# Unit : 1: ASP.NET using C#.Net

## Generics

* Generics in C# are a powerful feature that allows you to create classes, interfaces, methods, and delegates that work with data types as parameters, making your code more flexible and reusable.
* Generics are used for assign data type value dynamically to class or interface
  + Code for define generic:-

**public class SimpleGenericClass<T>**

**{**

**public T Field;**

**}**

* + Call this genrics into our main code :

**public static void Main()**

**{**

**SimpleGenericClass<string> g = new SimpleGenericClass<string>();**

**g.Field = "A string";**

**//...**

**Console.WriteLine("SimpleGenericClass.Field = \"{0}\"", g.Field);**

**Console.WriteLine("SimpleGenericClass.Field.GetType() = {0}", g.Field.GetType().FullName);**

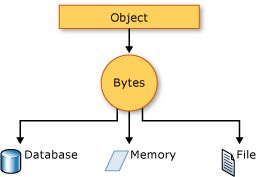
**}**

## Serialization in C#

Serialization is the process of converting an object into a stream of bytes to store the object or transmit it to memory, a database, or a file. Its main purpose is to save the state of an object in order to be able to recreate it when needed. The reverse process is called deserialization.

### How serialization works

This illustration shows the overall process of serialization.



The object is serialized to a stream, which carries not just the data, but information about the object's type, such as its version, culture, and assembly name. From that stream, it can be stored in a database, a file, or memory.

### Uses for serialization

Serialization allows the developer to save the state of an object and recreate it as needed, providing storage of objects as well as data exchange. Through serialization, a developer can perform actions like sending the object to a remote application by means of a Web Service, passing an object from one domain to another, passing an object through a firewall as an XML string, or maintaining security or user-specific information across applications.

-  use **BinaryFormatter.Serialize(stream, reference)** method to serialize the object.

1. **using** System;
2. **using** System.IO;
3. **using** System.Runtime.Serialization.Formatters.Binary;
4. **[Serializable]**
5. **class** Student
6. {
7. **int** rollno;
8. **string** name;
9. **public** Student(**int** rollno, **string** name)
10. {
11. **this**.rollno = rollno;
12. **this**.name = name;
13. }
14. }
15. **public** **class** SerializeExample
16. {
17. **public** **static** **void** Main(**string**[] args)
18. {
19. FileStream stream = **new** FileStream("e:\\sss.txt", FileMode.OpenOrCreate);
20. BinaryFormatter formatter=**new** BinaryFormatter();
22. Student s = **new** Student(101, "sonoo");
23. formatter.Serialize(stream, s);
25. stream.Close();
26. }
27. }

* Type of serialization
* Binary Serialization(uses using system.runtime.formatters.binary)
* XML Serialization (serializes only public properties and fields and does not preserve type fidelity)
* JSON Serialization ( serializes public properties by default, and can be configured to serialize private and internal members as well.)

## Globalization and Localization

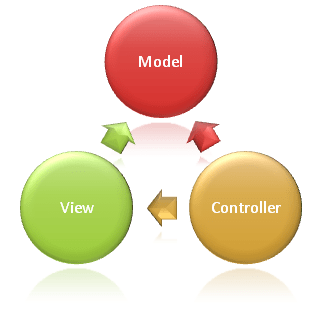
"Globalization is the process of designing and developing a software product that functions in multiple cultures/locales."

"Localization is the process of adapting a globalized application, which you have already processed for localizability, to a particular culture/locale."

# Unit : 2 : ASP.NET MVC

## MVC Architectural Pattern

The Model-View-Controller (MVC) architectural pattern separates an application into three main groups of components: Models, Views, and Controllers. This pattern helps to achieve separation of concerns. Using this pattern, user requests are routed to a Controller which is responsible for working with the Model to perform user actions and/or retrieve results of queries. The Controller chooses the View to display to the user, and provides it with any Model data it requires.



MVC is a software architecture pattern which follows the separation of concerns method. In this model .Net applications are divided into three interconnected parts which are called Model, View, and Controller.

The goal of the MVC pattern is that each of these parts can be developed, tested in relative isolation and also combined to create a very robust application.

Let see all of them in detail:

**Models**

Model objects are parts of the application which implement the logic for the application's **data domain**. It retrieves and stores model state in a database. For example, product object might retrieve information from a database, operate on it. Then write information back to products table in the SQL server.

**Views**

View are the components which are used to display the application's user interface (UI). It displays the .Net MVC application's which is created from the model data.

The common example would be an edit view of an Item table. It displays text boxes, pop-ups and checks boxes based on the current state of products & object.

**Controller**

Controllers handle user interaction, work with the model, and select a view to render that displays Ul. In a .Net MVC app, the view only displays information, the controller manages and responds to user input & interaction.

For example, the controller manages query-string values and passes those values to the model.

1. MVC stands for Model, View and Controller.
2. Model is responsible for maintaining application data and business logic.
3. View is a user interface of the application, which displays the data.
4. Controller handles user's requests and renders appropriate View with Model data.

## MVC Framework - Routing Engine

ASP.NET MVC Routing enables the use of URLs that are descriptive of the user actions and are more easily understood by the users. At the same time, Routing can be used to hide data which is not intended to be shown to the final user.

**MVC Routing URL Format**

To understand the MVC routing, consider the following URL −

http://servername/Products/Phones

In the above URL, Products is the first segment and Phone is the second segment which can be expressed in the following format −

*{controller}/{action}*

## explain wiring controller model and view in asp.net mvc

In ASP.NET MVC (Model-View-Controller), the architecture is designed to separate concerns, making it easier to develop and maintain web applications. It divides the application into three main components: Model, View, and Controller. Here's an explanation of how these components are wired together:

1. \*\*Model\*\*:

The Model represents the application's data and business logic. It encapsulates the application's state and defines how data is retrieved, processed, and updated. Models are responsible for data access, validation, and manipulation. Typically, they correspond to entities or objects in your application.

- \*\*Role\*\*: The Model communicates with the database, external APIs, or other data sources to retrieve and manipulate data. It provides data to the Controller and is responsible for data validation and business rules.

2. \*\*View\*\*:

The View represents the user interface and the presentation of the data. It is responsible for rendering the data to the user and presenting it in a readable and user-friendly format. Views are typically created using HTML, along with embedded code or templates to display data.

- \*\*Role\*\*: The View is responsible for displaying data to the user and capturing user input. It doesn't contain application logic but is aware of the Model and the Controller.

3. \*\*Controller\*\*:

The Controller acts as an intermediary between the Model and the View. It receives user input from the View, processes it, interacts with the Model to retrieve or update data, and then selects the appropriate View to render the response. Controllers contain the application's logic for handling requests and orchestrating the flow of data between the Model and the View.

- \*\*Role\*\*: The Controller processes user requests, interacts with the Model to obtain or modify data, and selects the appropriate View to display the response. It handles the application's business logic and controls the flow of the application.

The wiring between these components in an ASP.NET MVC application typically follows these steps:

1. \*\*User Interaction\*\*:

When a user interacts with the application, such as entering a URL or submitting a form, the request is sent to the appropriate Controller.

2. \*\*Controller Processing\*\*:

The Controller receives the request, processes it, and interacts with the Model to obtain or update data. It may perform tasks like data validation, authorization, and other business logic.

3. \*\*Model Interaction\*\*:

The Controller interacts with the Model to retrieve or update data. The Model handles data operations, such as database queries, and applies business rules or validation.

4. \*\*View Selection\*\*:

After processing the request and obtaining data from the Model, the Controller selects the appropriate View to render the response. It passes data to the View for rendering.

5. \*\*View Rendering\*\*:

The selected View takes the data provided by the Controller and renders the HTML response to be sent back to the user's browser.

6. \*\*User Response\*\*:

The HTML response is sent to the user's browser, and the user interacts with the application through the View. User input and actions trigger new requests, and the process repeats.

This separation of concerns in ASP.NET MVC allows for better organization of code, improved maintainability, and more straightforward collaboration among developers. It also enables unit testing of individual components (Controller and Model) without requiring a complete end-to-end test.

**// just for knowledge**

**User request -> controller process with model and create actions as per request -> both send their outputs to customer side -> on view section output will render.**

## TempData, ViewBag and ViewData

## **What is ViewBag?**

ViewBag is a **dynamic object to passes the data from the Controller to View**. This will pass the data as a property of the object ViewBag. And we have no need to typecast to read the data or for null checking. The scope of ViewBag is permitted to the current request, and the value of ViewBag will become null while redirecting.

* It passes data as a property of the object viewbag
* It will null while redirecting

**Example Controller**

Public ActionResult Index()

{

ViewBag.Title = “Welcome”;

return View();

}

C#

Copy

**Example View**

<[h2](https://webplatform.github.io/docs/html/elements/h2)>@ViewBag.Title</[h2](https://webplatform.github.io/docs/html/elements/h2)>

Markup

Copy

## **What is ViewData?**

ViewData is a dictionary object to pass the data from Controller to View, where data is passed in the form of a key-value pair. Typecasting is required to read the data in View if the data is complex, and we need to ensure a null check to avoid null exceptions. The scope of ViewData is similar to ViewBag, and it is restricted to the current request, and the value of ViewData will become null while redirecting.

**Example Controller**

Public ActionResult Index()

{

ViewData[”Title”] = “Welcome”;

return View();

}

C#

Copy

**Example View**

<[h2](https://webplatform.github.io/docs/html/elements/h2)>@ViewData[“Title”]</[h2](https://webplatform.github.io/docs/html/elements/h2)>

Markup

Copy

## **What is TempData?**

TempData is a dictionary object to passes the data from one action to another action in the same Controller or different Controllers. Usually, the TempData object will be stored in a session object. Tempdata is also required to typecast and for null checking before reading data from it. TempData scope is limited to the next request, and if we want TempData to be available even further, we should use Keep and Peek.

**Example Controller**

Public ActionResult Index()

{

TempData[”Data”] = “I am from Index action”;

return View();

}

Public string Get()

{

return TempData[”Data”] ;

}

C#

Copy

To summarize, ViewBag and ViewData are used to pass the data from a Controller action to View, and TempData is used to pass the data from action to another action or one Controller to another Controller.

View bag and view data -> controller to view

Tempdata -> controller to controller

## Razor view

Razor View engine is a markup syntax which helps us to write HTML and server-side code in web pages using C# or VB.NET. It is server-side markup language however it is not at all a programming language.

Razor is a templating engine and ASP.NET MVC has implemented a view engine which allows us to use Razor inside of an MVC application to produce HTML. However, Razor does not have any ties with ASP.NET MVC.

# Unit : 4 : Programming using C#.Net CORE

## Overview of C#.Net CORE

NET Core is a free, open-source, modular, cross-platform framework developed by Microsoft for building modern, high-performance applications. It is a reimplementation of the .NET Framework to be more lightweight, modular, and flexible.

.NET Core includes a runtime, a set of libraries, and a development environment that supports multiple programming languages such as C#, Visual Basic, and F#. It is designed to run on various platforms, including Windows, macOS, and Linux. It can be used to build a wide range of applications, including web applications, desktop applications, and microservices.

One of the key benefits of .NET Core is that it is modular, which means you can use only the components you need for your application, making it more lightweight and faster. Additionally, .NET Core includes many performance improvements over the .NET Framework, including faster startup times and lower memory usage.

## Characteristics of .NET Core

* **.NET Core is Free and Open Source.**
* **.NET Core is Cross-platform.**
* **.NET Core is Sharable.**
* **.NET Core is Modern.**
* **.NET Core is Fast.**
* **.NET Core is Lightweight.**
* **.NET Core is Friendly.**
* Rich Ecosystem
* (IDEs) – vs code and visual studio code
* Support

## Assemblies

Assemblies are libraries that could be shared. In addition to normal DLLs, assemblies contain extensible metadata with information about the library and a version number, and it’s possible to install multiple versions side by side in the global assembly cache. Microsoft tried to fix versioning issues, but this added another layer of complexity.

An assembly is a library or executable that includes additional metadata. Using .NET Core, the application containing the Main method is created as a library with the file extension .dll. This DLL needs a hosting process to load this library, which you do using .net run, or just .net from a runtime environment When you create standalone applications with .NET Core, different executables are created for every platform to load the library.

## Libraries

Libraries make it possible for you to reuse code in multiple applications. With Windows, libraries have a long history and architecture guidelines have taken different directions with newer technologies.

## Pattern Matching

**Pattern Matching** is a new feature which was introduced with **C#** 7.0 which allows us to write cleaner and concise code in many different scenarios. This feature can be said as an extension of is and as operators that we already have in **C#**.

Pattern matching is introduced in C# 7.0 to bring more logical ability in the language. Patterns are not new in the programming languages however it’s introduced first time in C# with the limited features and more is expected to come in the upcoming versions.

Pattern matching is useful in many ways however the cases are given below, which C# 7.0 currently supports.

* Pattern matching can be used with any data type including custom whereas if/else can only be used with primitive types.
* Pattern matching has the ability to extract the data from the expression.

With pattern matching in C# 7, the **is operator** and the **switch statement** have been enhanced with three kinds of patterns:

1. **The** ***const pattern***
2. **The *type pattern***
3. **The *var pattern*.**

**Program:**

static void Main()

{

var p1 = new Person("Katharina", "Nagel");

var p2 = new Person("Matthias", "Nagel");

var p3 = new Person("Stephanie", "Nagel");

object[] data = { null, 42, "astring", p1, new Person[] { p2, p3 } };

foreach (var item in data)

{

IsOperator(item);

}

foreach (var item in data)

{

SwitchStatement(item);

}

}

**Pattern Matching with the is Operator**

1. A simple match with the **is operator** is the ***const pattern***. With this pattern, you can compare an object to constant values such as null or 42

static void IsOperator(object item)

{

// const pattern

if **(item is null)**

{

Console.WriteLine("item is null");

}

if **(item is 42)**

{

Console.WriteLine("item is 42");

}

//...

}

Ans:

item is null

item is 42

1. The most interesting pattern match is the *type pattern*. With this pattern you can match for a specific type, such as int or string. This pattern also enables you to declare a variable, such as if (item is int i). The variable i is assigned to the item if the pattern applies:

static void IsOperator(object item)

{

//...

// type pattern

if **(item is int)**

{

Console.WriteLine($"Item is of type int");

}

if **(item is int i)**

{

Console.WriteLine($"Item is of type int with the value {i}");

}

if **(item is string s)**

{

Console.WriteLine($"Item is a string: {s}");

}

//...

}

Ans:

Item is of type int

Item is of type int with a value 42

Item is a string: astring

Declaring a variable of the type allows strongly typed access. You can access all the members of the type without the need for a cast. This also allows using logical operators in the if statement to check for other constraints than just the type, such as if the FirstName starts with the string Ka:

static void IsOperator(object item)

{

//...

if (**item is Person p && p.FirstName.StartsWith("Ka")**)

{

Console.WriteLine($"Item is a person: {p.FirstName} {p.LastName}");

}

}

Ans:

Item is a person: Katharina Nagel

1. One more pattern type needs to be discussed: the *var pattern*. Everything can be applied to a var; you just get the concrete type. With the sample code, the GetType method is invoked to get the name of the type and to write the concrete type to the console. When the value is null, the var pattern applies as well. That’s why the null-conditional operator is used with the every variable. every is null if the item is null, which writes the
2. string null to the console:

static void IsOperator(object item)

{

//...

// var pattern

if (item is var every)

{

Console.WriteLine($"it's var of type {every?.GetType().Name ?? "null"} " +

$"with the value {every ?? "nothing"}");

}

}

Ans:

The output of the application for the var pattern shows that all items of the array match with this pattern:

it's var of type null with the value nothing

it's var of type Int32 with the value 42

it's var of type String with the value astring

it's var of type Person with the value Katharina Nagel

it's var of type Person[] with the value PatternMatching.Person[]

**Pattern Matching with the switch Statement**

With the switch statement, the three pattern types can be used as well. The following code snippet shows the const pattern with cases for null and 42; the type pattern for int, string, and Person; and the var pattern. Like the extension of the is operator, with the switch statement a variable can be specified with the type pattern to write the matching result to this variable. You also can apply an additional filter with the when

clause. The first type match for the Person class applies only when the FirstName property of the Person has the value Katharina. With the switch statement, the ordering of the cases is important. As soon as one case applies, the other cases are not checked further. If the first match to the Person type with the when clause applies, the second case for Person does not apply. That’s why when filtering must be done before general cases for a type. The var pattern that is defined with the last case matches with every object passed to the switch. However, this case is checked only if none of the other cases that are defined earlier apply. The default clause can be on every position of the switch statement, and it applies only if none of the cases match.

static void SwitchStatement(object item)

{

switch (item)

{

case null:

case 42:

Console.WriteLine("it's a const pattern");

break;

case int i:

Console.WriteLine($"it's a type pattern with int: {i}");

break;

case string s:

Console.WriteLine($"it's a type pattern with string: {s}");

break;

case Person p when p.FirstName == "Katharina":

Console.WriteLine($"type pattern match with Person and " +

$"when clause: {p}");

break;

case Person p:

Console.WriteLine($"type pattern match with Person: {p}");

break;

case var every:

Console.WriteLine($"var pattern match: {every?.GetType().Name}");

break;

default:

}

}

Ans:

When you run the application, the const pattern of the switch statement applies with null and 42, the string pattern applies with the string astring, with the Person object the first Person case applies, and finally, the Person array matches the var pattern—because no other pattern applied earlier. A match to the type pattern with the int type did not apply because the const pattern was an earlier match:

it's a const pattern

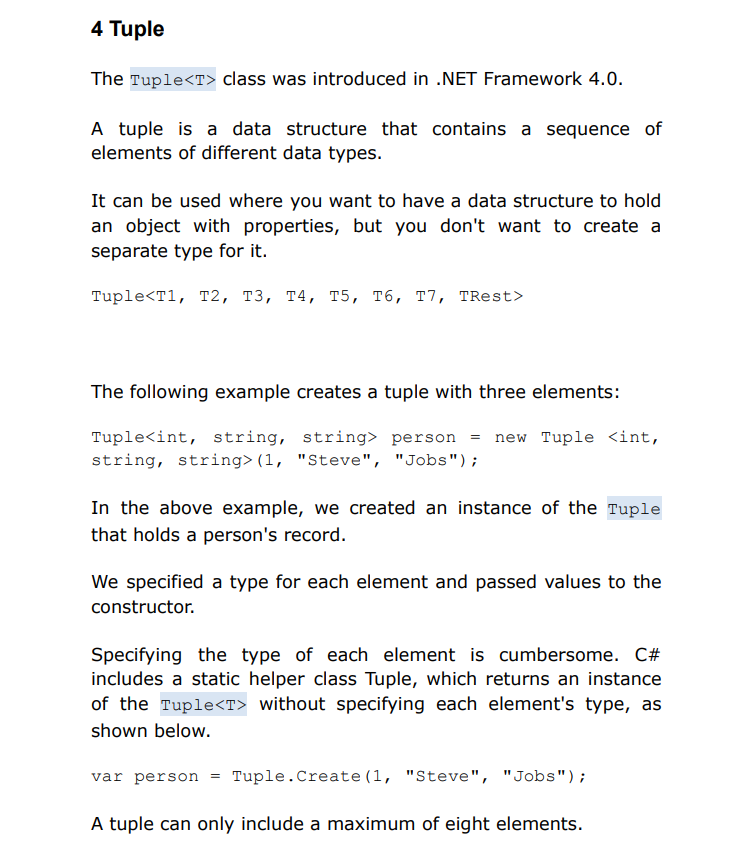
it's a const pattern

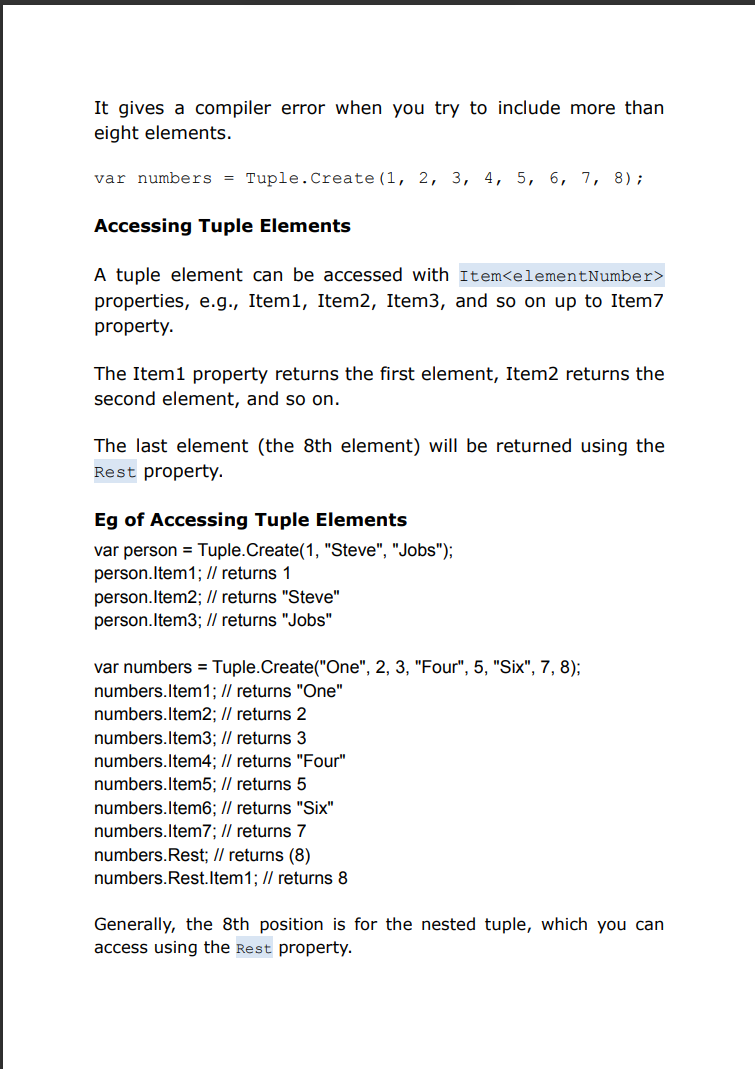
it's a type pattern with string: astring

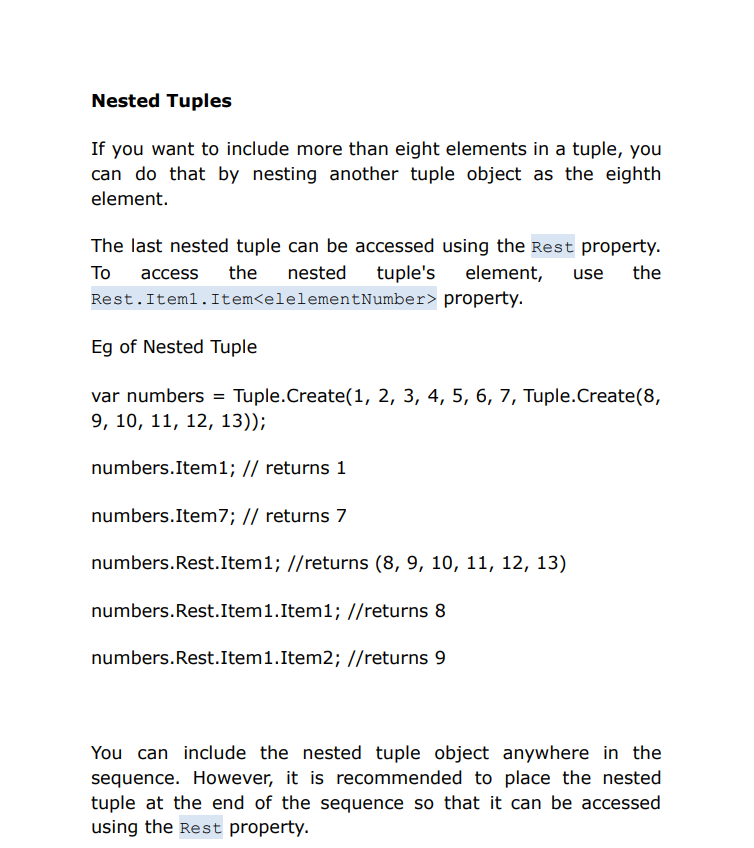
type pattern match with Person and when clause: Katharina Nagel

var pattern match: Person[]

## Tuple

****

****

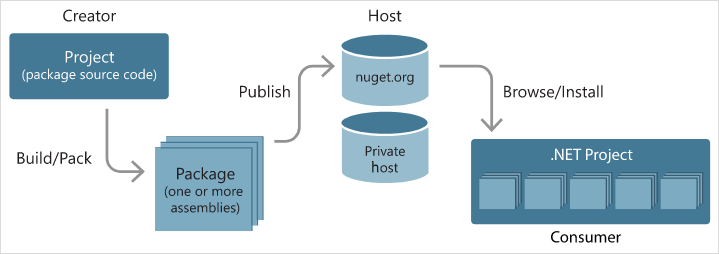
****

## NuGet Package

--> development platform is a mechanism through which developers can create, share, and consume useful code. Often such code is bundled into "packages" that contain compiled code (as DLLs) along with other content needed in the projects that consume these packages.

--> the Microsoft-supported mechanism for sharing code is NuGet, which defines how packages for .NET are created, hosted, and consumed, and provides the tools for each of those roles.

--> NuGet package is a single **ZIP file with the .nupkg extension** that contains compiled code (DLLs), other files related to that code, and a descriptive manifest that includes information like the package's version number. Developers with code to share create packages and publish them to a public or private host. Package consumers obtain those packages from suitable hosts, add them to their projects, and then call a package's functionality in their project code. NuGet itself then handles all of the intermediate details.



NuGet provides the central nuget.org repository with support for private hosting.

NuGet provides the tools developers need for creating, publishing, and consuming packages.

Most importantly, NuGet maintains a reference list of packages used in a project and the ability to restore and update those packages from that list.

NuGet does some behind-the-scenes optimizations. Most notably, NuGet manages a package cache and a global packages folder to shortcut installation and reinstallation. The cache avoids downloading a package that's already been installed on the machine.

The global packages folder allows multiple projects to share the same installed package. The cache and global packages folder are also very helpful when you're frequently restoring a larger number of packages.