to persist those changes to the database.

## Creating a DbContext Class

Here's a simple example of how to create a **DbContext** class:

```
using System.Data.Entity;
```

```
public class SchoolContext : DbContext
{
    public SchoolContext() : base("name=SchoolDb") // Connection string name
    {
    }

    public DbSet<Student> Students { get; set; } // Table for Students
    public DbSet<Course> Courses { get; set; } // Table for Courses
}
```

## Performing CRUD Operations

### 1. Creating Entities

To add a new student:

```
using (var context = new SchoolContext())
{
    var student = new Student { Name = "Alice", Age = 21 };
    context.Students.Add(student);
    context.SaveChanges(); // Save changes to the database
}
```

## 2. Reading Entities

To retrieve a list of students:

```
using (var context = new SchoolContext())
{
    var students = context.Students.ToList(); // Get all students
}
```

## 3. Updating Entities

To update an existing student:

```
using (var context = new SchoolContext())
{
    var student = context.Students.Find(1); // Find student by ID
    if (student != null)
    {
        student.Age = 22; // Update age
        context.SaveChanges(); // Save changes
    }
}
```

## 4. Deleting Entities

To delete a student:

```
using (var context = new SchoolContext())
{
    var student = context.Students.Find(1); // Find student by ID
    if (student != null)
    {
        context.Students.Remove(student); // Remove student
        context.SaveChanges(); // Save changes
    }
}
```

### Summary

- **DbContext** is a central part of Entity Framework that helps you interact with your database using your object model.
- It manages connections, tracks changes, and provides methods to perform CRUD operations.
- By defining **DbSet** properties, you can easily work with your entity classes, allowing you to query and manipulate data efficiently.

In short, the **DbContext API** simplifies data access in your application, making it easier to work with databases using C# objects.

# Querying Entity Data Model

Querying an Entity Data Model (EDM) in Entity Framework allows you to retrieve data from the database using LINQ (Language Integrated Query). This approach makes it easy to work with your data in a type-safe and readable way. Below are the main aspects of querying an EDM:

## Basics of Querying with LINQ

1. **DbContext**: Start by creating an instance of your **DbContext** class, which represents the session with the database.
2. **DbSet**: Use the **DbSet** properties defined in your **DbContext** to access the tables in your database.
3. **LINQ Queries**: Write LINQ queries to filter, sort, and manipulate data.

## Example of a DbContext Class

Assuming you have a `SchoolContext` class with `Student` and `Course` entities:

```
public class SchoolContext : DbContext
{
    public DbSet<Student> Students { get; set; }
    public DbSet<Course> Courses { get; set; }
}
```

## Querying Data

Here are some common examples of querying data using Entity Framework and LINQ:

### 1. Retrieving All Records

To get all students from the database:

```
using (var context = new SchoolContext())
{
    var students = context.Students.ToList(); // Gets all students
}
```

### 2. Filtering Records

To find students older than 20:

```
using (var context = new SchoolContext())
{
    var students = context.Students
        .Where(s => s.Age > 20)
        .ToList(); // Gets students older than 20
```

```
}
```

### 3. Sorting Records

To get students sorted by their names:

```
using (var context = new SchoolContext())
{
    var students = context.Students
        .OrderBy(s => s.Name)
        .ToList(); // Gets students sorted by name
}
```

### 4. Selecting Specific Properties

To get a list of student names and ages:

```
using (var context = new SchoolContext())
{
    var studentInfo = context.Students
        .Select(s => new { s.Name, s.Age })
        .ToList(); // Gets a list of names and ages
}
```

### 5. Joining Tables

To get students with their courses, assuming each student can have multiple courses:

```
using (var context = new SchoolContext())
{
    var studentCourses = context.Students
        .Select(s => new
        {
            StudentName = s.Name,
            Courses = s.Courses.Select(c => c.Title) // Assuming Courses is a navigation property
        })
        .ToList(); // Gets students with their course titles
}
```

### 6. Grouping Records

To count how many students are in each age group:

```
using (var context = new SchoolContext())
{
    var ageGroups = context.Students
        .GroupBy(s => s.Age)
        .Select(g => new
```

```

    {
        Age = g.Key,
        Count = g.Count()
    })
    .ToList(); // Gets the number of students in each age group
}

```

### Summary of Querying with Entity Framework

- **Use DbContext:** Create an instance of your **DbContext** to connect to the database.
- **DbSet Properties:** Access your entities through **DbSet** properties.
- **LINQ Queries:** Write queries using LINQ methods like Where, OrderBy, Select, and GroupBy to retrieve and manipulate data.
- **Type Safety:** LINQ provides type-safe queries, reducing runtime errors.

### Example of a Complete Query

Here's a complete example that combines several querying techniques:

```

using (var context = new SchoolContext())
{
    var results = context.Students
        .Where(s => s.Age > 20)
        .OrderBy(s => s.Name)
        .Select(s => new
    {
        s.Name,
        Courses = s.Courses.Select(c => c.Title)
    })
    .ToList(); // Gets students older than 20, sorted by name with their course titles
}

```

This example retrieves students older than 20, sorts them by name, and includes their course titles in the result.

In summary, querying an Entity Data Model using Entity Framework and LINQ is straightforward and allows for powerful data manipulation and retrieval.

## ASP.Net MVC

**ASP.NET MVC (Model-View-Controller)** is a framework for building web applications using the MVC design pattern. It separates an application into three main components, allowing for a more organized and manageable codebase. Here's a simple introduction to ASP.NET MVC:

## Key Concepts of ASP.NET MVC

1. **Model:**
  - **Purpose:** Represents the data and the business logic of the application. It handles data retrieval, storage, and manipulation.
  - **Components:** Usually includes classes that represent the data (like POCO classes) and might also include validation logic.
2. **View:**
  - **Purpose:** Represents the user interface of the application. It is responsible for displaying the data to the user and providing a way to interact with it.
  - **Components:** Typically made up of HTML and Razor syntax (a templating engine) that generates dynamic content.
3. **Controller:**
  - **Purpose:** Handles user input and interactions. It processes incoming requests, manipulates the model, and returns the appropriate view.
  - **Components:** Contains action methods that handle requests and return responses.

## How ASP.NET MVC Works

1. **User Request:**
  - A user interacts with the application (e.g., submits a form or clicks a link). This generates an HTTP request that is sent to the server.
2. **Routing:**
  - The request is processed by the ASP.NET routing engine, which determines which controller and action method should handle the request.
3. **Controller:**
  - The selected controller's action method is executed. This method interacts with the model to retrieve or manipulate data.
4. **Model:**
  - The controller may use the model to get data from the database or perform other business logic.
5. **View:**
  - The controller then passes the data to a view. The view generates HTML content using the data and Razor syntax.
6. **Response:**
  - The generated HTML is sent back to the user's browser as an HTTP response.

## Example Workflow

1. **User Interaction:** A user navigates to a URL like `http://example.com/Students/Index`.
2. **Routing:** The routing system maps this URL to the `Index` action method of the `StudentsController`.
3. **Controller Action:** The `Index` method of the `StudentsController` retrieves a list of students from the database and passes it to the view.
4. **Model:** The `Students` model is used to fetch the list of students.
5. **View:** The view (e.g., `Index.cshtml`) formats the student data into HTML.

6. **Response:** The HTML is sent back to the user's browser.

## Basic Components

### 1. Model (Student.cs)

```
public class Student
{
    public int StudentId { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}
```

### 2. View (Index.cshtml)

```
@model IEnumerable<Student>

<h2>Student List</h2>

<table>
    <tr>
        <th>ID</th>
        <th>Name</th>
        <th>Age</th>
    </tr>
    @foreach (var student in Model)
    {
        <tr>
            <td>@student.StudentId</td>
            <td>@student.Name</td>
            <td>@student.Age</td>
        </tr>
    }
</table>
```

### 3. Controller (StudentsController.cs)

```
public class StudentsController : Controller
{
    private readonly SchoolContext _context;

    public StudentsController()
    {
        _context = new SchoolContext();
    }

    public ActionResult Index()
    {
        var students = _context.Students.ToList();
```

```

        return View(students);
    }
}

```

### Advantages of ASP.NET MVC

- **Separation of Concerns:** MVC divides an application into three distinct parts, making it easier to manage and scale.
- **Testability:** The separation of concerns allows for easier unit testing of components.
- **Flexibility:** Provides greater control over HTML and URL routing, which can be beneficial for complex applications.
- **Built-in Features:** ASP.NET MVC includes features like model binding, validation, and routing, which streamline development.

### Summary

- **ASP.NET MVC** is a framework for building web applications using the MVC pattern.
- **Model** handles data and business logic.
- **View** handles the user interface.
- **Controller** handles user input and interactions.

ASP.NET MVC helps in organizing code efficiently and building scalable and maintainable web applications.

## **MVC Controller:**

The **MVC Controller** is an important part of ASP.NET MVC (Model-View-Controller). It helps manage how your web application works by handling user actions, processing data, and showing the right pages. Here's a simple breakdown:

### What Does a Controller Do?

1. **Handle User Actions:**
  - The controller responds when a user does something, like clicking a button or submitting a form.
2. **Process Requests:**
  - It processes the user's request by deciding what needs to happen next. This might mean getting data from a database or performing some calculations.
3. **Work with the Model:**
  - The controller talks to the model (which contains the data) to get or update information. For example, it might ask for a list of students.
4. **Choose the View:**
  - After processing, the controller decides which page (view) to show the user and sends the necessary data to that page.
5. **Send Back a Response:**
  - Finally, the controller returns a response to the user, which could be a web page or some data.

## Example of a Simple Controller

Here's a basic example of a controller for managing students:

### 1. The StudentsController Class

```
public class StudentsController : Controller
{
    private readonly SchoolContext _context; // This connects to the database

    public StudentsController()
    {
        _context = new SchoolContext(); // Set up the connection
    }

    // This shows a list of all students
    public ActionResult Index()
    {
        var students = _context.Students.ToList(); // Get all students
        return View(students); // Show the list in the view
    }

    // This shows details for a specific student
    public ActionResult Details(int id)
    {
        var student = _context.Students.Find(id); // Find the student by ID
        if (student == null)
        {
            return HttpNotFound(); // Show 404 if not found
        }
        return View(student); // Show the student's details
    }

    // This shows a form to create a new student
    public ActionResult Create()
    {
        return View(); // Show the create form
    }

    // This saves the new student to the database
    [HttpPost]
    public ActionResult Create(Student student)
    {
        if (ModelState.IsValid) // Check if the form data is valid
```

```

    {
        _context.Students.Add(student); // Add the new student
        _context.SaveChanges(); // Save to the database
        return RedirectToAction("Index"); // Go back to the student list
    }
    return View(student); // Show the form again if there are errors
}
}

```

### How the Controller Works

- **Setup:** The controller connects to the database using SchoolContext.
- **Index Action:** When a user wants to see all students, this action gets the list and shows it on the page.
- **Details Action:** If a user wants to see details about a specific student, this action finds that student and displays the information.
- **Create Action:** When a user wants to add a new student, this action shows a form. After the user fills it out and submits it, this action saves the new student to the database if everything is correct.

### Summary

- The **MVC Controller** manages user actions in an ASP.NET MVC application.
- It processes requests, interacts with data, chooses what to show, and sends responses back to the user.
- This keeps your code organized and makes it easier to develop web applications.

## ASP.NET Controller Actions and Parameters

An **action method** in ASP.NET MVC is a method within a controller that handles incoming requests from users. When a user interacts with a web application, such as clicking a link or submitting a form, the action method is executed to process that request.

The MVC application uses the routing rules that are defined in the **Global.asax.cs** file. This file is used to parse the URL and determine the path of the controller. Now, controller executes the appropriate action to handle the user request.

### ActionResult Return Type

The ActionResult class is the base class for all action results. Action methods return an instance of this class. There can be different action result types depending on the task that the action is implementing. For example, if an action is to call the View method, the View method returns an instance of the ViewResult which is derived from the ActionResult class.

We can also create action method that returns any type of object like: integer, string etc.

The following table contains built-in action result types.

Action Result	Helper Method	Description
ViewResult	View	It is used to render a view as a Web page.
PartialViewResult	PartialView	It is used to render a partial view.
RedirectResult	Redirect	It is used to redirect to another action method by using its URL.
RedirectToRouteResult	RedirectToAction RedirectToRoute	It is used to redirect to another action method.
ContentResult	Content	It is used to return a user-defined content type.
JsonResult	Json	It is used to return a serialized JSON object.
JavaScriptResult	JavaScript	It is used to return a script that can be executed on the client.
FileResult	File	It is used to return binary output to write to the response.
EmptyResult	(None)	It represents a return value that is used if the action method must return a null result.

### Adding an Action method

Here, we will add a new action method to the controller that we created in previous chapter.

To add action to the existing controller, we need to define a public method to our controller. Our **MusicStoreController.cs** file looks like the following after adding a welcome action method.

```
// MusicStoreController.cs

1. using System;
2. using System.Collections.Generic;
3. using System.Linq;
4. using System.Web;
5. using System.Web.Mvc;
6. namespace MvcApplicationDemo.Controllers
7. {
8.     public class MusicStoreController : Controller
9.     {
10.         // GET: MusicStrore
11.         public ActionResult Index()
12.         {
13.             return View();
14.         }
15.         public string Welcome()
16.         {
17.             return "Hello, this is welcome action message";
18.         }
19.     }
20. }
```

Output:

To access the welcome action method, execute the application then access it by using **MusicStore/Welcome** URL. It will produce the following output.



## Action Method Parameters

Action parameters are the variables that are used to retrieve user requested values from the URL.

The parameters are retrieved from the request's data collection. It includes name/value pairs for form data, query string value etc. the controller class locates for the parameters values based on the RouteData instance. If the value is present, it is passed to the parameter. Otherwise, exception is thrown.

The Controller class provides two properties Request and Response that can be used to get handle user request and response.

### Example

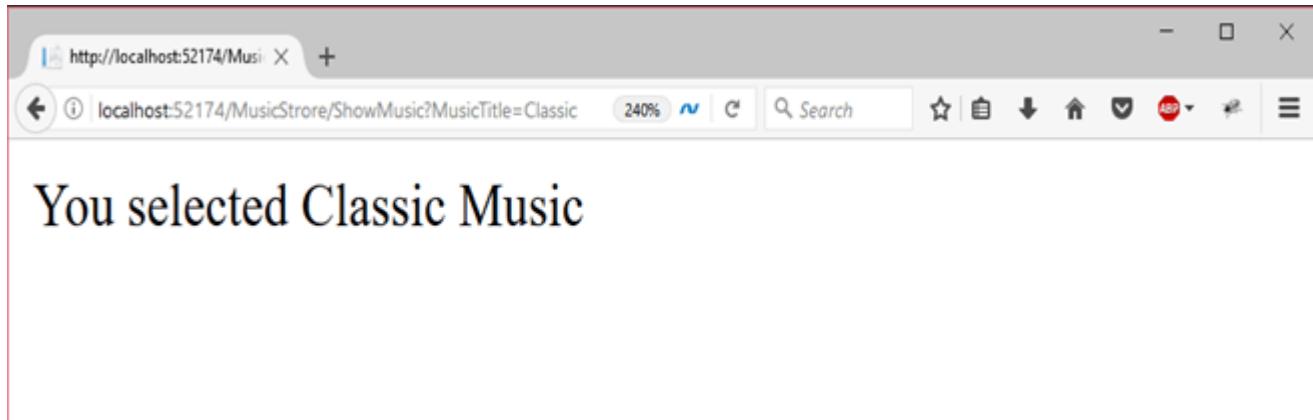
Here, we are creating an action method in the controller. This action method has a parameter. The controller code looks like this:

```
// MusicStoreController.cs

1. using System;
2. using System.Collections.Generic;
3. using System.Linq;
4. using System.Web;
5. using System.Web.Mvc;
6. namespace MvcApplicationDemo.Controllers
7. {
8.     public class MusicStoreController : Controller
9.     {
10.         // GET: MusicStrore
11.         public ActionResult Index()
12.         {
13.             return View();
14.         }
15.         public string ShowMusic(string MusicTitle)
16.         {
17.             return "You selected " + MusicTitle + " Music";
18.         }
19.     }
20. }
```

Output:

In URL, we have to pass the parameter value. So, we are doing it by this URL **localhost:port-no/MusicStore>ShowMusic?MusicTitle=Classic**. it produces the following result.



## **Action method:**

An **action method** in ASP.NET MVC is a method within a controller that handles incoming requests from users. When a user interacts with a web application, such as clicking a link or submitting a form, the action method is executed to process that request. Here's a breakdown of what action methods are and how they work:

### **Key Features of Action Methods**

1. **Location:**
  - Action methods are found in controllers, which are classes that inherit from the Controller base class.
2. **Naming:**
  - Action methods typically have descriptive names that indicate what they do, such as Index, Create, Edit, or Delete.
3. **Return Types:**
  - Action methods usually return an ActionResult, which represents the result of the action. This can be a view, JSON data, or a redirect to another action.
4. **HTTP Verbs:**
  - Action methods can be decorated with attributes to specify which HTTP methods (like GET, POST, PUT, DELETE) they respond to. For example, [HttpGet] for retrieving data and [HttpPost] for submitting data.

### **Example of Action Methods**

Here's a simple example of a controller with different action methods for managing students:

### StudentsController Class

```
using System.Web.Mvc;
using YourNamespace.Models; // Change to your actual namespace

public class StudentsController : Controller
{
    private readonly SchoolContext _context; // Database context

    public StudentsController()
    {
        _context = new SchoolContext(); // Initialize the context
    }

    // GET: Students
    public ActionResult Index()
    {
        var students = _context.Students.ToList(); // Retrieve all students
        return View(students); // Return the view with the student list
    }

    // GET: Students/Details/5
    public ActionResult Details(int id)
    {
        var student = _context.Students.Find(id); // Find student by ID
        if (student == null)
        {
            return HttpNotFound(); // Return 404 if not found
        }
        return View(student); // Return the details view
    }

    // GET: Students/Create
    public ActionResult Create()
    {
        return View(); // Return the create view
    }

    // POST: Students/Create
    [HttpPost]
    public ActionResult Create(Student student)
    {
        if (ModelState.IsValid) // Check if the model state is valid
        {
```

```

        _context.Students.Add(student); // Add the new student
        _context.SaveChanges(); // Save changes to the database
        return RedirectToAction("Index"); // Redirect to the index action
    }
    return View(student); // Return the view with the student data if validation fails
}
}

```

### Explanation of Action Methods in the Example

#### 1. Index Action:

- **Purpose:** Retrieves a list of all students from the database and displays it.
- **Method:** public ActionResult Index()
- **Returns:** A view that shows the list of students.

#### 2. Details Action:

- **Purpose:** Displays details of a specific student based on the ID provided.
- **Method:** public ActionResult Details(int id)
- **Returns:** A view with the student's details or a 404 error if the student is not found.

#### 3. Create Action (GET):

- **Purpose:** Shows a form to create a new student.
- **Method:** public ActionResult Create()
- **Returns:** The view for creating a new student.

#### 4. Create Action (POST):

- **Purpose:** Handles the form submission to create a new student.
- **Method:** public ActionResult Create(Student student) with the [HttpPost] attribute.
- **Returns:** Redirects to the index action if successful, or returns the form view with error messages if validation fails.

### Summary

- **Action methods** are the core of how controllers handle requests in ASP.NET MVC.
- They process user actions, interact with the model, and determine what to display to the user.
- Action methods can respond to different HTTP methods and return various types of results, such as views or data.

## ASP.NET MVC Model Binding

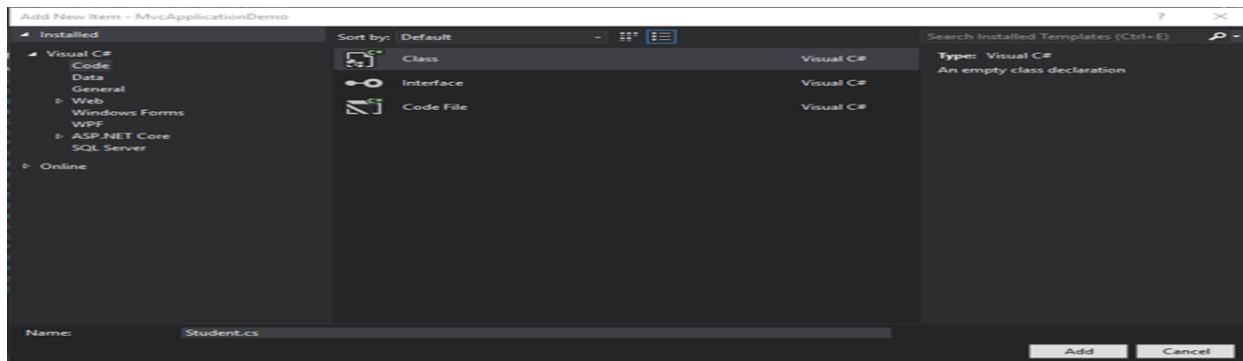
Model binding is a process in which we bind a model to controller and view. It is a simple way to map posted form values to a .NET Framework type and pass the type to an action method as a parameter. It acts as a converter because it can convert HTTP requests into objects that are passed to an action method.

### Example

Here, we are creating an example, in which a simple model is binding with the view and controller. We are creating a Student model that has some properties. These properties will be used to create form fields.

## Create a Model

Right click on the **Model** folder and add a class to create new model.



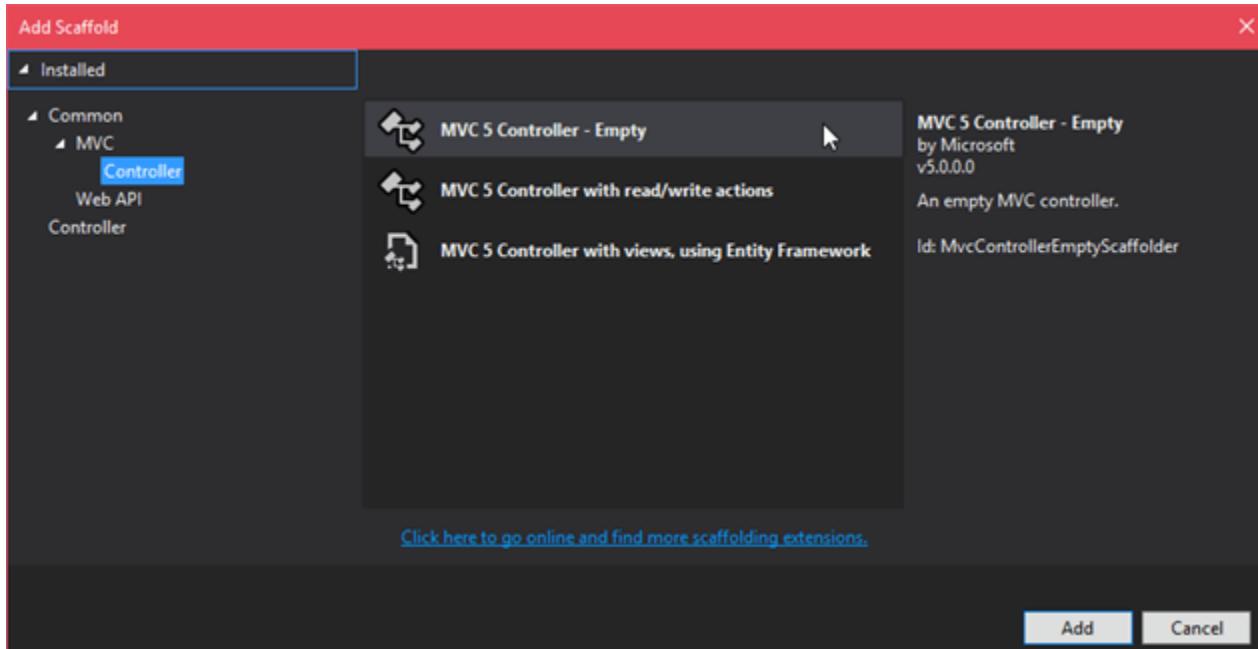
This file contains some default code, but we have added some properties to it. Model looks like this:

// Student.cs

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Web;
5. **namespace** MvcApplicationDemo.Models
6. {
7.   **public class** Student
8. {
9.     **public int** ID { **get; set;** }
10.    **public string** Name { **get; set;** }
11.    **public string** Email { **get; set;** }
12.    **public string** Contact { **get; set;** }
13. }
14. }

## Create a Controller

After creating model, now let's create a controller for this class. Right click on the **Controller** folder and add the controller class.



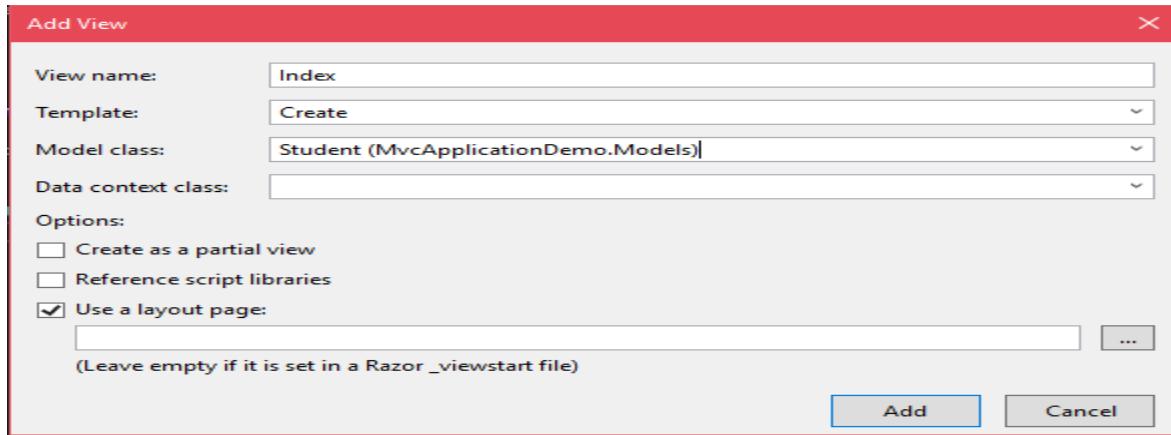
After adding, it provides the following predefined code.

```
// StudentsController.cs

1. using System;
2. using System.Collections.Generic;
3. using System.Linq;
4. using System.Web;
5. using System.Web.Mvc;
6. namespace MvcApplicationDemo.Controllers
7. {
8.     public class StudentsController : Controller
9.     {
10.         // GET: Students
11.         public ActionResult Index()
12.         {
13.             return View();
14.         }
15.     }
16. }
```

## Creating a View

To create view right click within the body of **Index** action method and select **Add View** option, it will pop up for the name of the view and Model to attach with the view.



After adding, it generates an index file within the Students folder that contains the following code.

```
// Index.cshtml
1. @model MvcApplicationDemo.Models.Student
2. @{
3.     ViewBag.Title = "Index";
4. }
5. <h2>Index</h2>
6. @using (Html.BeginForm())
7. {
8.     @Html.AntiForgeryToken()
9.     <div class="form-horizontal">
10.        <h4>Student</h4>
11.        <hr />
12.        @Html.ValidationSummary(true, "", new { @class = "text-danger" })
13.        <div class="form-group">
14.            @Html.LabelFor(model => model.Name, htmlAttributes: new { @class = "control-label col-md-2" })
15.            <div class="col-md-10">
16.                @Html.EditorFor(model => model.Name, new { htmlAttributes = new { @class = "form-control" } })
17.                @Html.ValidationMessageFor(model => model.Name, "", new { @class = "text-danger" })
18.            </div>
```

```

19. </div>
20. <div class="form-group">
21.     @Html.LabelFor(model => model.Email, htmlAttributes: new { @class = "control-label col-md-2" })
22.     <div class="col-md-10">
23.         @Html.EditorFor(model => model.Email, new { htmlAttributes = new { @class = "form-control" } })
24.         @Html.ValidationMessageFor(model => model.Email, "", new { @class = "text-danger" })
25.     </div>
26. </div>
27. <div class="form-group">
28.     @Html.LabelFor(model => model.Contact, htmlAttributes: new { @class = "control-label col-md-2" })
29.     <div class="col-md-10">
30.         @Html.EditorFor(model => model.Contact, new { htmlAttributes = new { @class = "form-control" } })
31.         @Html.ValidationMessageFor(model => model.Contact, "", new { @class = "text-danger" })
32.     </div>
33. </div>
34. <div class="form-group">
35.     <div class="col-md-offset-2 col-md-10">
36.         <input type="submit" value="Create" class="btn btn-default" />
37.     </div>
38. </div>
39. </div>
40. }
41. <div>
42.     @Html.ActionLink("Back to List", "Index")
43. </div>

```

## ASP.NET MVC Input Validation

Validation of user input is necessary task for the application programmer. An application should allow only valid user input so that we get only desired information.

ASP.NET MVC framework provides built-in annotations that we can apply on Model properties. It validates input and display appropriate message to the user.

---

## Commonly used Validation Annotations

Annotations	Description
Required	It is used to make a required field.
DisplayName	It is used to define the text we want to display for the fields.
StringLength	It defines a maximum length for a string field.
Range	It is used to set a maximum and minimum value for a numeric field.
Bind	It lists fields to exclude or include when binding parameter or form values to model properties.
ScaffoldColumn	It allows hiding fields from editor forms.
MaxLength	It is used to set max length for a field.
EmailAddress	It is used to validate email address.
DataType	It is used to specify data type for the field.
RegularExpression	It is used to associate regular expression for a field.

## Example

Let's create an example that will validate input by using the annotations. To create the example, first we are creating a **StudentsController** and then a **Student Model**.

### Controller

```
// StudentsController.cs
```

1. **using System;**
2. **using System.Collections.Generic;**
3. **using System.Linq;**

```
4. using System.Web;
5. using System.Web.Mvc;
6. namespace MvcApplicationDemo.Controllers
7. {
8.     public class StudentsController : Controller
9.     {
10.         // GET: Students
11.         public ActionResult Index()
12.         {
13.             return View();
14.         }
15.     }
16. }
```

---

## Model

*// Student.cs*

```
1. using System.ComponentModel.DataAnnotations;
2.
3. namespace MvcApplicationDemo.Models
4. {
5.     public class Student
6.     {
7.         public int ID { get; set; }
8.         // -- Validating Student Name
9.         [Required(ErrorMessage = "Name is required")]
10.        [MaxLength(12)]
11.        public string Name { get; set; }
12.        // -- Validating Email Address
13.        [Required(ErrorMessage = "Email is required")]
14.        [EmailAddress(ErrorMessage = "Invalid Email Address")]
15.        public string Email { get; set; }
16.        // -- Validating Contact Number
17.        [Required(ErrorMessage = "Contact is required")]
```

```

18.    [DataType(DataType.PhoneNumber)]
19.    [RegularExpression(@"^(\d{3})\d{3}(\d{4})$", ErrorMessage = "Not a valid Phone number")]
20.    public string Contact { get; set; }
21. }
22.

```

---

## View

// Index.cshtml

```

1. @model MvcApplicationDemo.Models.Student
2. @{
3.     ViewBag.Title = "Index";
4. }
5. <h2>Index</h2>
6. @using (Html.BeginForm())
7. {
8.     @Html.AntiForgeryToken()
9.     <div class="form-horizontal">
10.        <h4>Student</h4>
11.        <hr />
12.        @Html.ValidationSummary(true, "", new { @class = "text-danger" })
13.        <div class="form-group">
14.            @Html.LabelFor(model => model.Name, htmlAttributes: new { @class = "control-label col-md-2" })
15.            <div class="col-md-10">
16.                @Html.EditorFor(model => model.Name, new { htmlAttributes = new { @class = "form-control" } })
17.                @Html.ValidationMessageFor(model => model.Name, "", new { @class = "text-danger" })
18.            </div>
19.        </div>
20.        <div class="form-group">
21.            @Html.LabelFor(model => model.Email, htmlAttributes: new { @class = "control-label col-md-2" })
22.            <div class="col-md-10">
23.                @Html.EditorFor(model => model.Email, new { htmlAttributes = new { @class = "form-control" } })
24.                @Html.ValidationMessageFor(model => model.Email, "", new { @class = "text-danger" })

```

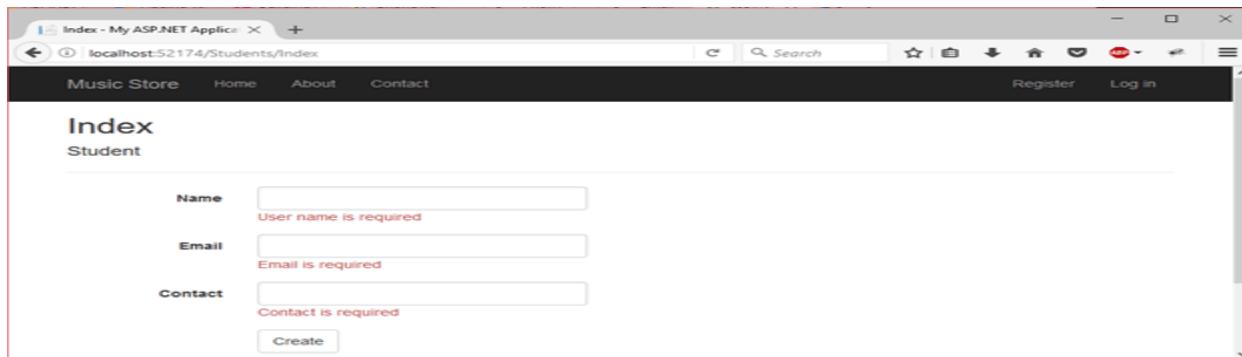
```

25.    </div>
26.    </div>
27.    <div class="form-group">
28.        @Html.LabelFor(model => model.Contact, htmlAttributes: new { @class = "control-label col-md-2" })
29.        <div class="col-md-10">
30.            @Html.EditorFor(model => model.Contact, new { htmlAttributes = new { @class = "form-control" } })
31.            @Html.ValidationMessageFor(model => model.Contact, "", new { @class = "text-danger" })
32.        </div>
33.    </div>
34.    <div class="form-group">
35.        <div class="col-md-offset-2 col-md-10">
36.            <input type="submit" value="Create" class="btn btn-default" />
37.        </div>
38.    </div>
39. </div>
40. }
41. <div>
42.     @Html.ActionLink("Back to List", "Index")
43. </div>
44. @section Scripts {
45.     @Scripts.Render("~/bundles/jqueryval")
46. }

```

Output:

To see the output, right click on the **Index.cshtml** file and select view in browser. This will produce the following result.



As we can see that it validates form fields and display error message to the user. In the below screenshot, we are validating that the entered data is as expected.

The screenshot shows a browser window for 'Index - My ASP.NET Applica...' at 'localhost:52174/Students/Index'. The page title is 'Index' and the sub-page title is 'Student'. There are three input fields: 'Name' (mohsin), 'Email' (mohsin), and 'Contact' (d56d89). The 'Email' field has a red error message 'Invalid Email Address'. The 'Contact' field has a red error message 'Not a valid Phone number'. A 'Create' button is visible at the bottom.

## ASP.NET MVC Entity Framework

ASP.NET MVC and Entity Framework (EF) are often used together to create powerful web applications. Here's a simple explanation of what Entity Framework is and how it integrates with ASP.NET MVC.

### What is Entity Framework?

Entity Framework is an Object-Relational Mapping (ORM) framework that allows developers to work with databases using .NET objects. Instead of writing complex SQL queries, you can use C# or VB.NET to interact with the database, making it easier to manage data.

### Key Features of Entity Framework:

1. **Modeling:** Allows you to define your data model using classes.
2. **Querying:** Use LINQ (Language Integrated Query) to query the database.
3. **Change Tracking:** Automatically tracks changes to your data and helps manage updates.
4. **Migration:** Provides tools to update your database schema as your model changes.

### How ASP.NET MVC Works with Entity Framework

When you combine ASP.NET MVC with Entity Framework, you typically follow these steps:

1. **Create Models:** Define your data models as classes.

```
public class Product
```

```
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```

2. **Set Up DbContext:** Create a context class that derives from DbContext, which acts as a bridge between your models and the database.

```
public class ApplicationDbContext : DbContext
{
    public DbSet<Product> Products { get; set; }
}
```

3. **Add Controller:** Create a controller to handle requests related to your models. Use dependency injection to include the DbContext.

```
public class ProductsController : Controller
{
    private readonly ApplicationDbContext _context;

    public ProductsController()
    {
        _context = new ApplicationDbContext();
    }

    // Action to list products
    public ActionResult Index()
    {
        var products = _context.Products.ToList();
        return View(products);
    }
}
```

4. **Create Views:** Create Razor views to display and interact with your data. For example, the Index.cshtml view might look like this:

```
@model IEnumerable<Product>

<h2>Products</h2>
<table>
<tr>
    <th>Name</th>
    <th>Price</th>
</tr>
```

```

@foreach (var product in Model)
{
    <tr>
        <td>@product.Name</td>
        <td>@product.Price</td>
    </tr>
}
</table>

```

5. **Perform CRUD Operations:** Use Entity Framework to create, read, update, and delete data in your database from your controllers.

## Example of a CRUD Operation

- **Create:**

```

public ActionResult Create(Product product)
{
    if (ModelState.IsValid)
    {
        _context.Products.Add(product);
        _context.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(product);
}

```

- **Read** (as shown in the Index action).
- **Update:**

```

public ActionResult Edit(int id)
{
    var product = _context.Products.Find(id);
    return View(product);
}

[HttpPost]
public ActionResult Edit(Product product)
{
    if (ModelState.IsValid)
    {
        _context.Entry(product).State = EntityState.Modified;
        _context.SaveChanges();
        return RedirectToAction("Index");
    }
}

```

```

    }
    return View(product);
}

```

- **Delete:**

```

public ActionResult Delete(int id)
{
    var product = _context.Products.Find(id);
    return View(product);
}

[HttpPost, ActionName("Delete")]
public ActionResult DeleteConfirmed(int id)
{
    var product = _context.Products.Find(id);
    _context.Products.Remove(product);
    _context.SaveChanges();
    return RedirectToAction("Index");
}

```

## ASP.NET MVC Authentication

Authentication in ASP.NET MVC is the process of verifying the identity of users who want to access your application. Here's a simple explanation of how it works:

### What is Authentication?

Authentication is like checking someone's ID before letting them into a club. In a web application, it ensures that users are who they say they are before allowing them access to certain features or data.

### How ASP.NET MVC Handles Authentication

1. **User Registration:**
  - Users create an account by providing information like their email and password.
  - This information is stored securely in a database.
2. **Login:**
  - Users enter their email and password to log in.
  - The application checks these credentials against the stored information in the database.
  - If the credentials match, the user is granted access.
3. **Authorization:**
  - After logging in, the application determines what the user can do based on their role or permissions (like admin or regular user).
  - For example, an admin can access the admin dashboard, while a regular user cannot.

## Steps to Implement Authentication in ASP.NET MVC

1. **Install Identity:** Use ASP.NET Identity, which is a membership system that provides authentication and user management features.
2. **Create a User Model:** This model represents the user in your application.

```
public class ApplicationUser : IdentityUser
{
    // Additional properties can be added here
}
```

3. **Configure Identity:** In the Startup.cs file, configure services to use Identity.

```
services.AddIdentity<ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();
```

4. **Create Login and Registration Views:** Create forms for users to register and log in.
  - o **Registration:** A form where users enter their email and password.
  - o **Login:** A form where users enter their email and password to log in.
5. **Create Controller Actions:**
  - o **Register:**

```
[HttpPost]
public async Task<IActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
        var result = await _userManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            // Sign in the user
            await _signInManager.SignInAsync(user, isPersistent: false);
            return RedirectToAction("Index", "Home");
        }
    }
    return View(model);
}
```

- o **Login:**

```
[HttpPost]
public async Task<IActionResult> Login(LoginViewModel model)
```

```

{
    if (ModelState.IsValid)
    {
        var result = await _signInManager.PasswordSignInAsync(model.Email, model.Password,
model.RememberMe, lockoutOnFailure: false);
        if (result.Succeeded)
        {
            return RedirectToAction("Index", "Home");
        }
    }
    return View(model);
}

```

6. **Secure Your Pages:** Use the [Authorize] attribute to restrict access to certain actions or controllers.

```
[Authorize]
public class AdminController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
}
```

### Summary

In ASP.NET MVC, authentication is a straightforward process that involves registering users, verifying their identities, and controlling their access to various parts of the application. By using ASP.NET Identity, you can easily implement secure authentication features in your application.

## MVC Routing:

Routing in ASP.NET MVC is the process of mapping a URL to a specific action in a controller. It helps the application understand which code to run when a user requests a particular URL. Here's a simple explanation of how routing works:

### How Routing Works

- URL Pattern:** When a user types a URL in the browser, the routing system checks the URL against predefined patterns to find a match.
- Controller and Action:** Each pattern specifies which controller and which action method should be executed. For example, a URL like /Products/Details/5 would match the Details action in the Products controller with an ID of 5.

## Default Routing

By default, ASP.NET MVC uses a route configuration that looks like this:

```
routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
);
```

- **{controller}**: The name of the controller (e.g., Products).
- **{action}**: The name of the action method (e.g., Details).
- **{id}**: An optional parameter, usually used for identifying specific data (like a product ID).

## Example Routes

1. **Home Page**:
  - URL: /
  - Matches: HomeController.Index
2. **Products List**:
  - URL: /Products
  - Matches: ProductsController.Index
3. **Product Details**:
  - URL: /Products/Details/5
  - Matches: ProductsController.Details(5)

## Custom Routes

You can also define custom routes for specific URLs. For example:

```
routes.MapRoute(
    name: "About",
    url: "AboutUs",
    defaults: new { controller = "Home", action = "About" }
);
```

- This route means that when a user visits /AboutUs, it will call the About action in the HomeController.

## Route Constraints

You can add conditions to your routes to make them more specific. For example, you can require that the id parameter must be a number:

```
routes.MapRoute(
```

```

name: "Product",
url: "Products/Details/{id}",
defaults: new { controller = "Products", action = "Details" },
constraints: new { id = @"\d+" } // id must be a number
);

```

### Summary

Routing in ASP.NET MVC is essential for directing incoming requests to the correct controller actions based on the URL. It helps create user-friendly URLs and organize your application better.

## ASP.NET Razor Introduction

Razor is a standard markup syntax that allows us to embed server code into the web pages. It uses its own syntax and keywords to generate view.

If there is server code in the web page, server executes that code first then send response to the browser. It allows us to perform logical tasks in the view page. We can create expressions, loops and variables in the view page.

It has simplified syntax which is easy to learn and code. This file extension is **.cshtml**.

### @ Character

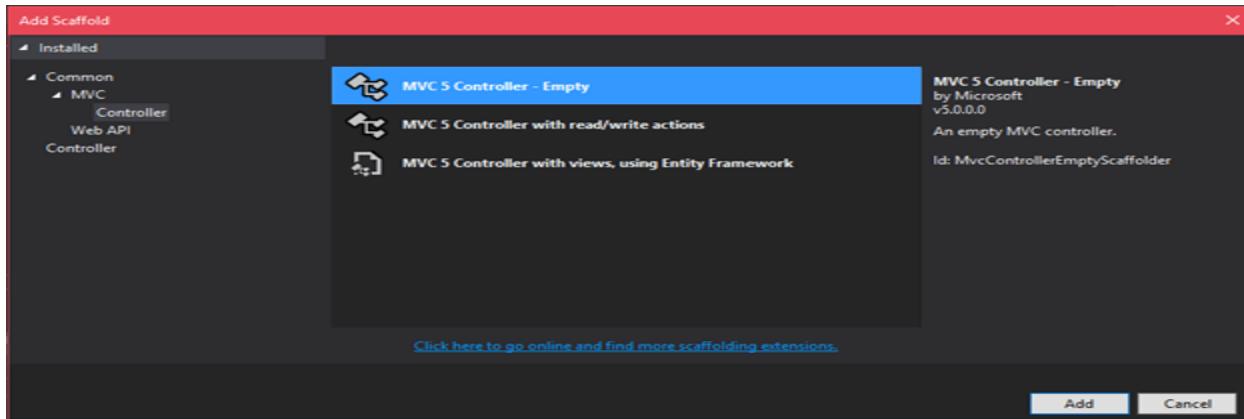
Razor uses this character to transit from HTML to C#. When @ symbol is used with razor syntax, it transits into Razor specific markup, otherwise it transitions into plain C#. We used it, to start single line expression, single statement block or multi-statement block.

### Razor Keywords

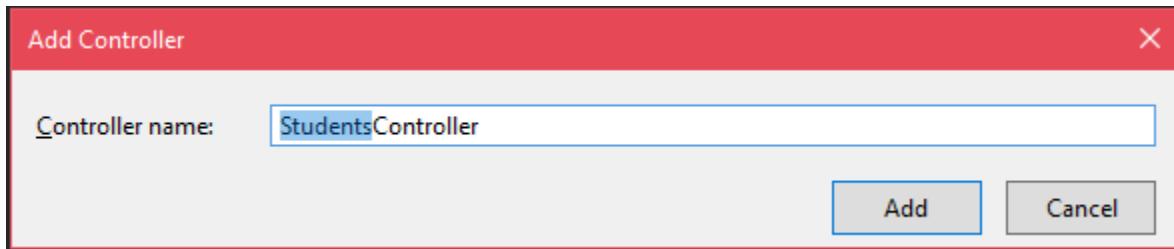
- o functions
- o inherits
- o model
- o section
- o helper (Not supported by ASP.NET Core.)

### Example

Let's create a view that has Razor syntax. Right click on the **Controller** folder and select **add->controller**, it will prompt the following dialog.



Provide a name to the controller.



Click on add button, this will create a controller and adding the following code.

`//StudentsController.cs`

1. `using System;`
2. `using System.Collections.Generic;`
3. `using System.Linq;`
4. `using System.Web;`
5. `using System.Web.Mvc;`
6. `namespace RazorViewExample.Controllers`
7. `{`
8. `public class StudentsController : Controller`
9. `{`
10. `// GET: Students`
11. `public ActionResult Index()`
12. `{`
13. `return View();`
14. `}`
15. `}`

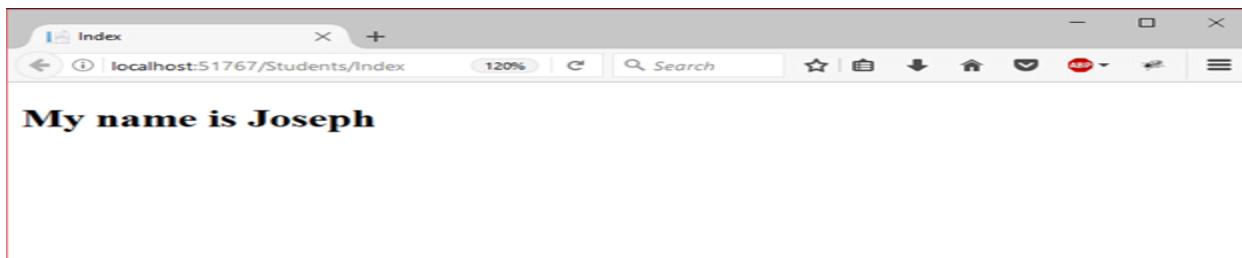
16. }

// Index.cshtml

```

1. @{
2.     Layout = null;
3.     var name = "Joseph";
4. }
5. <!DOCTYPE html>
6. <html>
7. <head>
8.     <meta name="viewport" content="width=device-width" />
9.     <title>Index</title>
10. </head>
11. <body>
12.     <h2>My name is @name </h2>
13. </body>
14. </html>
```

It produces the following output to the browser.



## ASP.NET Razor Code Expressions

Razor syntax is widely used with C# programming language. To write C# code into a view use @ (at) sign to start Razor syntax. We can use it to write single line expression or multiline code block. Let's see how we can use C# code in the view page.

The following example demonstrate code expression.

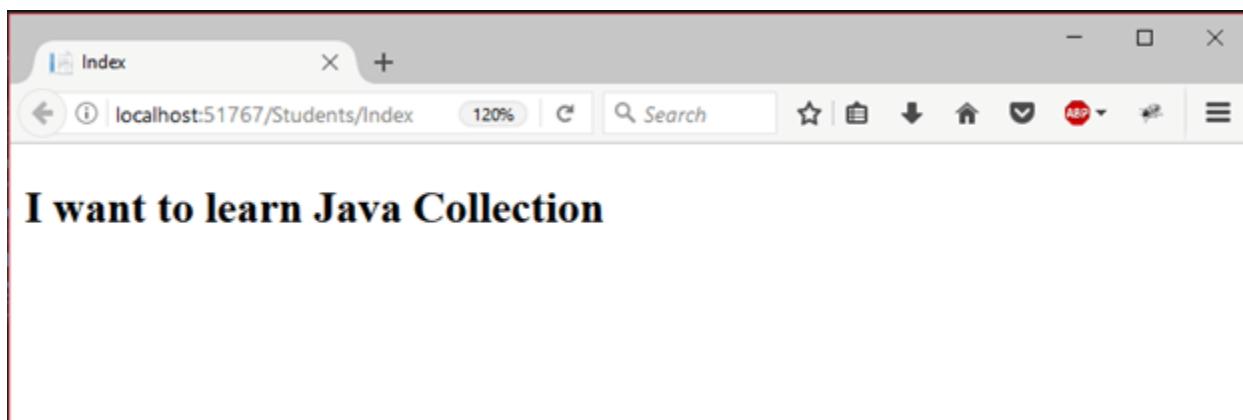
// Index.cshtml

1. @{

```
2. Layout = null;
3. var coursename = "Java Collection";
4. }
5. <!DOCTYPE html>
6. <html>
7. <head>
8.   <meta name="viewport" content="width=device-width" />
9.   <title>Index</title>
10. </head>
11. <body>
12.   <h2>I want to learn @coursename </h2>
13. </body>
14. </html>
```

Produce the following output.

Output:



## Implicit Razor Expressions

Implicit Razor expression starts with @ (at) character followed by C# code. The following example demonstrates about implicit expressions.

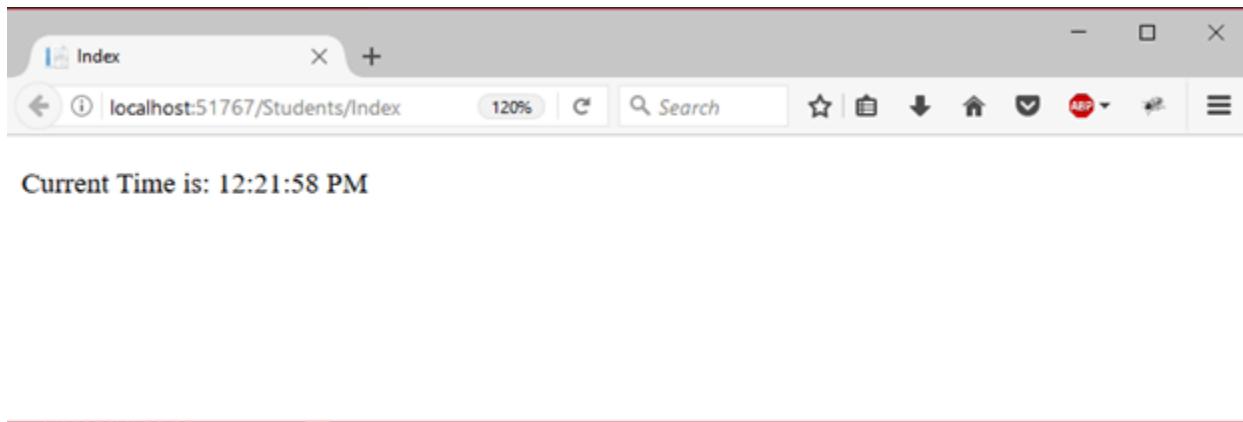
// Index.cshtml

```
1. @{
2.   Layout = null;
```

```
3. }
4. <!DOCTYPE html>
5. <html>
6. <head>
7.   <meta name="viewport" content="width=device-width" />
8.   <title>Index</title>
9. </head>
10. <body>
11.   <p>Current Time is: @DateTime.Now.ToString("T")</p>
12. </body>
13. </html>
```

It produces the following output.

Output:



## Explicit Razor Expressions

Explicit Razor expression consists of @ (at) character with balanced parenthesis. In the following example, expression is enclosed with parenthesis to execute safely. It will throw an error if it is not enclosed with parenthesis.

We can use explicit expression to concatenate text with an expression.

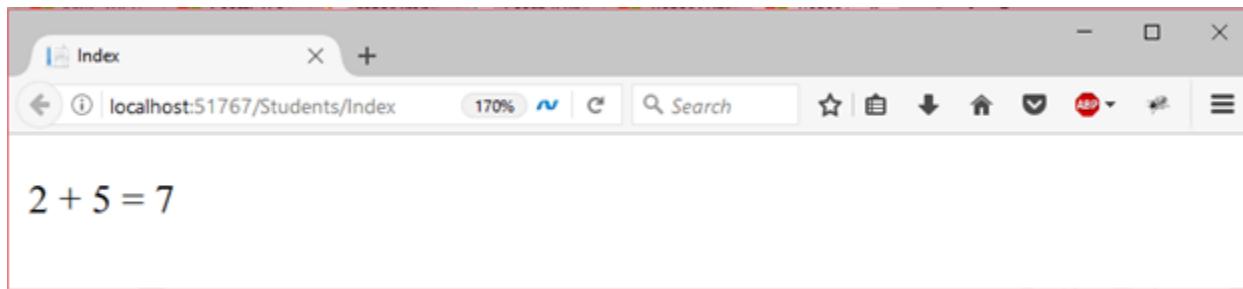
```
// Index.cshtml
```

```
1. @{
2.   Layout = null;
```

```
3. }
4. <!DOCTYPE html>
5. <html>
6. <head>
7.   <meta name="viewport" content="width=device-width" />
8.   <title>Index</title>
9. </head>
10. <body>
11.   <p>2 + 5 = @(2+5)</p>
12. </body>
13. </html>
```

It produces the following output.

Output:



## Razor Expression Encoding

Razor provides expression encoding to avoid malicious code and security risks. In case, if user enters a malicious script as input, razor engine encode the script and render as HTML output.

Here, we are **not using** razor syntax in view page.

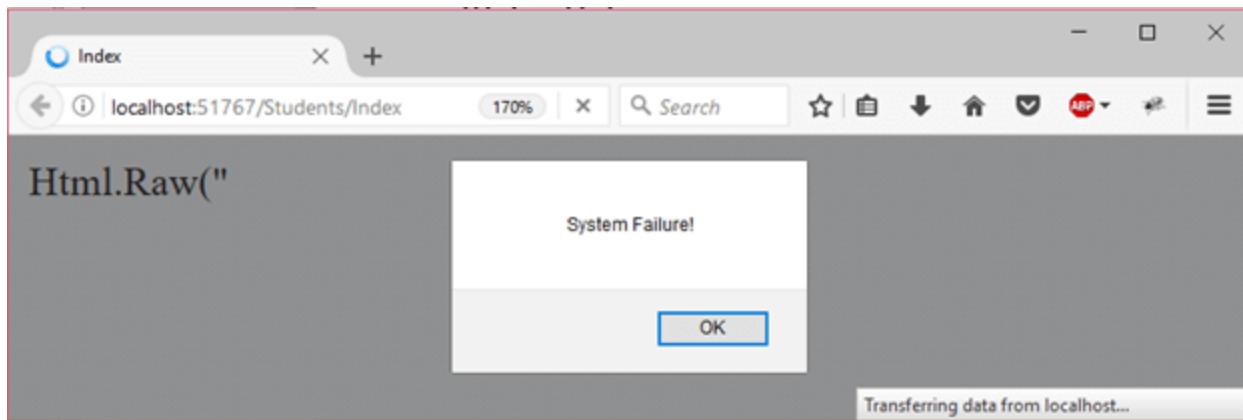
```
// Index.cshtml
```

```
1. @{
2.   Layout = null;
3. }
4. <!DOCTYPE html>
5. <html>
```

```
6. <head>
7.   <meta name="viewport" content="width=device-width" />
8.   <title>Index</title>
9. </head>
10. <body>
11.   Html.Raw("<script>alert('System Failure!')</script>")
12. </body>
13. </html>
```

It produces the following output.

Output:



---

In the following example, we are encoding JavaScript script.

```
// Index.cshtml
```

```
1. @{
2.   Layout = null;
3. }
4. <!DOCTYPE html>
5. <html>
6. <head>
7.   <meta name="viewport" content="width=device-width" />
8.   <title>Index</title>
9. </head>
```

```
10. <body>
11.   @("<script>alert('this is alert box')</script>")
12. </body>
13. </html>
```

Now, it produces the following output.

Output:



This time razor engine encodes the script and return as a simple HTML string.

## ASP.NET Razor Code Blocks

Code block is used to enclose C# code statements. It starts with @ (at) character and is enclosed by {} (curly braces). Unlike expressions, C# code inside code blocks is not rendered. The default language in a code block is C#, but we can transit back to HTML. HTML within a code block will be rendered as HTML.

---

### Example

```
// Index.cshtml
```

```
1. @{
2.   Layout = null;
3.   var name = "John";
4. }
5. <!DOCTYPE html>
6. <html>
```

```
7. <head>
8.   <meta name="viewport" content="width=device-width" />
9.   <title>Index</title>
10. </head>
11. <body>
12.   <h2>My name is: @name </h2>
13. </body>
14. </html>
```

It produces the following output.



## Implicit transitions

C# is default language in Razor code block. HTML written within code block is rendered as HTML, this is called implicit transition. Razor code blocks implicit transitions HTML code and render to the view page.

In the following code, HTML is written and it executes without error.

```
// Index.cshtml
```

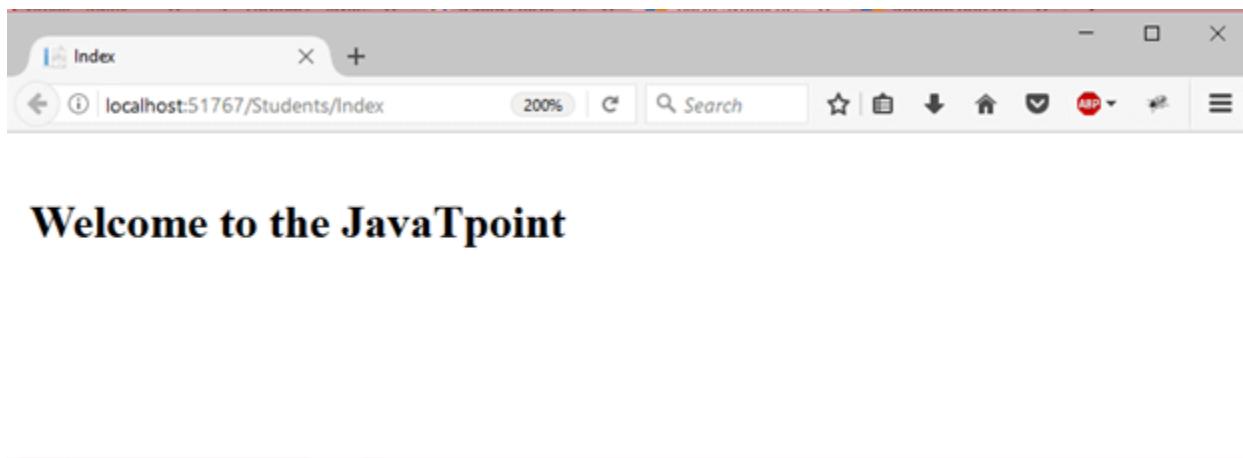
```
1. @{
2.   Layout = null;
3. }
4. <!DOCTYPE html>
5. <html>
6. <head>
7.   <meta name="viewport" content="width=device-width" />
8.   <title>Index</title>
9. </head>
```

```

10. <body>
11. @{
12.   var name = "JavaTpoint";
13.   <h4>Welcome to the @name </h4>
14. }
15. </body>
16. </html>

```

It produces the following output.



### Explicit delimited transition

Sometimes, when we define a sub-section of a code block that should render HTML, surround the characters to be rendered with the Razor `<text>` tag.

It is mandatory to use `<text>` tag. Otherwise, it throws a compile time error. See the following code.

```
// Index.cshtml

1. @{
2.   Layout = null;
3. }
4. <!DOCTYPE html>
5. <html>
6. <head>
7.   <meta name="viewport" content="width=device-width" />
```

```

8. <title>Index</title>
9. </head>
10. <body>
11. @for (var i = 0; i < 5; i++)
12. {
13.   <text>i= @i </text> <br/>
14. }
15. </body>
16. </html>

```

It produces the following output.



## **Data annotations:**

**Data Annotations** are used to validate user input by applying validation rules defined in your model. These annotations help ensure that the data users enter into forms is correct before it's processed or saved to the database.

### **How Data Annotations Work in Razor**

1. **Define Annotations in the Model:**
  - You add Data Annotation attributes to your model properties.
  - These attributes specify validation rules like "required," "string length," or "range."
2. **Automatic Validation in Razor Forms:**
  - When a form is submitted, Razor automatically checks the input against the validation rules defined by the Data Annotations.
  - If the input doesn't meet the rules, error messages are displayed next to the form fields.

### **Step-by-Step Example**

Let's go through a simple example:

## Step 1: Create a Model with Data Annotations

Here's a basic model with some Data Annotations:

```
public class User
{
    [Required(ErrorMessage = "Name is required.")]
    [StringLength(50, ErrorMessage = "Name cannot be longer than 50 characters.")]
    public string Name { get; set; }

    [Required(ErrorMessage = "Email is required.")]
    [EmailAddress(ErrorMessage = "Invalid email address.")]
    public string Email { get; set; }

    [Range(18, 99, ErrorMessage = "Age must be between 18 and 99.")]
    public int Age { get; set; }
}
```

- **[Required]**: Ensures the field is not empty.
- **[StringLength]**: Limits the length of the Name to 50 characters.
- **[EmailAddress]**: Checks that the Email is in a valid format.
- **[Range]**: Ensures that Age is between 18 and 99.

## Step 2: Create a Razor View for the Form

Now, create a form in your Razor view to collect user data:

```
@model YourNamespace.Models.User

@using (Html.BeginForm())
{
    <div>
        @Html.LabelFor(m => m.Name)
        @Html.TextBoxFor(m => m.Name)
        @Html.ValidationMessageFor(m => m.Name)
    </div>

    <div>
        @Html.LabelFor(m => m.Email)
        @Html.TextBoxFor(m => m.Email)
        @Html.ValidationMessageFor(m => m.Email)
    </div>

    <div>
```

```

@Html.LabelFor(m => m.Age)
@Html.TextBoxFor(m => m.Age)
@Html.ValidationMessageFor(m => m.Age)
</div>

<button type="submit">Submit</button>
}

```

```

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}

```

- **@Html.LabelFor**: Renders a label for the form field.
- **@Html.TextBoxFor**: Renders a text box for user input.
- **@Html.ValidationMessageFor**: Displays validation error messages if the input doesn't meet the rules.

### Step 3: Handle Validation in the Controller

In your controller, you handle the form submission and check for validation:

```

[HttpPost]
public ActionResult Register(User model)
{
    if (ModelState.IsValid)
    {
        // Save the data or perform other actions
        return RedirectToAction("Success");
    }

    // If the model is not valid, return the view with the same model to show errors
    return View(model);
}

```

- **ModelState.IsValid**: Checks if all validation rules are satisfied.
- If validation fails, the form is redisplayed with error messages next to the invalid fields.

### Summary

In Razor, Data Annotations make it easy to validate user input by defining rules in your model. The Razor view then automatically checks these rules when the form is submitted and displays error messages if something is wrong. This ensures that your application handles data safely and correctly.

## Client and server side validation:

**Client-side validation** and **server-side validation** are both important for ensuring that the data submitted through forms in your web application is valid and secure. Here's a simple explanation of each:

## Client-Side Validation

**Client-side validation** occurs in the user's browser before the data is sent to the server. It helps improve user experience by providing instant feedback when the user fills out a form.

### How It Works:

- Validation rules are applied directly in the user's browser using JavaScript (or a library like jQuery).
- When a user submits a form, the browser checks the input against these rules.
- If there's an error (e.g., a required field is empty), the user sees an error message immediately, and the form isn't submitted.

### Example:

- If you have a required field for an email address, the browser will check if the user has entered a valid email format before allowing the form to be sent.

### Benefits:

- **Instant Feedback:** Users know right away if they've made a mistake.
- **Reduced Server Load:** Invalid data is caught before it reaches the server, reducing unnecessary processing.

### Example in Razor:

```
@model YourNamespace.Models.User

@using (Html.BeginForm())
{
    <div>
        @Html.LabelFor(m => m.Name)
        @Html.TextBoxFor(m => m.Name)
        @Html.ValidationMessageFor(m => m.Name)
    </div>

    <div>
        @Html.LabelFor(m => m.Email)
        @Html.TextBoxFor(m => m.Email)
        @Html.ValidationMessageFor(m => m.Email)
    </div>
```

```

<div>
    @Html.LabelFor(m => m.Age)
    @Html.TextBoxFor(m => m.Age)
    @Html.ValidationMessageFor(m => m.Age)
</div>

<button type="submit">Submit</button>
}

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval") <!-- jQuery validation library -->
}

```

In this Razor view:

- The validation happens instantly on the client side using jQuery.

## Server-Side Validation

**Server-side validation** happens on the server after the data is sent from the browser. It acts as the final safeguard to ensure that the data is correct and secure before it is processed or stored.

How It Works:

- When a form is submitted, the data is sent to the server.
- The server checks the data against the validation rules defined in your model.
- If the data is invalid, the server sends the form back to the user with error messages.

Example:

- If a user disables JavaScript in their browser, client-side validation won't work. Server-side validation ensures that even in such cases, invalid data won't be processed.

Benefits:

- **Security:** Protects your application from malicious input, like SQL injection or cross-site scripting (XSS).
- **Reliability:** Ensures that data is validated even if the client-side validation is bypassed.

Example in ASP.NET MVC:

```

[HttpPost]
public ActionResult Register(User model)
{
    if (ModelState.IsValid)

```

```

{
    // Save the data or perform other actions
    return RedirectToAction("Success");
}

// If the model is not valid, return the view with the same model to show errors
return View(model);
}

```

In this controller action:

- The server checks if the model is valid using ModelState.IsValid. If not, the view is returned with error messages.

## Combining Client-Side and Server-Side Validation

**Best Practice:** Always use both client-side and server-side validation.

- **Client-Side:** Improves user experience with instant feedback.
- **Server-Side:** Ensures security and correctness by validating the data once it reaches the server.

By using both, you ensure that your application is user-friendly, secure, and reliable.

## MVC membership:

**ASP.NET MVC Membership** refers to the system that handles authentication and authorization in ASP.NET MVC applications. It provides a framework for managing user accounts, passwords, roles, and permissions, allowing you to control who can access your web application and what they can do within it.

### Key Concepts of MVC Membership

1. **Authentication:** Verifies the identity of a user (e.g., by checking their username and password).
2. **Authorization:** Determines what an authenticated user is allowed to do (e.g., which pages they can access).
3. **Membership Provider:** The component that interacts with the database to manage user information, such as creating users, validating credentials, and managing roles.

## ASP.NET Identity

In modern ASP.NET MVC applications, **ASP.NET Identity** is commonly used for managing membership. ASP.NET Identity replaces the older Membership system and offers more flexibility and features.

## Features of ASP.NET Identity:

- **User Management:** Create, update, and delete user accounts.
- **Role Management:** Assign roles to users and manage role-based access control.
- **Claims-Based Authentication:** Manage user permissions and data using claims.
- **OAuth and Social Logins:** Supports external login providers like Google, Facebook, and Microsoft.
- **Two-Factor Authentication:** Enhances security by requiring a second form of verification.

## Setting Up Membership with ASP.NET Identity

### Step 1: Install ASP.NET Identity

If you're starting a new project, ASP.NET Identity is usually included by default in templates like "Individual User Accounts." If you're adding it to an existing project, you might need to install the necessary NuGet packages:

Install-Package Microsoft.AspNet.Identity.EntityFramework

Install-Package Microsoft.AspNet.Identity.Owin

### Step 2: Configure Identity

ASP.NET Identity uses UserManager, SignInManager, and RoleManager classes to manage users and roles. These are typically set up in the Startup.cs file.

```
public void ConfigureServices(IServiceCollection services)
{
    // Configure Identity services
    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationContext>()
        .AddDefaultTokenProviders();

    // Other service configurations
}
```

- **ApplicationUser:** A custom user class that usually inherits from IdentityUser.
- **ApplicationContext:** The Entity Framework context that manages your database.

### Step 3: Create Users and Roles

You can create users and roles in your controller or during application setup.

```
public async Task<ActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
```

```

var result = await UserManager.CreateAsync(user, model.Password);
if (result.Succeeded)
{
    await SignInManager.SignInAsync(user, isPersistent: false, rememberBrowser: false);
    return RedirectToAction("Index", "Home");
}
AddErrors(result);
}

return View(model);
}

```

#### **Step 4: Authentication and Authorization**

Use [Authorize] attributes to protect your controllers or actions:

```

[Authorize(Roles = "Admin")]
public ActionResult AdminOnly()
{
    return View();
}

```

- This action will only be accessible to users in the "Admin" role.

#### **Summary**

- **ASP.NET MVC Membership:** Manages user authentication and authorization.
- **ASP.NET Identity:** The modern way to handle membership, offering features like role management, social logins, and two-factor authentication.
- **Setup:** Involves configuring services, managing users and roles, and applying authentication and authorization to your application.

This system allows you to build secure web applications with user management features, ensuring that only authorized users can access specific parts of your application.

## **Service Oriented Architecture (SOA)**

A Service-Oriented Architecture or SOA is a design pattern which is designed to build distributed systems that deliver services to other applications through the protocol. It is only a concept and not limited to any programming language or platform.

## What is Service?

A service is a well-defined, self-contained function that represents a unit of functionality. A service can exchange information from another service. It is not dependent on the state of another service. It uses a loosely coupled, message-based communication model to communicate with applications and other services.

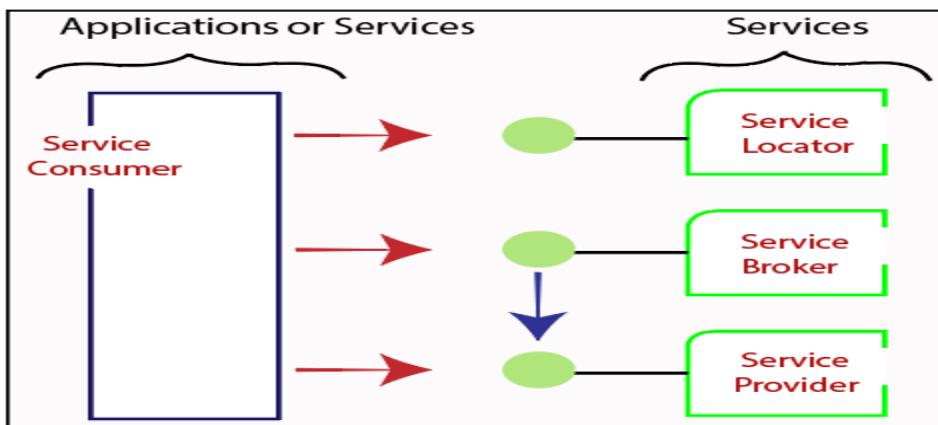
## Service Connections

The figure given below illustrates the service-oriented architecture. Service consumer sends a service request to the service provider, and the service provider sends the service response to the service consumer. The service connection is understandable to both the service consumer and service provider.



## Service-Oriented Terminologies

Let's see some important service-oriented terminologies:



- **Services** - The services are the logical entities defined by one or more published interfaces.
- **Service provider** - It is a software entity that implements a service specification.
- **Service consumer** - It can be called as a requestor or client that calls a service provider. A service consumer can be another service or an end-user application.

- **Service locator** - It is a service provider that acts as a registry. It is responsible for examining service provider interfaces and service locations.
- **Service broker** - It is a service provider that pass service requests to one or more additional service providers.

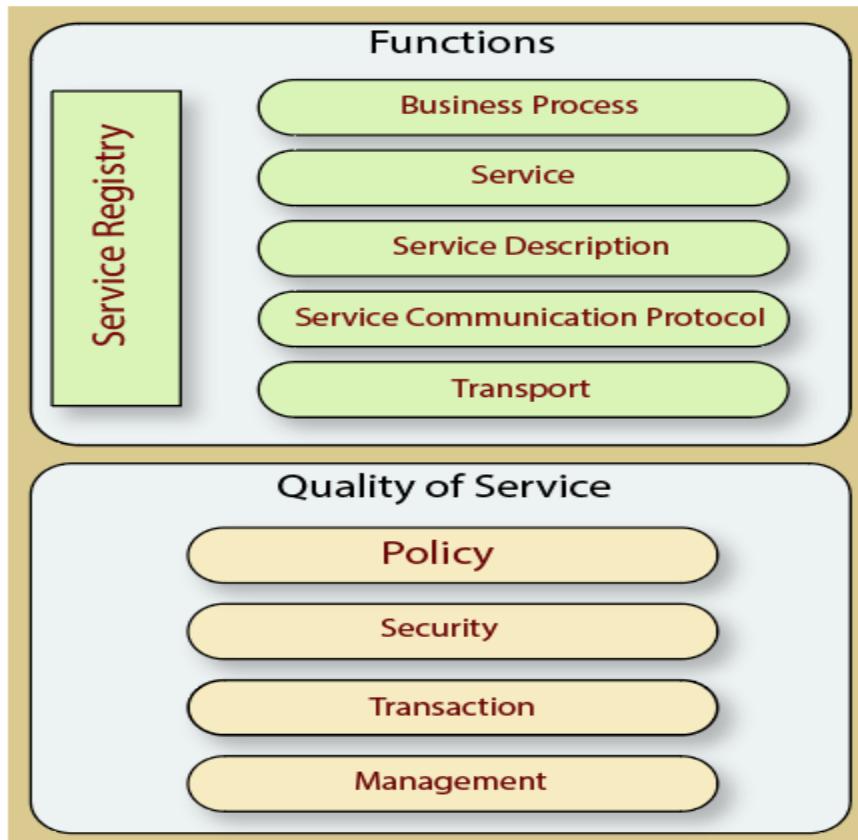
### Characteristics of SOA

The services have the following characteristics:

- They are loosely coupled.
- They support interoperability.
- They are location-transparent
- They are self-contained.

### Components of service-oriented architecture

The service-oriented architecture stack can be categorized into two parts - functional aspects and quality of service aspects.



## Functional aspects

The functional aspect contains:

- Transport - It transports the service requests from the service consumer to the service provider and service responses from the service provider to the service consumer.
- Service Communication Protocol - It allows the service provider and the service consumer to communicate with each other.
- Service Description - It describes the service and data required to invoke it.
- Service - It is an actual service.
- Business Process - It represents the group of services called in a particular sequence associated with the particular rules to meet the business requirements.
- Service Registry - It contains the description of data which is used by service providers to publish their services.

## Quality of Service aspects

The quality of service aspects contains:

- Policy - It represents the set of protocols according to which a service provider make and provide the services to consumers.
- Security - It represents the set of protocols required for identification and authorization.
- Transaction - It provides the surety of consistent result. This means, if we use the group of services to complete a business function, either all must complete or none of the complete.
- Management - It defines the set of attributes used to manage the services.

## Advantages of SOA

SOA has the following advantages:

- Easy to integrate - In a service-oriented architecture, the integration is a service specification that provides implementation transparency.
- Manage Complexity - Due to service specification, the complexities get isolated, and integration becomes more manageable.
- Platform Independence - The services are platform-independent as they can communicate with other applications through a common language.

- Loose coupling - It facilitates to implement services without impacting other applications or services.
- Parallel Development - As SOA follows layer-based architecture, it provides parallel development.
- Available - The SOA services are easily available to any requester.
- Reliable - As services are small in size, it is easier to test and debug them.

## **SOAP:**

**SOAP (Simple Object Access Protocol)** is a protocol used for exchanging structured information in web services. It relies on XML (eXtensible Markup Language) for its message format and usually operates over HTTP or HTTPS. SOAP is designed to allow programs running on different operating systems and programming languages to communicate with each other.

### **Key Features of SOAP**

1. **XML-Based:** SOAP messages are formatted in XML, making them platform- and language-independent.
2. **Protocol Independence:** While it commonly uses HTTP, SOAP can also operate over other protocols such as SMTP, TCP, or JMS.
3. **WSDL (Web Services Description Language):** SOAP services use WSDL files to describe the service, including available methods, input/output formats, and data types.
4. **Extensibility:** SOAP supports extensions such as WS-Security for security features like authentication and encryption.
5. **Reliability:** SOAP provides features to ensure message delivery, such as acknowledgments and retransmissions.

### **Basic Structure of a SOAP Message**

A SOAP message consists of the following parts:

1. **Envelope:** The root element that defines the XML document as a SOAP message.
2. **Header (optional):** Contains metadata about the message, such as authentication or transaction information.
3. **Body:** Contains the main message or data, including method calls and return values.
4. **Fault (optional):** Provides error and status information about the SOAP message.

### **Example of a SOAP Message**

Here's a simple example of a SOAP message for a service that adds two numbers:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Header>
<m:Transaction xmlns:m="http://www.example.org/transaction">
  12345
```

```

</m:Transaction>
</soap:Header>
<soap:Body>
  <m:Add xmlns:m="http://www.example.org/math">
    <m:Number1>5</m:Number1>
    <m:Number2>3</m:Number2>
  </m:Add>
</soap:Body>
</soap:Envelope>

```

## Creating a SOAP Web Service in C#

To create a SOAP web service in C#, you can use **Windows Communication Foundation (WCF)**, which provides a framework for building SOAP services.

### Step 1: Create a WCF Service

1. Open Visual Studio and create a new project.
2. Select **WCF Service Application**.

### Step 2: Define the Service Contract

Define the service interface that specifies the methods the service will expose. For example, a simple math service:

```

[ServiceContract]
public interface IMathService
{
  [OperationContract]
  int Add(int number1, int number2);
}

```

### Step 3: Implement the Service

Create a class that implements the service contract:

```

public class MathService : IMathService
{
  public int Add(int number1, int number2)
  {
    return number1 + number2;
  }
}

```

### Step 4: Configure the Service

In the Web.config file, you'll define the service endpoint and binding:

```

<system.serviceModel>
  <services>
    <service name="YourNamespace.MathService">
      <endpoint address="" binding="basicHttpBinding" contract="YourNamespace.IMathService" />
      <host>
        <baseAddresses>
          <add baseAddress="http://localhost:8733/Design_Time_Addresses/YourNamespace/MathService/" />
        </baseAddresses>
      </host>
    </service>
  </services>
</system.serviceModel>

```

### Step 5: Host and Test the Service

- Run your project, and the service will be hosted on a specified URL (e.g., [http://localhost:8733/...](http://localhost:8733/)).
- You can use a tool like **WCF Test Client** or **Postman** to test your SOAP service by sending a SOAP request.

### Summary

- **SOAP** is a protocol for exchanging structured information in web services.
- It uses XML and operates over various protocols, primarily HTTP.
- SOAP services are defined using WSDL files and can be created in C# using **WCF**.
- SOAP provides features like extensibility and reliability, making it suitable for enterprise-level applications.

## WSDL:

**WSDL (Web Services Description Language)** is an XML-based language used to describe the functionalities offered by a web service. It provides a standard way to define the interface, including the available methods, input and output parameters, data types, and how to access the service.

### Key Components of WSDL

A WSDL document typically consists of the following major parts:

1. **Definitions:** The root element that defines the entire WSDL document. It includes namespaces and overall structure.
2. **Types:** This section defines the data types used by the web service. It often uses XML Schema to specify complex types, elements, and data structures.
3. **Messages:** Messages describe the data that is being exchanged between the client and the service. Each message can consist of one or more parts, which represent the parameters of the method.
4. **Port Types:** This section defines a set of operations (methods) that the service can perform. Each operation specifies the input and output messages.

5. **Bindings:** Bindings define the communication protocols and message formats used by the service. For example, it specifies whether to use SOAP over HTTP.
6. **Service:** This section defines the endpoint (URL) where the web service can be accessed. It includes the service name and the address (URL) for the service.

### Example of a WSDL Document

Here's a simple example of a WSDL document for a web service that performs addition:

```

<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.example.org/math"
  name="MathService"
  targetNamespace="http://www.example.org/math">

  <types>
    <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.example.org/math">
      <xsd:element name="AddRequest">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="Number1" type="xsd:int"/>
            <xsd:element name="Number2" type="xsd:int"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="AddResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="Result" type="xsd:int"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </types>

  <message name="AddRequest">
    <part name="parameters" element="tns:AddRequest"/>
  </message>
  <message name="AddResponse">
    <part name="parameters" element="tns:AddResponse"/>
  </message>

  <portType name="MathServicePortType">
```

```

<operation name="Add">
  <input message="tns:AddRequest"/>
  <output message="tns:AddResponse"/>
</operation>
</portType>

<binding name="MathServiceBinding" type="tns:MathServicePortType">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="Add">
    <soap:operation soapAction="http://www.example.org/math/Add"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>

<service name="MathService">
  <port name="MathServicePort" binding="tns:MathServiceBinding">
    <soap:address
location="http://localhost:8733/Design_Time_Addresses/YourNamespace/MathService/" />
  </port>
</service>
</definitions>

```

### **Explanation of the Example**

- **Definitions:** The root element defines the WSDL and includes namespaces.
- **Types:** It defines the complex types used in the request and response.
- **Messages:** Each operation (like Add) has associated messages for input and output.
- **Port Types:** It defines the operations available in the service.
- **Bindings:** Specifies the protocol (SOAP over HTTP) used by the service.
- **Service:** Provides the endpoint where the service can be accessed.

### **How to Use WSDL**

1. **Client Generation:** Tools can automatically generate client code based on a WSDL document. This allows developers to easily interact with the web service without manually writing all the communication code.
2. **Service Discovery:** Clients can use WSDL to understand what operations a service offers and how to call them.
3. **Interoperability:** WSDL helps different systems (written in different programming languages) to communicate effectively, as they follow the standards defined in the WSDL.

## Summary

- **WSDL** is an XML format for describing web services and their interfaces.
- It includes definitions for data types, messages, operations, bindings, and service endpoints.
- WSDL enables automatic client code generation and helps ensure interoperability between different systems.

## Service contract:

A **Service Contract** in the context of web services, particularly in **Windows Communication Foundation (WCF)**, is an essential part of defining how a web service will operate. It specifies the methods (operations) that the service will expose, the data types that will be used, and the protocols that will be supported for communication. Essentially, it acts as a contract between the service and its clients.

### Key Elements of a Service Contract

1. **ServiceContract Attribute:** This attribute is used to define an interface as a service contract. It indicates that the interface will expose operations for clients to call.
2. **OperationContract Attribute:** This attribute is applied to methods within the service contract to indicate that they are operations that can be called by clients. Each operation may have additional attributes for configuration.
3. **Data Types:** The data types used as parameters and return values for the operations must be serializable (convertible to and from XML or JSON).

### Example of a Service Contract

Here's a simple example of a service contract for a calculator service in C#:

```
using System.ServiceModel;

[ServiceContract]
public interface ICalculatorService
{
    [OperationContract]
    int Add(int number1, int number2);

    [OperationContract]
    int Subtract(int number1, int number2);

    [OperationContract]
    int Multiply(int number1, int number2);

    [OperationContract]
```

```
int Divide(int number1, int number2);
}
```

### Explanation of the Example

- **ServiceContract:** The ICalculatorService interface is decorated with the [ServiceContract] attribute, indicating that it defines a service contract.
- **OperationContract:** Each method (like Add, Subtract, Multiply, and Divide) is decorated with the [OperationContract] attribute, making it available for clients to call.
- **Method Signatures:** Each operation takes two integers as input and returns an integer as output. This is the expected behavior for a simple calculator service.

### Using the Service Contract

Once you define a service contract, you need to implement it in a service class. Here's how you might implement the ICalculatorService interface:

```
public class CalculatorService : ICalculatorService
{
    public int Add(int number1, int number2)
    {
        return number1 + number2;
    }

    public int Subtract(int number1, int number2)
    {
        return number1 - number2;
    }

    public int Multiply(int number1, int number2)
    {
        return number1 * number2;
    }

    public int Divide(int number1, int number2)
    {
        if (number2 == 0) throw new DivideByZeroException("Cannot divide by zero.");
        return number1 / number2;
    }
}
```

### Key Points About Service Contracts

1. **Loose Coupling:** Service contracts promote loose coupling between the client and service implementation. Clients only depend on the contract rather than the specific implementation.

2. **Interoperability:** Service contracts can be exposed using standards like SOAP and WSDL, ensuring that clients written in different languages can interact with the service.
3. **Versioning:** You can evolve your service contract over time without breaking existing clients by introducing new operations and deprecating old ones.

## Summary

- A **Service Contract** defines the operations that a web service exposes to clients.
- It is created using the [ServiceContract] and [OperationContract] attributes in C#.
- It allows for loose coupling, interoperability, and easier versioning of services.

## Data contract:

A **Data Contract** in the context of Windows Communication Foundation (WCF) defines the data types that can be used in the service's messages. It specifies how data is serialized (converted to a format suitable for transmission) and deserialized (converted back to an object) when it is exchanged between a client and a service.

### Key Elements of a Data Contract

1. **DataContract Attribute:** This attribute is used to mark a class or struct as a data contract. It indicates that the class is intended for serialization in WCF.
2. **DataMember Attribute:** This attribute is applied to the fields or properties of a data contract to indicate that they should be serialized. You can control the order of serialization and whether the member is required.
3. **Serialization:** Data contracts allow you to define how the data will be serialized into XML or JSON format for transmission.

### Example of a Data Contract

Here's a simple example of a data contract for a Product class in C#:

using System.Runtime.Serialization;

```
[DataContract]
public class Product
{
    [DataMember(Order = 1)]
    public int Id { get; set; }

    [DataMember(Order = 2)]
    public string Name { get; set; }

    [DataMember(Order = 3)]
}
```

```
public decimal Price { get; set; }
}
```

### Explanation of the Example

- **DataContract:** The [DataContract] attribute indicates that the Product class is a data contract that will be serialized.
- **DataMember:** Each property in the Product class is decorated with the [DataMember] attribute. This tells WCF to include these properties when serializing and deserializing the object. The Order property specifies the order in which members should appear in the serialized output.

### Using the Data Contract

Data contracts are typically used in conjunction with service contracts. When defining a WCF service, you would use the data contract as part of the messages sent between the client and the service.

#### Example of Using Data Contracts in a Service Contract

Here's how you might use the Product data contract in a service contract:

```
[ServiceContract]
public interface IProductService
{
    [OperationContract]
    Product GetProduct(int id);

    [OperationContract]
    void AddProduct(Product product);
}
```

### Key Points About Data Contracts

1. **Serialization Control:** Data contracts provide control over how data is serialized, allowing you to specify which members to include and their order.
2. **Versioning:** You can evolve data contracts over time by adding new members. Existing members will still be serialized correctly for clients expecting the old version.
3. **Interoperability:** Data contracts ensure that the data can be easily understood and processed by clients written in different programming languages.

### Summary

- A **Data Contract** defines the data types used in messages exchanged between a client and a WCF service.
- It is created using the [DataContract] and [DataMember] attributes in C#.
- Data contracts control serialization, support versioning, and promote interoperability.

### XML:

**XML (eXtensible Markup Language)** is a markup language used to store and transport data. It is both human-readable and machine-readable, making it an ideal choice for representing structured information. XML is widely used in various applications, including web services, configuration files, and data interchange between systems.

### Key Features of XML

1. **Self-Describing:** XML data is structured in a way that makes it easy to understand. The tags define the data, so the structure and meaning of the data are clear.
2. **Hierarchical Structure:** XML documents have a tree-like structure, with a single root element and nested child elements. This makes it easy to represent complex data relationships.
3. **Flexibility:** XML is extensible, meaning you can create custom tags to suit your needs. You can define the structure of your data without being limited by predefined tags.
4. **Platform-Independent:** XML is text-based and can be used across different systems and platforms, making it suitable for data interchange.

### Basic Structure of an XML Document

An XML document consists of the following components:

1. **Declaration:** The XML declaration specifies the version of XML being used and optionally the encoding.
2. **Root Element:** The root element is the top-level element that contains all other elements.
3. **Child Elements:** These are the nested elements that contain data or further structure.
4. **Attributes:** Elements can have attributes that provide additional information about the element.

### Example of an XML Document

Here's a simple example of an XML document representing a list of books:

```
<?xml version="1.0" encoding="UTF-8"?>
<library>
  <book id="1">
    <title>The Great Gatsby</title>
    <author>F. Scott Fitzgerald</author>
    <price>10.99</price>
    <publishDate>1925-04-10</publishDate>
  </book>
  <book id="2">
    <title>1984</title>
    <author>George Orwell</author>
    <price>9.99</price>
    <publishDate>1949-06-08</publishDate>
  </book>
</library>
```

## Explanation of the Example

- **Declaration:** <?xml version="1.0" encoding="UTF-8"?> specifies the XML version and encoding used.
- **Root Element:** <library> is the root element containing all other elements.
- **Child Elements:** Each <book> element contains nested elements like <title>, <author>, <price>, and <publishDate>.
- **Attributes:** The <book> element has an attribute id that provides an identifier for the book.

## XML vs. Other Data Formats

- **XML vs. JSON:** Both XML and JSON are used for data interchange, but JSON is often preferred for web APIs due to its simpler syntax and smaller size. XML is more verbose but offers more features like attributes and mixed content.
- **XML vs. CSV:** XML is hierarchical and supports complex structures, while CSV (Comma-Separated Values) is flat and best for simple tabular data.

## Common Uses of XML

1. **Web Services:** XML is commonly used in SOAP web services for message formatting.
2. **Configuration Files:** Many applications use XML files to store configuration settings.
3. **Data Interchange:** XML is widely used for exchanging data between different systems and applications.

## Summary

- **XML** is a flexible and self-describing markup language used to store and transport structured data.
- It has a hierarchical structure with a root element and nested child elements.
- XML is platform-independent and widely used in web services, configuration files, and data interchange.

## WCF binding:

### What are WCF Bindings?

**WCF bindings** are like rules that tell your WCF service how to communicate with other applications (like clients). They define things like how data is sent, what formats to use, and how to keep the data secure.

### Types of WCF Bindings

1. **BasicHttpBinding:**
  - **What It Is:** A simple way to send messages over the web.
  - **When to Use:** Good for basic web services that work with older services (like ASMX).
2. **WsHttpBinding:**
  - **What It Is:** A more advanced way to send messages that supports extra features.
  - **When to Use:** Ideal for services that need better security and reliability.
3. **NetTcpBinding:**

- **What It Is:** A fast way to send messages over a network using TCP.
- **When to Use:** Best for applications on the same network (like a company's internal apps).

#### 4. **NetNamedPipeBinding:**

- **What It Is:** A very fast way to send messages between applications on the same computer.
- **When to Use:** Good for services that run in the same place and need quick communication.

#### 5. **NetMsmqBinding:**

- **What It Is:** A way to send messages using a queue, which allows messages to be stored until the receiver is ready.
- **When to Use:** Best for situations where messages need to be delivered reliably, even if the service is offline.

#### 6. **WebHttpBinding:**

- **What It Is:** A way to create services that use web standards like REST.
- **When to Use:** Great for building APIs that can send data in formats like JSON or XML.

### **Example of a WCF Binding Configuration**

Here's a simple example of setting up a WCF service using WsHttpBinding in a configuration file:

```
<system.serviceModel>
  <services>
    <service name="YourNamespace.YourService">
      <endpoint address=""
                binding="wsHttpBinding"
                contract="YourNamespace.IYourService" />
      <host>
        <baseAddresses>
          <add baseAddress="http://localhost:8733/YourService/" />
        </baseAddresses>
      </host>
    </service>
  </services>
</system.serviceModel>
```

### **Explanation of the Example**

- **Service Name:** This tells where to find the service.
- **Endpoint:** This is where clients connect to the service. Here, we're using wsHttpBinding to send messages.
- **Base Address:** This is the URL where the service can be accessed.

### **Summary**

- **WCF bindings** are rules for how services communicate with clients.
- There are different types of bindings, each suited for specific needs (like speed, security, and data format).

- Common bindings include BasicHttpBinding, WsHttpBinding, NetTcpBinding, and more.

## ABC of WCF

Here's a simplified overview of the **ABC of WCF** (Windows Communication Foundation), covering the basics of what WCF is, its components, and key concepts:

### A - Architecture

- **What is WCF?**: WCF is a framework for building service-oriented applications. It allows different applications to communicate with each other over networks using various protocols.
- **Service-Oriented Architecture (SOA)**: WCF follows the SOA design principle, where services are independent and can be used by multiple clients.

### B - Bindings

- **What are Bindings?**: Bindings define how messages are sent and received between clients and services.
- **Types of Bindings**:
  - **BasicHttpBinding**: Simple web service communication.
  - **WsHttpBinding**: Supports advanced features like security and reliability.
  - **NetTcpBinding**: Fast communication over TCP, ideal for intranet.
  - **WebHttpBinding**: Used for RESTful services and web APIs.

### C - Contracts

- **Service Contract**: An interface that defines what operations (methods) the service offers. It uses the [ServiceContract] attribute.
- **Data Contract**: Defines the data types that the service will use. It specifies how data is structured and serialized using the [DataContract] and [DataMember] attributes.
- **Operation Contract**: Marks methods in the service contract that can be called by clients, using the [OperationContract] attribute.

### Key Concepts

1. **Endpoints**: Each WCF service has one or more endpoints, which specify the address (URL), binding (communication method), and contract (operations) that clients use to interact with the service.
2. **Hosting**: WCF services can be hosted in various environments, including:
  - **IIS (Internet Information Services)**: For web applications.
  - **Windows Services**: For background services.
  - **Self-hosting**: Running a service in a console application or other applications.
3. **Security**: WCF supports various security mechanisms to protect data during transmission, including transport security (like HTTPS) and message security (encrypting the message itself).
4. **Transactions**: WCF supports distributed transactions, allowing multiple operations to be grouped together so that they either all succeed or all fail.

5. **Error Handling:** WCF provides ways to handle errors gracefully, allowing developers to log exceptions or return friendly error messages to clients.

## Summary

- **WCF** is a framework for building services that communicate over networks.
- It uses **bindings** to define how data is transmitted and **contracts** to define what services offer.
- Key concepts include **endpoints, hosting, security, transactions, and error handling**.

## Restful services:

**RESTful services** are web services that follow the principles of **Representational State Transfer (REST)** architecture. REST is an architectural style used for designing networked applications and is commonly used for building APIs (Application Programming Interfaces) that enable communication between client and server over HTTP.

### Key Principles of REST

1. **Statelessness:**
  - Each request from a client to a server must contain all the information needed to understand and process that request.
  - The server does not store any session information about the client between requests.
2. **Resource-Based:**
  - Everything in REST is treated as a resource, identified by a unique URI (Uniform Resource Identifier).
  - Resources can be anything: data objects, services, or even collections of data.
3. **HTTP Methods:**
  - RESTful services use standard HTTP methods to perform operations on resources:
    - **GET:** Retrieve data from the server.
    - **POST:** Send data to the server to create a new resource.
    - **PUT:** Update an existing resource.
    - **DELETE:** Remove a resource from the server.
4. **Representations:**
  - Resources can be represented in various formats (e.g., JSON, XML, HTML).
  - Clients can request the desired format using the Accept header in HTTP requests.
5. **Stateless Communication:**
  - Each request from the client to the server must contain all the information needed to complete the request, allowing for improved scalability.

### Example of a RESTful API

Here's an example of a simple RESTful API for managing a collection of books:

## Endpoints

1. **Get all books:**
  - o **Method:** GET
  - o **URL:** /api/books
  - o **Response:** A list of books in JSON format.
2. **Get a specific book:**
  - o **Method:** GET
  - o **URL:** /api/books/{id}
  - o **Response:** Details of the book with the specified ID.
3. **Add a new book:**
  - o **Method:** POST
  - o **URL:** /api/books
  - o **Request Body:** JSON representation of the new book.
  - o **Response:** Confirmation of the book created.
4. **Update an existing book:**
  - o **Method:** PUT
  - o **URL:** /api/books/{id}
  - o **Request Body:** JSON representation of the updated book.
  - o **Response:** Confirmation of the book updated.
5. **Delete a book:**
  - o **Method:** DELETE
  - o **URL:** /api/books/{id}
  - o **Response:** Confirmation of the book deleted.

### Example of a RESTful Service Implementation in C#

Here's a simple example of how you might implement a RESTful service using ASP.NET Core:

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;

[Route("api/[controller]")]
[ApiController]
public class BooksController : ControllerBase
{
    private static List<Book> books = new List<Book>();

    // GET: api/books
    [HttpGet]
    public ActionResult<List<Book>> GetBooks()
    {
        return books;
    }
}
```

```
// GET: api/books/1
[HttpGet("{id}")]
public ActionResult<Book> GetBook(int id)
{
    var book = books.Find(b => b.Id == id);
    if (book == null) return NotFound();
    return book;
}

// POST: api/books
[HttpPost]
public ActionResult<Book> CreateBook(Book book)
{
    books.Add(book);
    return CreatedAtAction(nameof(GetBook), new { id = book.Id }, book);
}

// PUT: api/books/1
[HttpPut("{id}")]
public IActionResult UpdateBook(int id, Book book)
{
    var existingBook = books.Find(b => b.Id == id);
    if (existingBook == null) return NotFound();

    existingBook.Title = book.Title;
    existingBook.Author = book.Author;
    return NoContent();
}

// DELETE: api/books/1
[HttpDelete("{id}")]
public IActionResult DeleteBook(int id)
{
    var book = books.Find(b => b.Id == id);
    if (book == null) return NotFound();

    books.Remove(book);
    return NoContent();
}

public class Book
{
```

```

public int Id { get; set; }
public string Title { get; set; }
public string Author { get; set; }
}

```

### Benefits of RESTful Services

- **Simplicity:** RESTful services are easy to use and understand, following standard HTTP methods.
- **Scalability:** Statelessness allows for better scalability since the server does not need to manage client sessions.
- **Flexibility:** Resources can be represented in multiple formats (JSON, XML, etc.).
- **Interoperability:** RESTful services can be consumed by any client that can send HTTP requests.

### Summary

- **RESTful services** are web services that use standard HTTP methods to communicate and are based on resources identified by URIs.
- They follow principles like statelessness, resource-based communication, and use of representations.
- Common HTTP methods used include GET, POST, PUT, and DELETE.

## Consuming rest services(CRUD Operations) using jquery,AJAX And JSON:

Here's a simple guide on how to use **jQuery** and **AJAX** to consume **RESTful services** for **CRUD** (Create, Read, Update, Delete) operations using **JSON**. This will help you manage data easily, such as adding, retrieving, updating, and deleting records.

### What Are CRUD Operations?

CRUD operations are the basic tasks you can do with data:

1. **Create:** Add new data.
2. **Read:** Get existing data.
3. **Update:** Change existing data.
4. **Delete:** Remove data.

### Steps to Use jQuery, AJAX, and JSON

#### 1. Setup Your HTML

First, make sure you include jQuery in your HTML file. Here's a simple example:

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">

```

```

<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>CRUD with jQuery</title>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.6.0/jquery.min.js"></script>
</head>
<body>
  <h1>Book Management</h1>
  <!-- Add your UI elements here -->
  <script>
    // Place your jQuery code here
  </script>
</body>
</html>

```

### Example API Endpoints

Assume you have a RESTful API for managing books with these endpoints:

- **GET /api/books**: Get all books.
- **GET /api/books/{id}**: Get a book by its ID.
- **POST /api/books**: Add a new book.
- **PUT /api/books/{id}**: Update a book.
- **DELETE /api/books/{id}**: Delete a book.

### CRUD Operations Using jQuery and AJAX

#### 2. Read (Get All Books)

To get a list of all books, you can use the following code:

```

function getAllBooks() {
  $.ajax({
    url: '/api/books', // URL of the API endpoint
    method: 'GET', // Use GET method to retrieve data
    success: function(data) {
      console.log(data); // Display the list of books in the console
    },
    error: function(error) {
      console.error('Error fetching books:', error); // Handle errors
    }
  });
}

// Call the function to fetch books
getAllBooks();

```

### 3. Read (Get a Book by ID)

To get a specific book by its ID:

```
function getBookById(id) {
  $.ajax({
    url: '/api/books/' + id, // URL with the book ID
    method: 'GET', // Use GET method to retrieve data
    success: function(data) {
      console.log(data); // Display the book data
    },
    error: function(error) {
      console.error('Error fetching book:', error); // Handle errors
    }
  });
}

// Call the function to fetch a book with ID 1
getBookById(1); // Replace 1 with the desired book ID
```

### 4. Create (Add a New Book)

To add a new book to the collection:

```
function addBook(book) {
  $.ajax({
    url: '/api/books', // URL of the API endpoint
    method: 'POST', // Use POST method to add data
    contentType: 'application/json', // Specify that you're sending JSON data
    data: JSON.stringify(book), // Convert the book object to JSON format
    success: function(data) {
      console.log('Book created:', data); // Confirmation of creation
      getAllBooks(); // Refresh the list of books
    },
    error: function(error) {
      console.error('Error creating book:', error); // Handle errors
    }
  });
}

// Example book object to add
const newBook = {
  title: "New Book Title", // Title of the new book
  author: "Author Name" // Author of the new book
};
```

```
// Call the function to add the new book
addBook(newBook);
```

## 5. Update (Edit an Existing Book)

To update an existing book:

```
function updateBook(id, updatedBook) {
  $.ajax({
    url: '/api/books/' + id, // URL with the book ID
    method: 'PUT', // Use PUT method to update data
    contentType: 'application/json', // Specify that you're sending JSON data
    data: JSON.stringify(updatedBook), // Convert the updated book object to JSON
    success: function(data) {
      console.log('Book updated:', data); // Confirmation of update
      getAllBooks(); // Refresh the list of books
    },
    error: function(error) {
      console.error('Error updating book:', error); // Handle errors
    }
  });
}
```

```
// Example updated book object
```

```
const updatedBook = {
  title: "Updated Book Title", // New title for the book
  author: "Updated Author Name" // New author for the book
};
```

```
// Call the function to update a book with ID 1
```

```
updateBook(1, updatedBook); // Replace 1 with the desired book ID
```

## 6. Delete (Remove a Book)

To delete a book:

```
function deleteBook(id) {
  $.ajax({
    url: '/api/books/' + id, // URL with the book ID
    method: 'DELETE', // Use DELETE method to remove data
    success: function(data) {
      console.log('Book deleted:', data); // Confirmation of deletion
      getAllBooks(); // Refresh the list of books
    },
    error: function(error) {
```

```

        console.error('Error deleting book:', error); // Handle errors
    }
});
}

// Call the function to delete a book with ID 1
deleteBook(1); // Replace 1 with the desired book ID

```

### Summary

- **GET** requests are used to read data (Get all books or Get a book by ID).
- **POST** requests are used to create new data (Add a new book).
- **PUT** requests are used to update existing data (Edit a book).
- **DELETE** requests are used to remove data (Delete a book).

## Introduction to Web API

A **Web API** (Application Programming Interface) is a set of rules and protocols that allows different software applications to communicate with each other over the web. It enables developers to access the functionality of a service or application using standard web protocols. Here's a breakdown of the key concepts and components of Web APIs.

### What is a Web API?

- **Interface:** A Web API acts as a bridge between different software systems. It allows one application to communicate with another, regardless of the programming language or platform they are built on.
- **HTTP Protocol:** Web APIs typically use the HTTP protocol for communication. This means you can use standard web methods like GET, POST, PUT, and DELETE to interact with the API.
- **Data Formats:** Web APIs usually exchange data in formats like JSON (JavaScript Object Notation) or XML (eXtensible Markup Language), making it easy to read and parse.

### Key Components of a Web API

1. **Endpoints:** An endpoint is a specific URL where the API can be accessed. Each endpoint corresponds to a particular resource or action (e.g., retrieving a list of books).
2. **HTTP Methods:**
  - **GET:** Retrieve data from the server (e.g., get a list of users).
  - **POST:** Send data to the server to create a new resource (e.g., add a new user).
  - **PUT:** Update an existing resource (e.g., update user information).
  - **DELETE:** Remove a resource from the server (e.g., delete a user).
3. **Request and Response:**
  - **Request:** When a client makes a call to a Web API, it sends an HTTP request to a specific endpoint, which may include parameters and data.
  - **Response:** The server processes the request and sends back an HTTP response, which contains the requested data or confirmation of the action taken.

4. **Status Codes:** Each response from a Web API includes an HTTP status code that indicates the result of the request. Common status codes include:
- **200 OK:** The request was successful.
  - **201 Created:** A new resource was successfully created.
  - **400 Bad Request:** The request was invalid.
  - **404 Not Found:** The requested resource could not be found.
  - **500 Internal Server Error:** An error occurred on the server.

### Benefits of Using Web APIs

- **Interoperability:** Web APIs allow different applications and services to work together, regardless of their technology stacks.
- **Scalability:** APIs can be designed to handle a large number of requests, making it easy to scale applications as needed.
- **Reusability:** APIs enable developers to reuse existing services, reducing the need to build everything from scratch.
- **Integration:** APIs can connect to various third-party services, such as payment gateways, social media platforms, and data analytics tools.

### Examples of Web APIs

- **RESTful APIs:** These APIs follow REST (Representational State Transfer) principles, which emphasize stateless interactions and a uniform interface. They typically use JSON for data exchange.
- **SOAP APIs:** These APIs use the Simple Object Access Protocol (SOAP) for communication. They are more rigid and require a specific message structure, often using XML.
- **GraphQL APIs:** A modern API design that allows clients to request specific data and aggregate responses in a single request.

## WEB API USING CRUD EXAMPLE

Here's a simple explanation of how to create a **Web API** that can manage a collection of **books** using **CRUD** operations (Create, Read, Update, Delete). This example uses **ASP.NET Core**.

### What is a Web API?

A Web API allows different applications to talk to each other over the internet. In this case, we will build an API to manage books, so other applications can add, get, update, and delete book information.

### Step-by-Step Guide

#### Step 1: Create the Project

1. **Open Visual Studio** and create a new project.
2. Choose **ASP.NET Core Web Application**.

3. Select the **API** template to create a Web API project.

### Step 2: Define the Book Model

This model represents a book with some properties:

```
// Book.cs
public class Book
{
    public int Id { get; set; }      // Unique ID for the book
    public string Title { get; set; } // Title of the book
    public string Author { get; set; } // Author of the book
    public string ISBN { get; set; } // ISBN number of the book
}
```

### Step 3: Create a Data Store

We'll use a simple list to store the books in memory. In real applications, you would use a database.

```
// BookStore.cs
using System.Collections.Generic;
using System.Linq;

public static class BookStore
{
    private static List<Book> books = new List<Book>(); // List to store books
    private static int nextId = 1; // ID counter

    public static List<Book> GetAllBooks() => books; // Get all books

    public static Book GetBook(int id) => books.FirstOrDefault(b => b.Id == id); // Get a book by ID

    public static void AddBook(Book book)
    {
        book.Id = nextId++; // Assign a new ID to the book
        books.Add(book); // Add the book to the list
    }

    public static void UpdateBook(Book book)
    {
        var existingBook = GetBook(book.Id); // Find the book to update
        if (existingBook != null)
        {
            existingBook.Title = book.Title;
            existingBook.Author = book.Author;
        }
    }
}
```

```

        existingBook.ISBN = book.ISBN;
    }
}

public static void DeleteBook(int id)
{
    var book = GetBook(id); // Find the book to delete
    if (book != null)
    {
        books.Remove(book); // Remove the book from the list
    }
}
}

```

#### Step 4: Create the Books Controller

This controller will handle the requests for books.

```

// BooksController.cs
using Microsoft.AspNetCore.Mvc;

[Route("api/[controller]")]
[ApiController]
public class BooksController : ControllerBase
{
    [HttpGet]
    public ActionResult<List<Book>> GetAllBooks()
    {
        return BookStore.GetAllBooks(); // Return all books
    }

    [HttpGet("{id}")]
    public ActionResult<Book> GetBook(int id)
    {
        var book = BookStore.GetBook(id);
        if (book == null)
        {
            return NotFound(); // Return 404 if the book is not found
        }
        return book; // Return the found book
    }

    [HttpPost]
    public ActionResult<Book> AddBook(Book book)
    {

```

```

BookStore.AddBook(book); // Add a new book
return CreatedAtAction(nameof(GetBook), new { id = book.Id }, book); // Return the created book
}

[HttpPut("{id}")]
public IActionResult UpdateBook(int id, Book book)
{
    if (id != book.Id)
    {
        return BadRequest(); // Return 400 if IDs don't match
    }
    BookStore.UpdateBook(book); // Update the book
    return NoContent(); // Return 204 No Content
}

[HttpDelete("{id}")]
public IActionResult DeleteBook(int id)
{
    BookStore.DeleteBook(id); // Delete the book
    return NoContent(); // Return 204 No Content
}

```

### Step 5: Configure the Application

In your project, make sure to configure it to use controllers. This is usually done in the Startup.cs file:

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers(); // Add support for controllers
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage(); // Show detailed errors in development
    }

    app.UseRouting(); // Enable routing

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers(); // Map controller endpoints
    });
}

```

}

### Step 6: Testing the API

You can test the API using tools like **Postman**. Here are examples of how to use it:

#### 1. Create a Book (POST)

##### Request:

```
POST /api/books  
Content-Type: application/json
```

```
{  
  "title": "The Great Gatsby",  
  "author": "F. Scott Fitzgerald",  
  "ISBN": "9780743273565"  
}
```

**Response:** The API will return the created book with a status of 201 Created.

#### 2. Get All Books (GET)

##### Request:

```
GET /api/books
```

**Response:** The API will return a list of all books.

#### 3. Get a Book by ID (GET)

##### Request:

```
GET /api/books/1
```

**Response:** The API will return the book with ID 1.

#### 4. Update a Book (PUT)

##### Request:

```
PUT /api/books/1  
Content-Type: application/json
```

```
{  
    "id": 1,  
    "title": "The Great Gatsby - Updated",  
    "author": "F. Scott Fitzgerald",  
    "ISBN": "9780743273565"  
}
```

**Response:** The API will return 204 No Content, meaning the update was successful.

## 5. Delete a Book (DELETE)

### Request:

```
DELETE /api/books/1
```

**Response:** The API will return 204 No Content, indicating the book has been deleted.

### Summary

This example shows how to create a simple Web API to manage books using ASP.NET Core. You can create, read, update, and delete books through HTTP requests. This allows your applications to manage book data easily.